Oracle® Fusion Middleware

Developing Fusion Web Applications with Oracle Application Development Framework

12*c* (12.2.1.4.0)
E80055-02
August 2020

Oracle Fusion Middleware Developing Fusion Web Applications with Oracle Application Development Framework, 12*c* (12.2.1.4.0)

E80055-02

Primary Authors: Ralph Gordon (lead), Walter Egan, Patrick Keegan, Landon Ott, Kathryn Munn, Vaibhav Gupta, Krithika Gandadhar, Sandhya Sriram

Contributors: ADF development team, Lynn Munsinger

Contents

Preface

Part I Getting Started with Fusion Web Applications

1 Introduction to Building Fusion Web Applications with Oracle ADF

2 Introduction to the ADF Sample Application

Part II Building Your Business Services

3 Getting Started with ADF Business Components

4 Creating a Business Domain Layer Using Entity Objects

5 Defining SQL Queries Using View Objects

6 Defining Master-Detail Related View Objects

7 Defining Polymorphic View Objects

8 Testing View Instance Queries

9 Tuning View Object Performance

10 Defining Validation and Business Rules Declaratively

11 Working Programmatically with View Objects

14 Sharing Application Module View Instances

15 Creating SOAP Web Services with Application Modules

Part III Using the ADF Model Layer

18 Using ADF Model in a Fusion Web Application

19 Using Validation in the ADF Model Layer

20 Designing a Page Using Placeholder Data Controls

21 Creating ADF REST Data Controls from ADF RESTful Web Services

22 Consuming ADF RESTful Web Services

Part IV Creating ADF Task Flows

23 Getting Started with ADF Task Flows

24 Working with Task Flow Activities

25 Using Parameters in Task Flows

26 Using Task Flows as Regions

27 Creating Complex Task Flows

28 Using Dialogs in Your Application

Part V Creating a Databound Web User Interface

33 Using Command Components to Invoke Functionality in the View Layer

34 Displaying Master-Detail Data

37 Creating Databound Calendar and Carousel Components

38 Creating Databound Chart, Picto Chart, and Gauge Components

42 Creating Databound Gantt Chart and Timeline Components

43 Creating Databound Hierarchy Viewer, Treemap, and Sunburst Components

44 Creating Databound Diagram Components

45 Creating Databound Tag Cloud Components

46 Using Contextual Events

Part VI Completing Your Application

47 Enabling ADF Security in a Fusion Web Application

51 Customizing Applications with MDS

52 Allowing User Customizations at Runtime

53 Using the Active Data Service

54 Configuring High Availability for Fusion Web Applications

55 Deploying Fusion Web Applications

57 Tuning Application Module Pools

Part VII Appendixes

A Oracle ADF XML Files

B Oracle ADF Binding Properties

C ADF Security Permission Grants

D Most Commonly Used ADF Business Components Methods

G Performing Common Oracle Forms Tasks in Oracle ADF

H Deploying ADF Applications to GlassFish

Preface

Welcome to *Developing Fusion Web Applications with Oracle Application Development Framework*.

Audience

This document is intended for enterprise developers who need to create and deploy database-centric Java EE applications with a service-oriented architecture using the Oracle Application Development Framework (Oracle ADF). This guide explains how to build Fusion web applications using ADF Business Components, ADF Controller, ADF Faces, and JavaServer Faces.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc`.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info` or visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs` if you are hearing impaired.

Related Documents

For more information, see the following documents:

- *Understanding Oracle Application Development Framework*
- *Developing Web User Interfaces with Oracle ADF Faces*
- *Developing Applications with Oracle ADF Desktop Integration*
- *Developing Applications with Oracle ADF Data Controls*
- *Developing Applications with Oracle JDeveloper*
- *Developing ADF Skins*
- *Administering Oracle ADF Applications*
- *Tuning Performance*
- *High Availability Guide*
- *Installing Oracle JDeveloper*
- *Oracle JDeveloper Online Help*

- *Oracle JDeveloper Release Notes*, included with your JDeveloper installation, and on Oracle Technology Network
- *Java API Reference for Oracle ADF Model*
- *Java API Reference for Oracle ADF Controller*
- *Java API Reference for Oracle ADF Lifecycle*
- *Java API Reference for Oracle ADF Faces*
- *JavaScript API Reference for Oracle ADF Faces*
- *Java API Reference for Oracle ADF Data Visualization Components*
- *Java API Reference for Oracle ADF Share*
- *Java API Reference for Oracle ADF Model Tester*
- *Java API Reference for Oracle Generic Domains*
- *Java API Reference for Oracle Metadata Service (MDS)*
- *Java API Reference for Oracle ADF Resource Bundle*
- *Tag Reference for Oracle ADF Faces*
- *Tag Reference for Oracle ADF Faces Skin Selectors*
- *Tag Reference for Oracle ADF Faces Data Visualization Tools*
- *Tag Reference for Oracle ADF Data Visualization Tools Skin Selectors*

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements (for example, menus and menu items, buttons, tabs, dialog controls), including options that you select.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates language and syntax elements, directory and file names, URLs, text that appears on the screen, or text that you enter.

What's New in This Guide for Release 12c (12.2.1.4.0)

The following topics introduce the new and changed features of Oracle JDeveloper and Oracle Application Development Framework (Oracle ADF) and other significant changes, which are described in this guide. This document in previous JDeveloper releases had been titled *Fusion Developer's Guide for Oracle Application Development Framework*.

New and Changed Features for Release 12c (12.2.1.4.0)

Oracle Fusion Middleware Release 12c (12.2.1.4.0) of Oracle JDeveloper and Oracle Application Development Framework (Oracle ADF) includes the following new and changed development features, which are described in this guide.

- ADF REST Framework Runtime and Design Time
  - New runtime support for REST resources is provided with the addition of four new ADF REST framework versions. By declaring a default framework version to process requests, the REST application developer may opt into the features introduced by a specific ADF REST framework. The default ADF REST framework version is defined by the application in the `adf-config.xml` file. In JDeveloper release 12.2.1.4.0, framework versions 4, 5, 6, and 7 were added to existing versions 1 through 3.
    - Versions 4 and later enables exception details to appear in the response when the ADF REST request is made for either `application/vnd.oracle.adf.error+json` or `application/json` media types. See Obtaining an Exception Payload Error Response.
    - Versions 5 and later supports filtering resources by using a row match finder to access nested LOV resources, for example: `/rest/v1/Cities?finder=ByState;name='California'`. Starting with this version, row context LOV URLs are no longer returned in the payload or in the describe. Only LOV resource URLs that point to top-level resources are described and included in payloads. See Retrieving LOV-Enabled Attribute Values with Framework Version 5 and Later.
    - Versions 6 and later makes it easier to differentiate between resource fields and item context information, such as links and headers. A new element `@context` is introduced in this version and all the information for an item is moved under `@context`. Version 6 also supports obtaining warning details in the response. See Fetching a Resource with Grouped Context Information and Obtaining Warning Messages in the Payload Response.
    - Version 7 (latest version in release 12.2.1.4.0) supports top-level LOVs, and removes row-level LOV resource descriptions in the ADF REST describe and removes LOV links from the resource item payload links section. Earlier framework versions support LOV in a context of a row in the REST resource. See What You May Need to Know About Versioning the ADF REST Framework.

For an overview of the features introduced by all available framework versions, see What You May Need to Know About Versioning the ADF REST Framework.

  - A variety of new ways to limit the ADF REST resource catalog describe are available. The catalog describe can now be obtained by retrieving only minimal details (a minimal describe), or by retrieving based on resource visibility—private, public, or both (as specified on the resource at design time), or by retrieving parent resources and optionally excluding or including nested, child resources. See Retrieving the ADF REST Catalog Describe.
  - A variety of new ways to describe ADF REST resources are available. The resource describe can now be obtained by retrieving two or more named resources, by using parameters `filters` or `partialDescription` to limit the resource describe, and by retrieving resources based on ETag values as assigned by the web server to identify specific versions of a resource. See Retrieving the ADF REST Resource Describe.
  - The ADF REST resource describe now identifies whether or not an attribute is mandatory using the property “`mandatory`”. The “`mandatory`” property in the resource describe is based on the view object attribute metadata specified at design time. See What You May Need to Know About Mandatory Attributes in the ADF REST Resource.
  - A new Upsert mode on a POST method lets you merge payload content to either update resource items that exist, or if not, create the resource items. See Updating or Creating Resource Items (Upsert).
  - The use of special characters in the query parameter string of an ADF REST GET request is supported by the use of single quotes (in framework version 2 and later) to define literal values. See GET Method Operations.
  - Case-sensitive sorting of resource items of type String is now supported by the `orderBy` query string parameter of an ADF REST GET request. By default, the `orderBy` clause sorts the records with uppercase letters first and then lower case letters. To achieve case-insensitive sorting, the `orderBy` clause can be combined with `upper(`*`attrName`*`)` or `lower(`*`attrName`*`)`. See Sorting a Resource Collection.
  - The ADF REST framework now supports configuration options for `Cache-Control`. You can specify these additional configuration options in the REST resource source editor. When defined on the resource, the `Cache-Control` header will be added in the payload of the ADF REST response. See What You May Need to Know About the Cache-Control Header.
  - The JDeveloper wizard Create Business Components from Tables combines generating the various business components together with creating ADF REST resources in a single end to end process. This wizard helps you to create ADF REST resources quickly and easily. See How to Create ADF REST Resources Using the Create Business Components from Tables Wizard.
  - Attribute hints defined in the JDeveloper overview editor for view objects can now optionally appear as attribute properties in the REST resource describe. See What You May Need to Know About ADF REST Attribute Hints.

- ADF Business Components
  - The ADF application now supports custom script expression classes to define a class that can be used by other script classes. A script class file allows you

    to maintain common Groovy script expressions that can be called from within any business component class. See How to Create a Script Expression Class File.

  - The ADF application now allows the user to turn on/off data security of a View Object by setting a flag `EOSecurityOverride`. This is a metadata level setting on the `viewdef.xml` of the View Object. See What You May Need to Know About Overriding Security for Entity Objects.

- ADF Task Flows
  - The ADF application now supports EL expressions set on the optional task flow binding property `processRegionForRefresh` to determine if the associated ADF region should participate in a refresh operation. By default, the ADF region participates in refresh operations. See How to Configure the Refresh of an ADF Region.

- ADF Web User Interface
  - The ADF application now supports creating an `af:inputSearch` LOV component to fetch list data from a REST URL on LOV-enabled attributes defined as type `inputSearch`. See How to Provide af:inputSearch LOV Component the REST URL.

Other Significant Changes in this Document for Release 12c (12.2.1.4.0)

For Release 12c (12.2.1.4.0), this document has been updated in several ways. Following are the sections that have been added or changed.

Part II Building Your Business Services

- Removed sections related to creating custom methods for ADF REST resources. This is no longer supported in release 12.2.1.4.0.
- Removed sections that described obsolete use cases for bind variables. The generation of bind variables on custom view object classes is no longer an option in the Select Java Options dialog that you open from the Java page of the view object overview editor in JDeveloper. The recommended way to allow UI fields to update bind parameter values is through view criteria.

Part III Using the Model Layer

- Revised the ADF REST batch requests section to remove the resource version in the payload sample that also shows the resource version in the POST URL to clarify that only the POST URL must provide a resource version. See Making Batch Requests.
- Revised the ADF REST headers section to update the list of available headers for REST requests and responses. See ADF REST HTTP Headers Support.
- Removed sections related to executing custom methods for ADF REST resources. This feature is no longer supported in release 12.2.1.4.0.

Part IV Creating ADF Task Flows

- Revised the ADF skins style properties section to remove the deprecated `web.xml` context parameter `oracle.adf.view.rich.remote.PRODUCER_STYLESHEET_DEBUG_LEVEL`. See What You May Need to Know About ADF Skins and Remote Regions.

- Revised the ADF Region refresh sections as follows:
  - Revised sections to clarify that **ifNeeded** is recommended over a **RefreshCondition** expression when choosing an option to determine region refresh behavior. See step 3 of How to Configure the Refresh of an ADF Region and see also What You May Need to Know About Configuring an ADF Region to Refresh.
  - Added a section to clarify the behavior of ADF regions in a page refresh. To avoid refreshing the regions that are not visible in the active view tree, set the `oracle.adfinternal.controller.regionPruning` parameter value to `true`. See What You May Need to Know About Refreshing an ADF Region.

Part VI Completing Your Application

- Revised the passivating transient view objects section to note that to enable a transient view object to passivate or activate correctly, requires a non-null key attribute value. See What You May Need to Know About Passivating Transient View Objects.

Part VII Appendices

- Revised the topic on `EntityImpl` methods to note that downcasting the return value of an `EntityImpl.getStructureDef()` method invocation to `EntityCache` is not supported. To obtain the instance of `EntityCache` invoke an existing `getEntityCache()`method on the `EntityImpl` object. See Methods You Typically Call on EntityImpl.

Part I

Getting Started with Fusion Web Applications

This part describes the architecture and key functionality of Oracle Application Development Framework (Oracle ADF) when used to build a Fusion web application that uses ADF Business Components, ADF Model, ADF Controller, and ADF Faces (the full Fusion technology stack).

Part I contains the following chapters:

1

Introduction to Building Fusion Web Applications with Oracle ADF

This chapter describes the architecture and key functionality of Oracle Application Development Framework (Oracle ADF) when used to build a Fusion web application that uses ADF Business Components, ADF Model, ADF Controller, and ADF Faces, in other words, the full Fusion technology stack. It also provides high-level development practices.

This chapter includes the following sections:

For definitions of unfamiliar terms found in this and other books, see the Glossary.

Introduction to Oracle ADF

Oracle ADF is a commercial Java framework for building enterprise applications. It supports rapid application development based on ready-to-use design patterns, metadata-driven and visual tools.

The **Oracle Application Development Framework** (Oracle ADF) is an end-to-end application framework that builds on Java Platform, Enterprise Edition (Java EE) standards and open-source technologies. You can use Oracle ADF to implement enterprise solutions that search, display, create, modify, and validate data using web, wireless, desktop, or web services interfaces. Because of its declarative nature, Oracle ADF simplifies and accelerates development by allowing users to focus on the logic of application creation rather than coding details. Used in tandem, Oracle JDeveloper and Oracle ADF give you an environment that covers the full development lifecycle from design to deployment, with drag and drop data binding, visual UI design, and team development features built in.

> **Tip:**
>
> You can also develop ADF applications using Oracle Enterprise Pack for Eclipse (OEPE). OEPE is a set of plug-ins designed for the Eclipse IDE to support Java EE development. OEPE also includes support for ADF application development, though that support is more limited than that provided by JDeveloper. See `http://www.oracle.com/technetwork/developer-tools/eclipse/overview/index.html`.

You can develop Fusion web applications, which are applications built with the full Fusion technology stack—ADF Business Components, ADF Model, ADF Controller, and JavaServer Faces pages with ADF Faces components. As a companion to this guide, you can download and view the Summit sample applications for Oracle ADF, which help to illustrate the concepts and procedures in this guide (and other Oracle Fusion Middleware developer guides). The examples in this sample application are built using the Fusion web application technology stack. Screenshots and code samples from this application are used throughout this guide to provide you with real-world examples of using the Oracle ADF technologies. For information about downloading and using the Summit ADF sample applications, see Introduction to the ADF Sample Application.

Oracle ADF Architecture

Oracle ADF architecture is based on Model-View-Controller (MVC) design pattern that consists of four layers – Model, View, Controller, and Business Services. Separating applications into these layers helps in the maintenance and reusability of components across applications.

In line with community best practices, applications you build using the Fusion web technology stack achieve a clean separation of business logic, page navigation, and user interface by adhering to a model-view-controller architecture. As shown in Figure 1-1, in an MVC architecture:

- The model layer represents the data values related to the current page, along with the model-level business rules, security, and application logic used against the data values.
- The view layer contains the UI pages used to view or modify that data.
- The controller layer processes user input and determines page navigation.
- The business service layer handles data access and encapsulates business logic.

Figure 1-1 MVC Architecture Cleanly Separates UI, Business Logic and Page Navigation

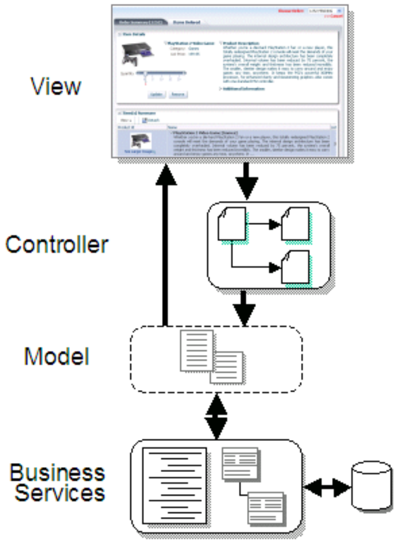
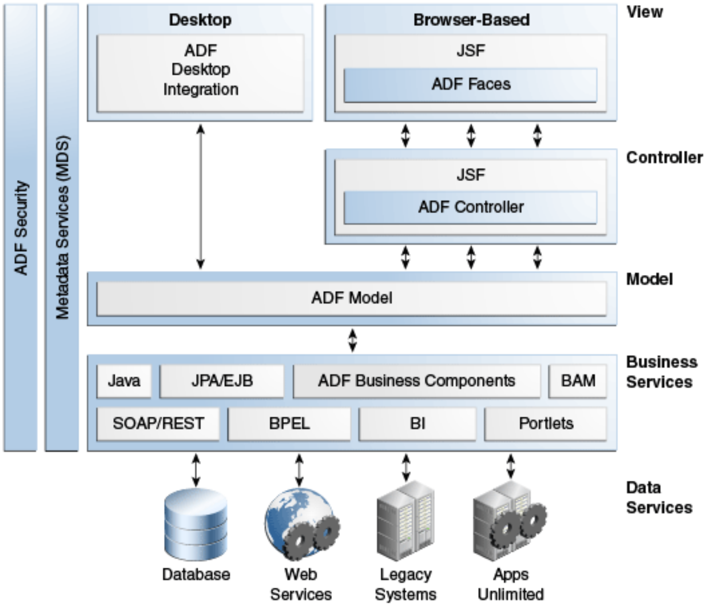


Figure 1-2 illustrates where each ADF module fits in the Fusion web application architecture. The core module in the framework is ADF Model, a data binding facility. ADF Model enables a unified approach to bind any user interface to any business service, without the need to write code. The other modules that make up a Fusion web application technology stack are:

- **ADF Business Components**, which simplify building business services.
- **ADF Faces**, which offers a rich library of ajax-enabled UI components for web applications built with JavaServer Faces (JSF).
- **ADF Controller**, which integrates JSF with ADF Model. ADF Controller extends the standard JSF controller by providing additional functionality, such as reusable task flows that pass control not only between JSF pages, but also between other activities, for instance method calls or other task flows. The ADF Controller also works with the data model to ensure the proper data is used for each request or **region**.

> **Note:**
>
> In addition to ADF Faces, Oracle ADF also supports using the Java and standard JSF view technologies. For information about these technologies, see Getting Started with Developing Java Applications in *Developing Applications with Oracle JDeveloper*. Oracle ADF also provides support for using Microsoft Excel as a view layer for your application. See About ADF Desktop Integration with Microsoft Excel in *Developing Applications with Oracle ADF Desktop Integration*.

Figure 1-2 Simple Oracle ADF Architecture



For information about the Oracle ADF architecture and the individual ADF technologies, see Overview of Oracle ADF in *Understanding Oracle Application Development Framework*.

Overview of Building an Application with Oracle ADF

You can build a Fusion web application with Oracle ADF using JDeveloper. Use the step-by-step ADF application development process that minimizes the coding effort of the developer to build an application’s infrastructure and to implement complex business logic of the application.

Oracle ADF emphasizes the use of the declarative programming paradigm throughout the development process to allow users to focus on the logic of application creation without having to get into implementation details. Using JDeveloper with Oracle ADF, you benefit from a high-productivity environment that automatically manages your application's declarative metadata for data access, validation, page control and navigation, user interface design, and data binding.

At a high level, the development process for a Fusion web application usually involves the following:

- Creating an application workspace: Using a wizard, JDeveloper automatically adds the libraries and configuration needed for the technologies you select, and structures your application into projects with packages and directories.
- Modeling the database objects: You can create an online database or offline replica of any database, and use JDeveloper editors and diagrammers to edit definitions and update schemas.
- Creating use cases: Using the UML modeler, you can create use cases for your application.
- Designing application control and navigation: You use diagrammers to visually determine the flow of application control and navigation. JDeveloper creates the underlying XML for you.
- Identifying shared resources: You use a resource library that allows you to view and use imported libraries by simply dragging and dropping them into your application.
- Creating business components to access data: From your database tables, you create entity objects using wizards or dialogs. From those entity objects, you create the view objects used by the pages in your application. You can implement validation rules and other types of business logic using editors.
- Implementing the user interface with JSF: The **Data Controls panel** in JDeveloper contains a representation of the view objects in your application. Creating a user interface is as simple as dragging an object onto a page and selecting the UI component you want to display the underlying data. For UI components that are not databound, you use the Components window to drag and drop components. JDeveloper creates the page code for you.
- Binding UI components to data using ADF Model: When you drag an object from the Data Controls panel, JDeveloper automatically creates the bindings between the page and the data model.
- Incorporating validation and error handling: Once your application is created, you use editors to add additional validation and to define error handling.

- Securing the application: You use editors to create roles and populate these with users. You then use a flat file editor to define security policies for these roles and assign them to specific resources in your application.
- Testing and debugging: JDeveloper includes an integrated application server that allows you to fully test your application without needing to package it up and deploy it. JDeveloper also includes the ADF Declarative Debugger, a tool that allows you to set breakpoints and examine the data.
- Deploying the application: You use wizards and editors to create and edit deployment descriptors, JAR files, and application server connections.

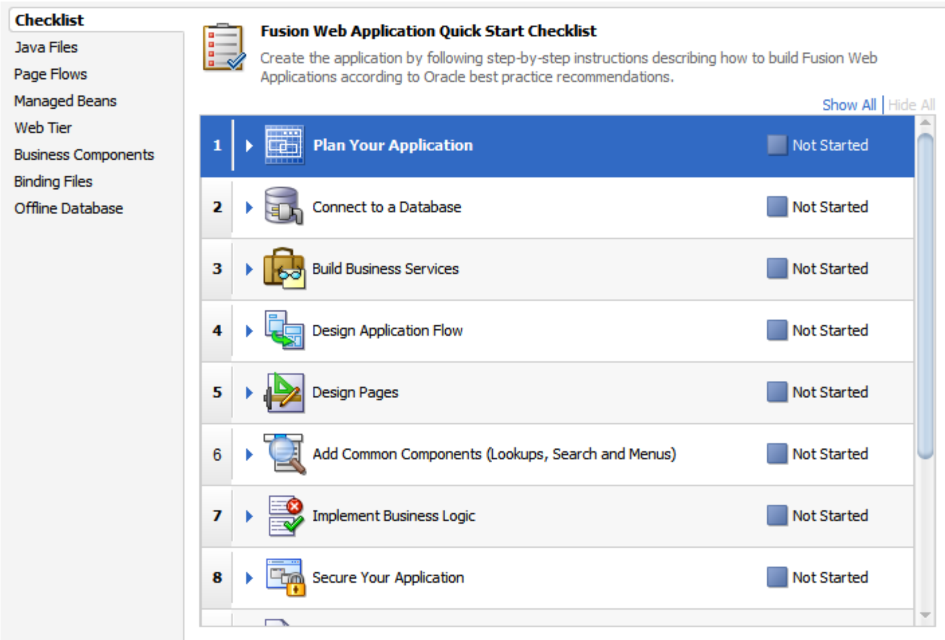
Creating an Application Workspace

The first step in building a new application is to assign it a name and to specify the directory where its source files will be saved. When you create an application using the application templates provided by JDeveloper, it organizes your workspace into projects and creates and organizes many of the configuration files required by the type of application you are creating.

One of these templates is the ADF Fusion Web Application template, which provides the correctly configured set of projects you need to create a web application that uses ADF Faces for the view, ADF Controller for the page flow, and ADF Business Components for business services. When you create an application workspace using this template, JDeveloper automatically creates the JSF and ADF configuration files needed for the application.

One part of the application overview is the Fusion Web Application Quick Start Checklist. This checklist provides you with the basic steps for creating a Fusion web application. Included are links to pertinent documentation, prerequisites, and the ability to keep track of status for each step in the checklist, as shown in Figure 1-3.

Figure 1-3 Fusion Web Application Quick Start Checklist



Fusion Web Application Quick Start Checklist

Create the application by following step-by-step instructions describing how to build Fusion Web Applications according to Oracle best practice recommendations.

1	Plan Your Application	Not Started
2	Connect to a Database	Not Started
3	Build Business Services	Not Started
4	Design Application Flow	Not Started
5	Design Pages	Not Started
6	Add Common Components (Lookups, Search and Menus)	Not Started
7	Implement Business Logic	Not Started
8	Secure Your Application	Not Started

JDeveloper also creates a project named `Model` that will contain all the source files related to the business services in your application, and a project named `ViewController` that will contain all the source files for your ADF Faces view layer and model layer, including files for the controller.

To save you having to add the associated libraries yourself, JDeveloper automatically adds the following libraries to the data model project, once you create a business component:

- ADF Model Runtime
- BC4J Oracle Domains
- BC4J Runtime
- BC4J Security
- MDS Runtime
- MDS Runtime Dependencies
- Oracle JDBC

JDeveloper also adds the following libraries to the view project:

- JSP Runtime
- JSF 2.1
- JSTL 1.2
- ADF Page Flow Runtime
- ADF Controller Runtime
- ADF Controller Schema
- ADF Faces Runtime 11
- ADF Common Runtime
- ADF Web Runtime
- MDS Runtime
- MDS Runtime Dependencies
- Commons Beanutils 1.6
- Commons Logging 1.0.4
- Commons Collections 3.1
- ADF DVT Faces Runtime
- ADF DVT Faces Databinding Runtime
- ADF DVT Faces Databinding MDS Runtime

Once you add a JSF page, JDeveloper adds the Oracle JEWT library.

You can then use JDeveloper to create additional projects, and add the packages and files needed for your application.

Note:

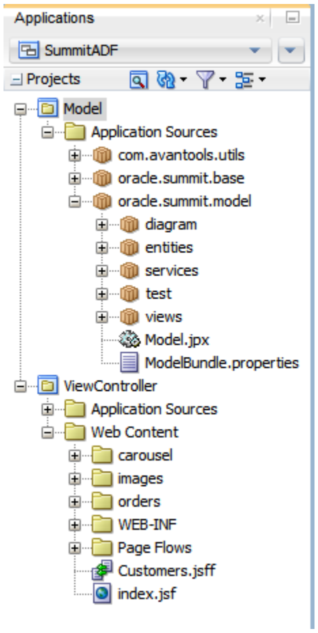
If you plan to reuse artifacts in your application (for example, task flows), then you should follow the naming guidelines presented in Reusing Application Components in order to prevent naming conflicts.

Tip:

You can edit the default values used in application templates, as well as create your own templates. To do so, choose **Application > Manage Templates**.

Figure 1-4 shows the different projects, packages, directories, and files for the Summit ADF sample application, as displayed in the Applications window.

Figure 1-4 Summit ADF Sample Application Projects, Packages, and Directories

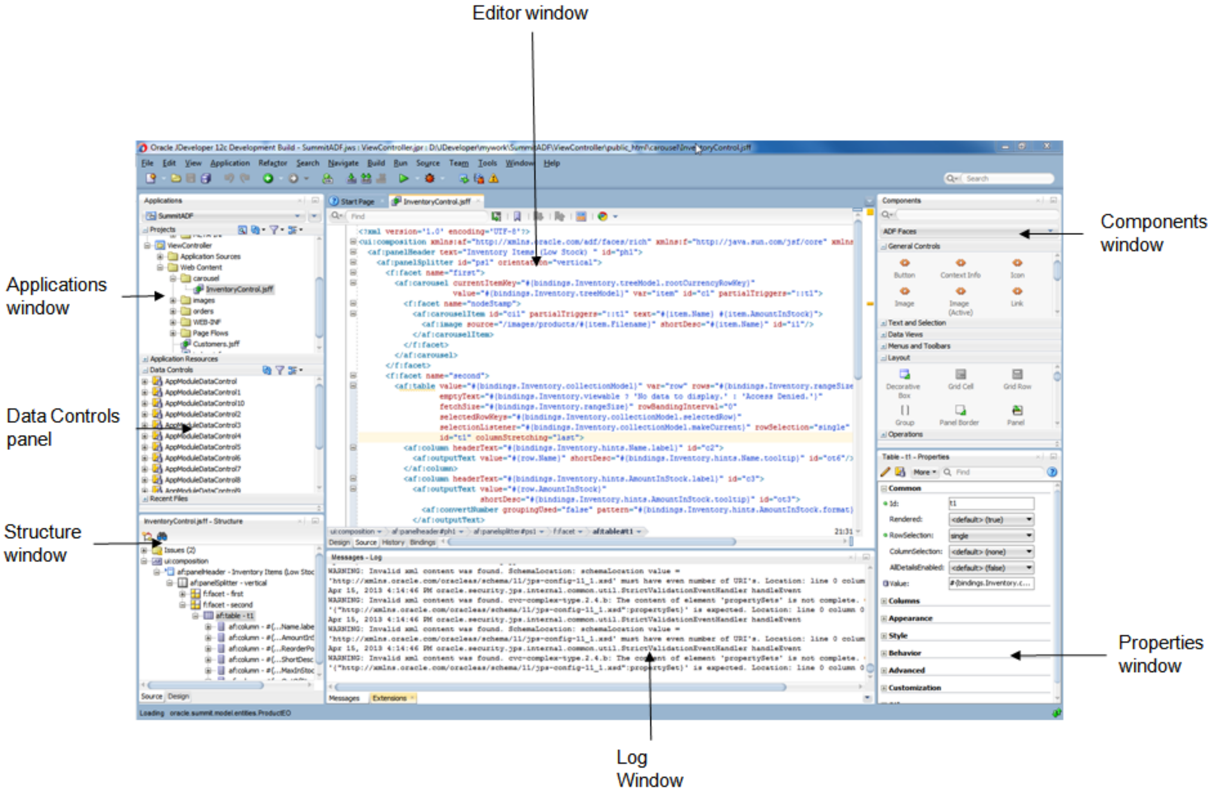


```
Applications
SummitADF
Projects
Model
  Application Sources
    com.avantools.utils
    oracle.summit.base
    oracle.summit.model
      diagram
      entities
      services
      test
      views
      Model.jpx
      ModelBundle.properties
ViewController
  Application Sources
  Web Content
    carousel
    images
    orders
    WEB-INF
    Page Flows
    Customers.jsff
    index.jsf
```

For more information, see the "Managing Applications and Projects" section of *Developing Applications with Oracle JDeveloper*.

When you work with your files, you use mostly the Applications window, editor window, the Structure window, and the Properties window, as shown in Figure 1-5. The editor window allows you to view many of your files in a WYSIWYG environment, or you can view a file in an overview editor where you can declaratively make changes, or you can view the source code for the file. The Structure window shows the structure of the currently selected file. You can select objects in this window and then edit the properties for the selection in the Properties window.

Figure 1-5 The JDeveloper Workspace



```
<?xml version='1.0' encoding='UTF-8'?>
<ui:composition xmlns:af="http://xmlns.oracle.com/adf/faces/rich" xmlns:f="http://java.sun.com/jsf/core" xmlns
  <af:panelHeader text="Inventory Items (Low Stock) " id="ph1">
    <af:panelSplitter id="ps1" orientation="vertical">
      <f:facet name="first">
        <af:carousel currentItemKey="#{bindings.Inventory.treeModel.rootCurrencyRowKey}"
                     value="#{bindings.Inventory.treeModel}" var="item" id="c1" partialTriggers="::t1">
          <f:facet name="nodeStamp">
            <af:carouselItem id="ci1" partialTriggers="::t1" text="#{item.Name} #{item.AmountInStock}">
              <af:image source="/images/products/#{item.Filename}" shortDesc="#{item.Name}" id="i1"/>
            </af:carouselItem>
          </f:facet>
        </af:carousel>
      </f:facet>
      <f:facet name="second">
        <af:table value="#{bindings.Inventory.collectionModel}" var="row" rows="#{bindings.Inventory.rangeSize
                  emptyText="#{bindings.Inventory.viewable ? 'No data to display.' : 'Access Denied.'}"
                  fetchSize="#{bindings.Inventory.rangeSize}" rowBandingInterval="0"
                  selectedRowKeys="#{bindings.Inventory.collectionModel.selectedRow}"
                  selectionListener="#{bindings.Inventory.collectionModel.makeCurrent}" rowSelection="single"
                  id="t1" columnStretching="last">
          <af:column headerText="#{bindings.Inventory.hints.Name.label}" id="c2">
            <af:outputText value="#{row.Name}" shortDesc="#{bindings.Inventory.hints.Name.tooltip}" id="ot6"/>
          </af:column>
          <af:column headerText="#{bindings.Inventory.hints.AmountInStock.label}" id="c3">
            <af:outputText value="#{row.AmountInStock}"
                           shortDesc="#{bindings.Inventory.hints.AmountInStock.tooltip}" id="ot3">
              <af:convertNumber groupingUsed="false" pattern="#{bindings.Inventory.hints.AmountInStock.format}
            </af:outputText>
```

```
WARNING: Invalid xml content was found. SchemaLocation: schemaLocation value =
'http://xmlns.oracle.com/oracleas/schema/11/jps-config-11_1.xsd' must have even number of URI's. Location: line 0 colum
Apr 15, 2013 4:14:46 PM oracle.security.jps.internal.common.util.StrictValidationEventHandler handleEvent
WARNING: Invalid xml content was found. cvc-complex-type.2.4.b: The content of element 'propertySets' is not complete.
'{"http://xmlns.oracle.com/oracleas/schema/11/jps-config-11_1.xsd":propertySet}' is expected. Location: line 0 column 0
Apr 15, 2013 4:14:46 PM oracle.security.jps.internal.common.util.StrictValidationEventHandler handleEvent
WARNING: Invalid xml content was found. SchemaLocation: schemaLocation value =
'http://xmlns.oracle.com/oracleas/schema/11/jps-config-11_1.xsd' must have even number of URI's. Location: line 0 colum
Apr 15, 2013 4:14:46 PM oracle.security.jps.internal.common.util.StrictValidationEventHandler handleEvent
WARNING: Invalid xml content was found. cvc-complex-type.2.4.b: The content of element 'propertySets' is not complete.
'{"http://xmlns.oracle.com/oracleas/schema/11/jps-config-11_1.xsd":propertySet}' is expected. Location: line 0 column 0
```

Modeling with Database Object Definitions

In JDeveloper, you can work with a database that is online, or, after you create your application workspace, you can copy database objects from a database schema to an offline database or project where they become available as offline database objects, saved as `.xml` files. With either an online or offline database, you can then create and edit database object definitions within a project. You can also compare your offline database objects with other offline or live database schemas and generate SQL statements (including `CREATE`, `REPLACE`, and `ALTER`).

For example, you can drag a table from a database connection that your application defines onto a database diagram and JDeveloper will give you the choice to model the database object live or offline (to create the `.xml` file representation of the object). Modeling database definitions, such as tables and foreign keys, visually captures the essential information about a schema. You can use the diagram to drag and drop columns and keys to duplicate, move, and create foreign key relationships. Working in offline mode, whenever you model a node on a diagram, JDeveloper creates the underlying offline object and lists it in the Applications window. Working with a live schema, JDeveloper updates the live database object as you amend the diagram. You can create multiple diagrams from the same offline database objects and spread your offline database across multiple projects.

Using a database diagram like the one shown in Figure 1-6 you can visualize the following:

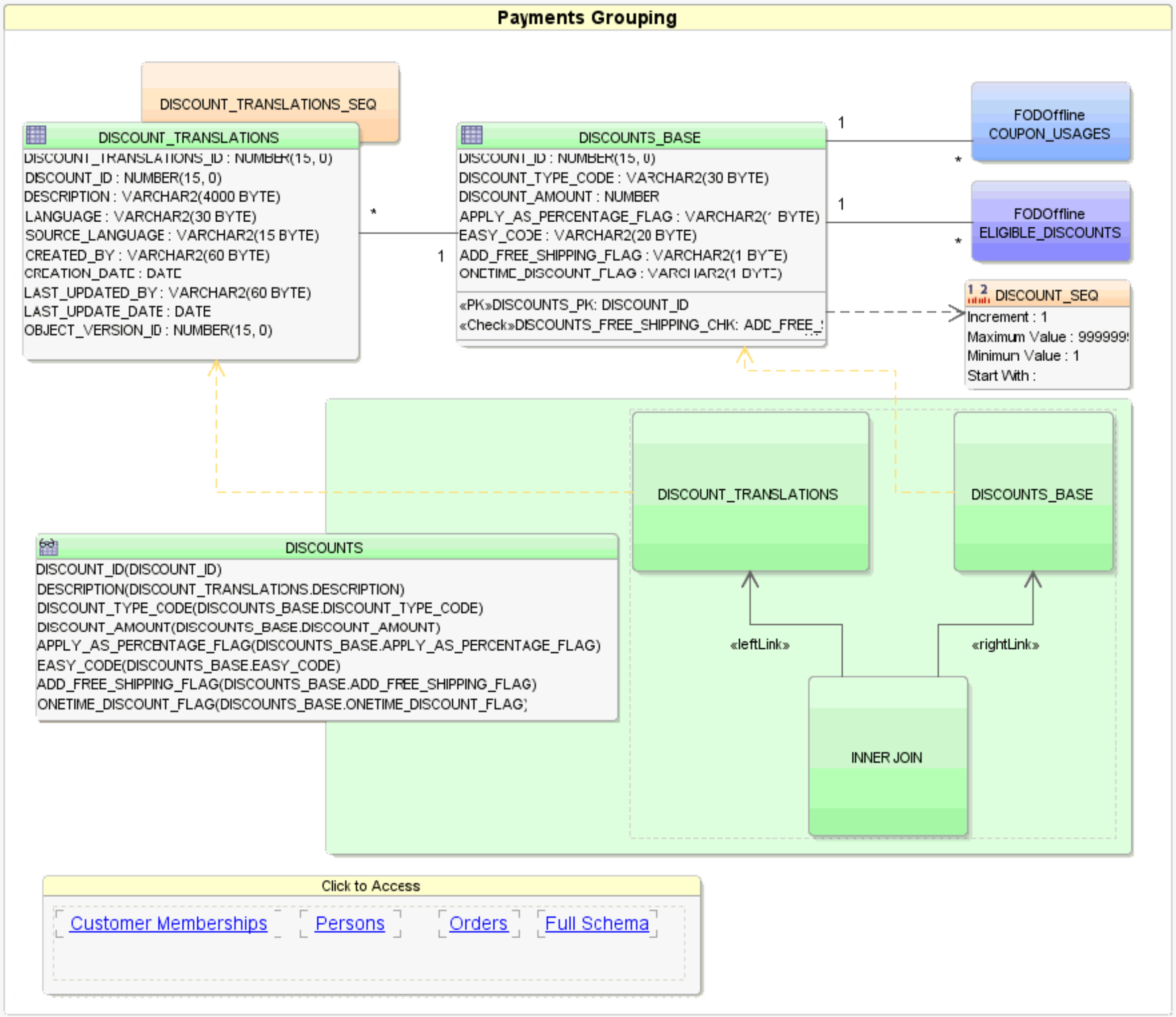
- Tables and their columns

- Foreign key relationships between tables
- Views
- Offline sequences and synonyms

In addition to using the diagram directly to create and edit database objects, you can work with specific database object editors. After you have finished working with the offline database objects, you can generate new and updated database definitions to online database schemas.

When you work with the database diagram you can customize the diagram to change the layout, change how objects are represented and grouped, add diagram annotations to specify dependencies or links (such as URLs), and change visual properties, such as color and font of diagram elements.

Figure 1-6   Database Diagram for Payments Grouping

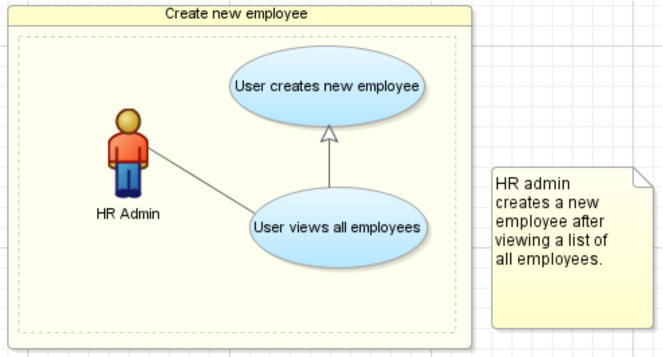

Payments Grouping
DISCOUNT_TRANSLATIONS_SEQ
DISCOUNT_TRANSLATIONS
DISCOUNT_TRANSLATIONS_ID : NUMBER(15, 0)
DISCOUNT_ID : NUMBER(15, 0)
DESCRIPTION : VARCHAR2(4000 BYTE)
LANGUAGE : VARCHAR2(30 BYTE)
SOURCE_LANGUAGE : VARCHAR2(15 BYTE)
CREATED_BY : VARCHAR2(60 BYTE)
CREATION_DATE : DATE
LAST_UPDATED_BY : VARCHAR2(60 BYTE)
LAST_UPDATE_DATE : DATE
OBJECT_VERSION_ID : NUMBER(15, 0)
*
1
DISCOUNTS_BASE
DISCOUNT_ID : NUMBER(15, 0)
DISCOUNT_TYPE_CODE : VARCHAR2(30 BYTE)
DISCOUNT_AMOUNT : NUMBER
APPLY_AS_PERCENTAGE_FLAG : VARCHAR2(1 BYTE)
EASY_CODE : VARCHAR2(20 BYTE)
ADD_FREE_SHIPPING_FLAG : VARCHAR2(1 BYTE)
ONETIME_DISCOUNT_FLAG : VARCHAR2(1 BYTE)
«PK»DISCOUNTS_PK: DISCOUNT_ID
«Check»DISCOUNTS_FREE_SHIPPING_CHK: ADD_FREE_S
1
*
FODOffline
COUPON_USAGES
1
*
FODOffline
ELIGIBLE_DISCOUNTS
DISCOUNT_SEQ
Increment : 1
Maximum Value : 999999
Minimum Value : 1
Start With :
DISCOUNT_TRANSLATIONS
DISCOUNTS_BASE
«leftLink»
«rightLink»
INNER JOIN
DISCOUNTS
DISCOUNT_ID(DISCOUNT_ID)
DESCRIPTION(DISCOUNT_TRANSLATIONS.DESCRIPTION)
DISCOUNT_TYPE_CODE(DISCOUNTS_BASE.DISCOUNT_TYPE_CODE)
DISCOUNT_AMOUNT(DISCOUNTS_BASE.DISCOUNT_AMOUNT)
APPLY_AS_PERCENTAGE_FLAG(DISCOUNTS_BASE.APPLY_AS_PERCENTAGE_FLAG)
EASY_CODE(DISCOUNTS_BASE.EASY_CODE)
ADD_FREE_SHIPPING_FLAG(DISCOUNTS_BASE.ADD_FREE_SHIPPING_FLAG)
ONETIME_DISCOUNT_FLAG(DISCOUNTS_BASE.ONETIME_DISCOUNT_FLAG)
Click to Access
Customer Memberships
Persons
Orders
Full Schema

For more information about modeling database definitions with database diagrams, see the "Creating, Editing, and Dropping Database Objects" section in *Developing Applications with Oracle JDeveloper*.

Creating Use Cases

After creating an application workspace, you may decide to begin the development process by doing use case modeling to capture and communicate end-user requirements for the application to be built. Figure 1-7 shows a simple diagram for an HR administrator creating a new employee that you might design using the UML modeler in JDeveloper. Using diagram annotations, you can capture particular requirements about what end users might need to see on the screens that will implement the use case. For example, in this use case, it is noted that the user will choose to create a new employee while viewing a list of all employees.

Figure 1-7 Use Case Diagram for Viewing Order History


Create new employee
User creates new employee
HR Admin
User views all employees
HR admin creates a new employee after viewing a list of all employees.

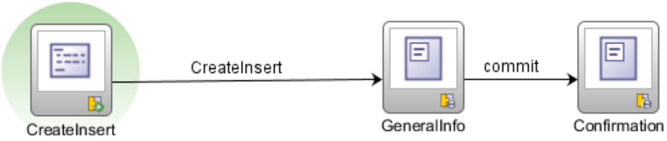
For more information about creating use case diagrams, see the "Developing Applications Using Modeling" chapter of *Developing Applications with Oracle JDeveloper*.

Designing Application Control and Navigation Using ADF Task Flows

By modeling the use cases, you begin to understand the kinds of user interface pages that will be required to implement end-user requirements. At this point, you can use these diagrams as documentation tools that will help as you begin to design the flow of your application. In a Fusion web application, you use **ADF task flows** instead of standard JSF navigation flows. Task flows provide a more modular and transaction-aware approach to navigation and application control. Like standard JSF navigation flows, task flows contain mostly viewable pages (or page fragments). Aside from navigation, task flows can also have nonvisual activities that can be chained together to affect the page flow and application behavior. For example, these nonvisual activities can call methods on managed beans, evaluate an EL expression, or call another task flow. This facilitates reuse, as business logic can be invoked independently of the page being displayed.

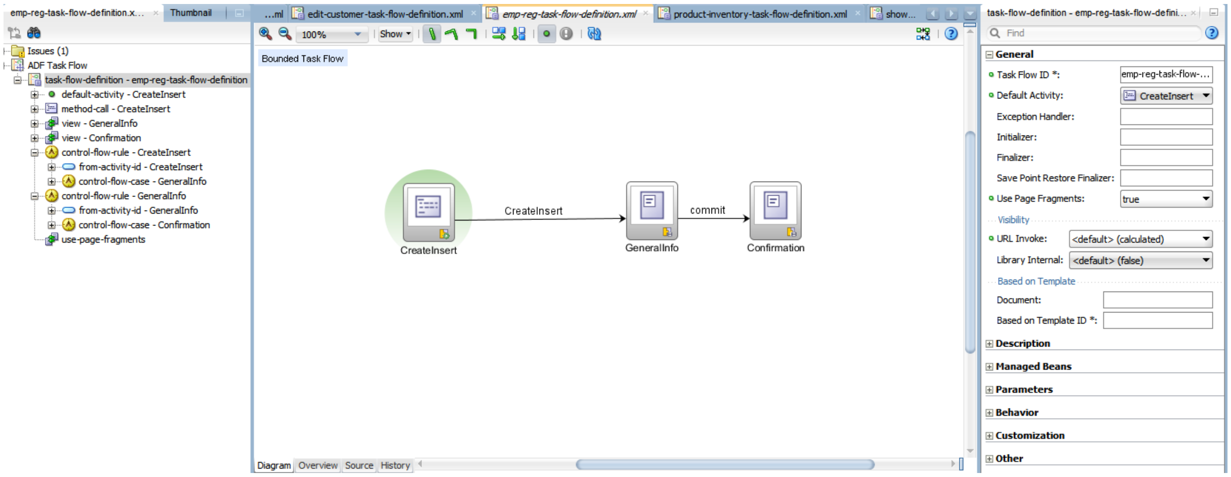
Figure 1-8 shows a task flow for creating an employee. In this task flow, `GeneralInfo` and `Confirmation` are view activities that represent pages, while `CreateInsert` is a method call activity. When the user enters this flow, the `CreateInsert` activity is invoked (because it is the entry point for the flow, as denoted by the green circle) and the corresponding method is called. Because this is a call to a method, and the method itself is not included in the task flow, the method can be reused elsewhere in the application. Or, the logic in the method could be changed, without affecting the metadata for the page flow. From there, the flow continues to the `GeneralInfo` page, and once the data is submitted, to the `Confirmation` page.

Figure 1-8 Task Flow to Create an Employee



ADF Controller provides a mechanism to define navigation using **control flow** rules. The control flow rule information, along with other information regarding the flow, is saved in a configuration file. Figure 1-9 shows the Structure window for the task flow. This window shows each of the items configured in the flow, such as the control flow rules. The Properties window (by default, located at the bottom right) allows you to set values for the different elements in the flow.

Figure 1-9 Task Flow Elements in the Structure Window and Properties Window


emp-reg-task-flow-definition.x...
Thumbnail
...ml
edit-customer-task-flow-definition.xml
emp-reg-task-flow-definition.xml
product-inventory-task-flow-definition.xml
show...
100%
Show
Bounded Task Flow
CreateInsert
CreateInsert
GeneralInfo
commit
Confirmation
Diagram
Overview
Source
History
task-flow-definition - emp-reg-task-flow-defini...
Find

- Issues (1)
- ADF Task Flow
  - task-flow-definition - emp-reg-task-flow-definition
    - default-activity - CreateInsert
    - method-call - CreateInsert
    - view - GeneralInfo
    - view - Confirmation
    - control-flow-rule - CreateInsert
      - from-activity-id - CreateInsert
      - control-flow-case - GeneralInfo
    - control-flow-rule - GeneralInfo
      - from-activity-id - GeneralInfo
      - control-flow-case - Confirmation
    - use-page-fragments

General

Task Flow ID *:	emp-reg-task-flow-...
Default Activity:	CreateInsert
Exception Handler:	
Initializer:	
Finalizer:	
Save Point Restore Finalizer:	
Use Page Fragments:	true
Visibility	
URL Invoke:	<default> (calculated)
Library Internal:	<default> (false)
Based on Template	
Document:	
Based on Template ID *:	

Description
Managed Beans
Parameters
Behavior
Customization
Other

Aside from pages, task flows can also coordinate page fragments. Page fragments are JSF documents that are rendered as content in other JSF pages. You can create page fragments and the control between them in a **bounded task flow** as you would create pages, and then insert the entire task flow into another page as a **region**. Because it is simply another task flow, the region can independently execute methods, evaluate expressions, and display content, while the remaining content on the containing page stays the same.

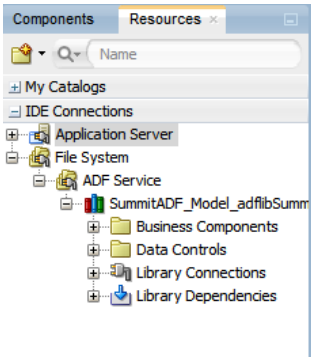
Regions also facilitate reuse. You can create a task flow as a region, determine the pieces of information required by a task and the pieces of information it might return, define those as parameters and return values of the region, then drop the region on any page in an application. Depending on the value of the parameter, a different view can display.

The chapters in Creating ADF Task Flows contain information about using task flows. For general information about task flows and creating them, see Getting Started with ADF Task Flows . For information about task flow activities, see Working with Task Flow Activities . If you need to pass parameters into or out of task flows, see Using Parameters in Task Flows. For more information about regions, see Using Task Flows as Regions . For information about advanced functionality that task flows can provide, such as transactional capabilities and creating mandatory sequences of pages (known as **trains**), see Creating Complex Task Flows . For information about using task flows to create dialogs, see Using Dialogs in Your Application.

Identifying Shared Resources

When designing the application, you may find that some aspects of your application can be reused throughout the application. For example, you might have one developer that creates the business components, and another that creates the web interface. The business component developer can then save the project and package it as a library. The library can be sent to other developers who can add it to their resource catalog, from which they can drag and drop it onto any page where it's needed. Figure 1-10 shows a resources catalog in the Resources window of JDeveloper.

Figure 1-10 Resources Window in JDeveloper



Be sure to note all the tasks that can possibly become candidates for reuse. Reusing Application Components provides more information about the ADF artifacts that can be packaged and reused as an ADF library, along with procedures both for creating and using the libraries.

Creating a Data Model to Access Data with ADF Business Components

Typically, when you implement business logic as **ADF Business Components**, you do the following:

- Create **entity objects** to represent tables that you expect your application to perform a transaction against. The entity objects can actually be created from an existing database schema. Add validation and business rules as needed.
- Create **view object**s that work with the entity objects to query the database. These view objects will be used to make the data available for display at your view layer. You can also create read-only view objects, which you might use to

display static lists. Like entity objects, view objects can be created from an existing database schema.

- Create the **application module**, which is the transactional component that UI clients use to work with application data. The application module provides the interface to the business services that a consumer of those services (such as the UI layer of your application) will use. This application module contains view object instances in its data model along with any custom methods that users will interact with through the application's web pages.
- If needed, publish your services as web services for remote invocation.

The chapters contained in Building Your Business Services provide information on creating each of these artifacts. The chapters in Completing Your Application provide additional information, such as extending business objects, tuning, and state management.

Creating a Layer of Business Domain Objects for Tables

Once you have an understanding of the data that will be presented and manipulated in your application, if you haven't already done so, you can build your database (for more information, see the "Designing Databases Within Oracle JDeveloper" chapter of *Developing Applications with Oracle JDeveloper*). Once the database tables are in place, you can create a set of entity objects that represents them and simplifies modifying the data they contain. When you use entity objects to encapsulate data access and validation related to the tables, any pages you build today or in the future that work with these tables are consistently validated. Any relationships between the tables will be reflected as associations between the corresponding entity objects.

Once the entity objects are created, you can define control and attribute hints that simplify the display of the entities in the UI, and you can also add behaviors to the objects. For more information, see Creating a Business Domain Layer Using Entity Objects.

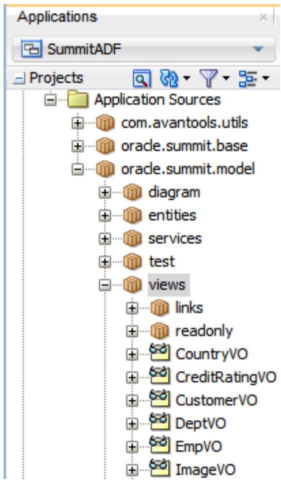
Building the Business Services

Once the reusable layer of business objects is created, you can implement the application module's data model and service methods with which a UI client can work.

The application module's data model is composed of instances of the view object components you create that encapsulate the necessary queries. View objects can join, project, filter, sort, and aggregate data into the shape required by the end-user task being represented in the user interface. When the end user needs to update the data, your view objects reference entity objects in your reusable business domain layer. View objects are reusable and can be used in multiple application modules. For more information, see Defining SQL Queries Using View Objects.

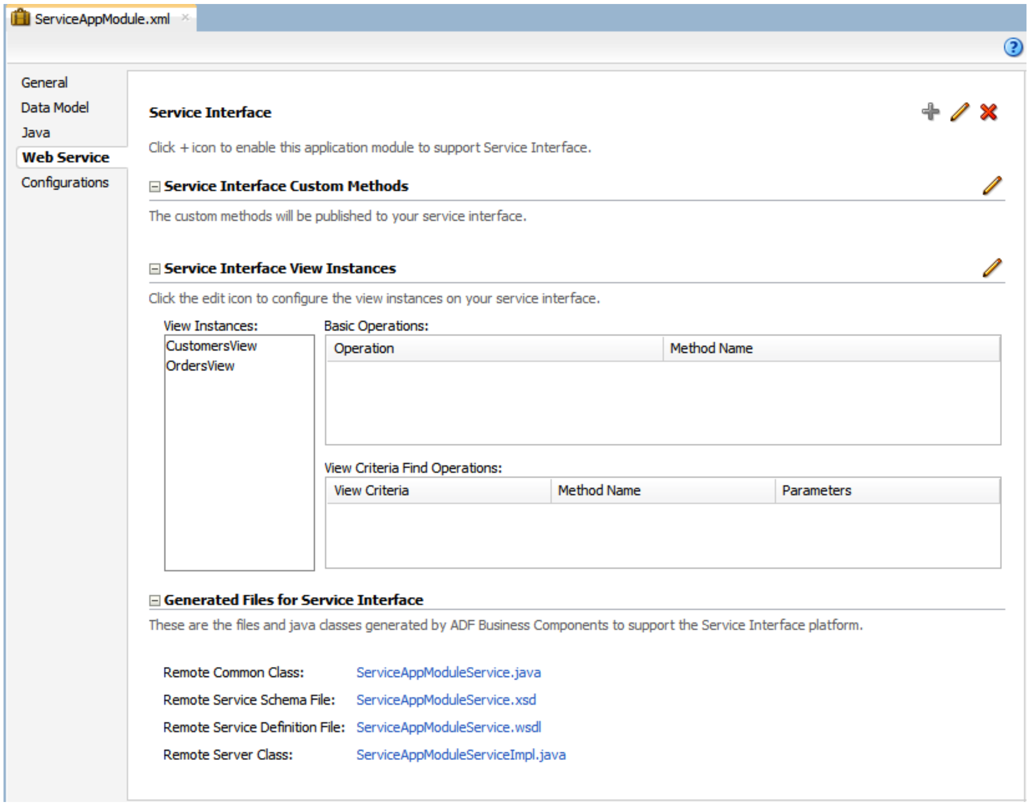
Figure 1-11 shows the `oracle.summit.model.view` package, which contains many of the queries needed by the application.

Figure 1-11   View Objects in the Summit ADF Sample Application



Additionally, you may find that you need to expose functionality to external applications. You can do this by exposing this functionality through a service interface. For example, the `ServiceAppModule` application module is exposed as a web service. This web service exposes the `CustomersView` and OrdersView view instances, as shown in Figure 1-12. For more information, see Creating SOAP Web Services with Application Modules.

Figure 1-12   The ServiceAppModule Application Module as a Web Service



Service Interface

Click + icon to enable this application module to support Service Interface.

Service Interface Custom Methods

The custom methods will be published to your service interface.

Service Interface View Instances

Click the edit icon to configure the view instances on your service interface.

View Instances:
CustomersView
OrdersView

Basic Operations:

Operation	Method Name

View Criteria Find Operations:

View Criteria	Method Name	Parameters

Generated Files for Service Interface

These are the files and java classes generated by ADF Business Components to support the Service Interface platform.

Remote Common Class: ServiceAppModuleService.java
Remote Service Schema File: ServiceAppModuleService.xsd
Remote Service Definition File: ServiceAppModuleService.wsdl
Remote Server Class: ServiceAppModuleServiceImpl.java

Testing and Debugging Business Services with the Oracle ADF Model Tester

While you develop your application, you can iteratively test your business services using the Oracle ADF Model Tester. The tester allows you to test the queries, business logic, and validation of your business services without having to use or create a user interface or other client to test your services. Using the tester allows you to test out the latest queries or business rules you've added, and can save you time when you're trying to diagnose problems. For more information about developing and testing application modules, see Implementing Business Services with Application Modules.

The tester also interacts with the ADF Declarative Debugger to allow debug your business services. You can set breakpoints on any custom methods you create. For more information, see Using the Oracle ADF Model Tester for Testing and Debugging.

Implementing the User Interface with JSF

From the page flows you created during the planning stages, you can double-click the page icons to create the actual JSF files. When you create a JSF page for an ADF Faces application, by default, you create a Facelets XHTML file that uses the `.jsf` extension. Facelets is a JSF-centric XML view definition technology that provides an alternative to using the JSP engine.

> **Tip:**
>
> While Facelet pages can use any well formed XML file, when you create a Facelet page in JDeveloper, it is created as an XHTML file.

ADF Faces provides a number of components that you can use to define the overall layout of the page. JDeveloper contains predefined quick start layouts that use these components to provide you with an efficient way to correctly determine the layout of your pages. You can choose from one-, two-, or three-column layouts, and then determine how you want the columns to behave. You can also choose to apply themes to the layouts, which adds color to some of the components for you. For more information see the "Using Quick Start Layouts" section of *Developing Web User Interfaces with Oracle ADF Faces*.

Oracle ADF also allows you to create and use your own page templates. When creating templates, you can determine the layout of the page (either using one of the quick layout templates or creating the layout manually), provide static content that must appear on all pages, and create placeholder attributes that can be replaced with valid values for each page. Each time the template is changed, for example if the layout changes, any page that uses the template will reflect the update.

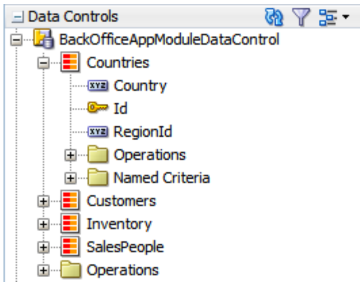
The chapters in Creating a Databound Web User Interface provide information on creating different types of UI functionality, from basic forms to more complex search capabilities.

Data Binding with ADF Model

In a typical JSF application, you bind the UI component attributes to properties or methods on managed beans. The JSF runtime manages instantiating these beans on demand when any EL expression references them for the first time. However, in an application that uses ADF Model, JDeveloper automatically binds the UI component attributes to ADF Model, which uses XML configuration files that drive generic data binding features. It implements concepts that enable decoupling the user interface technology from the business service implementation: **data controls** and **declarative bindings**.

Data controls use XML configuration files to describe a service. At design time, visual tools like JDeveloper can leverage that metadata to allow you to declaratively bind UI components to any data control operation or data collection, creating bindings. For example, Figure 1-13 shows the `BackOfficeAppModuleDataControl` data control as it appears in the Data Controls panel of JDeveloper.

Figure 1-13 BackOfficeAppModuleDataControl

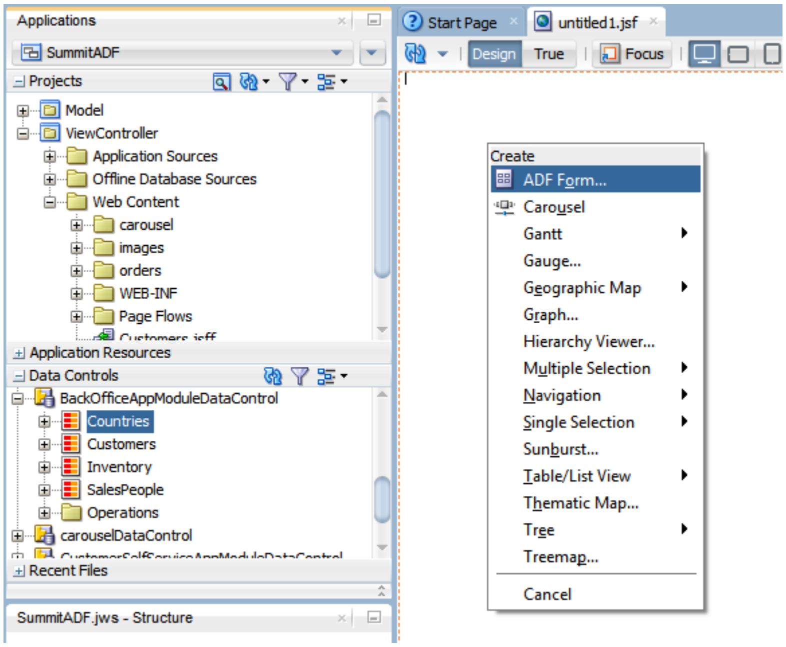


Note that the collections that display in the panel represent the set of rows returned by the query in each view object instance contained in the `BackOfficeAppModuleDataControl` application module. For example, the `Countries` data collection in the Data Controls panel represents the `Countries` view object instance in the `BackOfficeAppModuleDataControl's` data model. The attributes available in each row of the respective data collections appear as child nodes. The data collection level Operations node contains the built-in operations that ADF Model supports on data collections, such as `previous`, `next`, `first`, `last`, and so on.

> **Note:**
>
> If you create other kinds of data controls for working with web services, XML data retrieved from a URL, JavaBeans, or EJBs, these would also appear in the Data Controls panel with an appropriate display. When you create one of these data controls in a project, JDeveloper creates metadata files that contain configuration information. These additional files do not need to be explicitly created when you are working with Oracle ADF application modules, because application modules are already metadata-driven components, and so contain all the information necessary to be exposed automatically as ADF Model data controls.

Using the Data Controls panel, you can drag and drop a data collection onto a page in the visual editor, and JDeveloper creates the necessary bindings for you. Figure 1-14 shows the `Countries` collection being dragged from the Data Controls panel, and dropped as a form onto a JSF page.

Figure 1-14   Declaratively Creating a Form Using the Data Controls Panel



The first time you drop a databound component from the Data Controls panel on a page, JDeveloper creates an associated **page definition file**. This XML file describes the group of bindings supporting the UI components on a page. ADF Model uses this file at runtime to instantiate the page's bindings. These bindings are held in a request-scoped map called the **binding container**. Each time you add components to the page using the Data Controls panel, JDeveloper adds appropriate binding entries into this page definition file. Additionally, as you perform drag and drop data binding operations, JDeveloper creates the required tags representing the JSF UI components on the JSF page. For more information about using the Data Controls panel, see Using ADF Model in a Fusion Web Application.

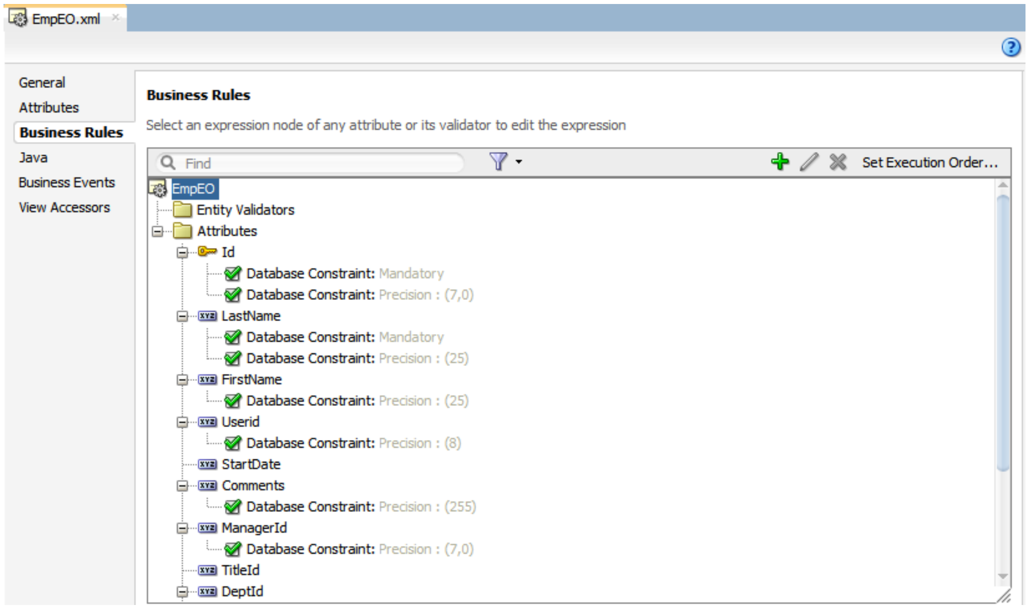
Aside from forms and tables that display or update data, you can also create search forms, and databound charts and graphs. For more information about using data controls to create different types of pages, see the chapters contained in Creating a Databound Web User Interface . For more information about the Data Controls panel and how to use it to create any UI data bound component, see Using ADF Model in a Fusion Web Application. For detailed information about ADF Model, see About ADF Model in *Developing Applications with Oracle ADF Data Controls*.

Validation and Error Handling

You can add validation to your business objects declaratively using the overview editors for entity and view objects. In the case of entities, the business rules always reflect the data type constraints defined on the attributes.

Figure 1-15 shows the Business Rules page of the overview editor for the EmpEO entity object.

Figure 1-15   Setting Validation in the Overview Editor



Business Rules

Select an expression node of any attribute or its validator to edit the expression

```
EmpEO
  Entity Validators
  Attributes
    Id
      Database Constraint: Mandatory
      Database Constraint: Precision : (7,0)
    LastName
      Database Constraint: Mandatory
      Database Constraint: Precision : (25)
    FirstName
      Database Constraint: Precision : (25)
    Userid
      Database Constraint: Precision : (8)
    StartDate
    Comments
      Database Constraint: Precision : (255)
    ManagerId
      Database Constraint: Precision : (7,0)
    TitleId
    DeptId
```

Along with providing the validation rules, you also set the error messages to display when validation fails. To supplement this declarative validation, you can also use Groovy-scripted expressions. For more information about creating validation at the service level, see Defining Validation and Business Rules Declaratively.

Additionally, ADF Faces input components have built-in validation capabilities. You set one or more validators on a component either by setting the `required` attribute or by using the prebuilt ADF Faces validators. You can also create your own custom validators to suit your business needs. For more information, see the "Validating and Converting Input" chapter of *Developing Web User Interfaces with Oracle ADF Faces*.

You can create a custom error handler to report errors that occur during execution of an application. Once you create the error handler, you only need to register the handler in one of the application's configuration files.

Adding Security

Oracle ADF provides a security implementation that is based on Java Authentication and Authorization Service (JAAS). It enables applications to authenticate users and enforce authorization. The Oracle ADF implementation of JAAS is permission-based. You define these permissions and then grant them on application roles that you associate with users of the application. For more information about securing your application, see Enabling ADF Security in a Fusion Web Application.

Testing and Debugging the Web Client Application

Testing an Oracle ADF web application is similar to testing and debugging any other Java EE application. Most errors result from simple and easy-to-fix problems in the declarative information that the application defines or in the EL expressions that access the runtime objects of the page's ADF binding container. In many cases, examination of the declarative files and EL expressions resolve most problems.

For errors not caused by the declarative files or EL expressions, you can use the ADF Logger, which captures runtime trace messages from ADF Model API. The trace includes runtime messages that may help you to quickly identify the origin of an application error. You can also search the log output for specific errors.

JDeveloper also includes the ADF Declarative Debugger, a tool that allows you to set breakpoints on declarative aspects of Oracle ADF, such as the binding layer, taskflows and more. When a breakpoint is reached, the execution of the application is paused and you can examine the data that the ADF binding container has to work with, and compare it to what you expect the data to be. Testing and Debugging ADF Components contains useful information and tips on how to successfully debug a Fusion web application.

For testing purposes, JDeveloper provides integration with JUnit. You use a wizard to generate regression test cases. For more information, see Regression Testing with JUnit.

Refactoring Application Artifacts

Using JDeveloper, you can easily rename or move the different components in your application. When you use JDeveloper to refactor the business components of your data model project, you can rename components or move them to a different package and JDeveloper will find and update all references to the refactored component. Whenever possible, you should use JDeveloper's refactoring actions to avoid error-prone manual editing of the project. For more information, see Refactoring a Fusion Web Application .

Deploying a Fusion Web Application

You can deploy a Fusion web application to either the integrated WebLogic server within JDeveloper or to a supported standalone instance. For more information about deployment, see Deploying Fusion Web Applications.

Integrating a Fusion Web Application

You can build your Fusion web application so that it can easily integrate with other applications. You can publish your application modules as services. You can also create events that can be used for example, to initiate business processes. See Creating SOAP Web Services with Application Modules. Your application modules can also call other web services directly. See Calling a Web Service from an Application Module. You can also integrate your application using task flows. For example, a task flow can be used to initiate a business process flow.

Working Productively in Teams

Working effectively in a team is crucial for application development. It requires effective communication, collaboration, and time management between the developers working on developing different parts of an ADF application.

Often, applications are built in a team development environment. While a team-based development process follows the development cycle outlined in Overview of Building an Application with Oracle ADF ,many times developers are creating the different parts of the application simultaneously. Working productively means team members divide the work, understand how to enforce standards, and manage source files with a source control system, in order to ensure efficient application development.

Before beginning development on any large application, a design phase is typically required to assess use cases, plan task flows and screens, and identify resources that can be shared.

The following list shows how the work for a typical Fusion web application might be broken up once an initial design is in place:

- Infrastructure

  An administrator creates Ant scripts (or other script files) for building and deploying the finished application. SQL scripts are developed to create the database schema used by the application.

- Entity objects

  In a large development environment, it may be that a separate development group builds all entity objects for the application. Because the rest of the application depends on these objects, entity objects should be one of the first steps completed in development of the application or part of the application.

  Once the entity objects are finished, they can be shared with other teams using Oracle ADF libraries (see Packaging a Reusable ADF Component into an ADF Library for more information). The other teams then access the objects by adding to them to a catalog in the Resources window. In your own application development process, you may choose not to divide the work this way. In many applications, entity objects and view objects might be developed by the same team (or even one person) and would be stored within one project.

- View objects

  After the entity objects are created and provided either in a library or within the project itself, view objects can be created as needed to display data (in the case of building the UI) or supply service data objects (when data is needed by other applications in a SOA infrastructure).

  During development, you may find that two or more view objects are providing the same functionality. In some cases, these view objects can be easily combined by altering the query in one of the objects so that it meets the needs of each developer's page or service. **View criteria**, in particular, enable developers to easily adapt a query to distinct usages.

  Once the view objects are in place, you can create the application module, data controls, and add any needed custom methods. The process of creating view objects, reviewing for redundancy, and then adding them to the application module can be an iterative one.

- User interface (UI) creation

  With a UI design in place, the view objects in place and the data controls created, the UI can be built either by the team that created the view objects (as described in the previous bullet point) or by a separate team. You can also develop using a UI-first strategy, which would allow UI designers to create pages before the data controls are in place. Oracle ADF provides placeholder data controls that UI designers can use early in the development cycle. See Designing a Page Using Placeholder Data Controls .

Enforcing Standards

Because numerous individuals divided into separate teams will be developing the application, you should enforce a number of standards before development begins to ensure that all components of the application will work together efficiently. The following are areas within an application where it is important to have standardization in place when working in a team environment:

- Code layout style

  So that more than one person can work efficiently in the code, it helps to follow specific code styles. JDeveloper allows you to choose how the built-in code editor behaves. While many of the settings affect how the user interacts with the code editor (such as display settings), others affect how the code is formatted. For example, you can select a code style that determines things like the placement of opening brackets and the size of indents. You can also import any existing code styles you may have, or you can create your own and export them for use by the team. For more information, see the "How to Set Preferences for the Source Editor" section in *Developing Applications with Oracle JDeveloper*.

- Package naming conventions

  You should determine not only how packages should be named, but also the granularity of how many and what kinds of objects will go into each package.

- Pages

  You can create templates to be used by all developers working on the UI, as described in Implementing the User Interface with JSF. This not only ensures that all pages will have the same look and feel, but also allows you to make a change in the template and have the change appear on all pages that use it. For more information, see Using Page Templates.

  Aside from using templates, you should also devise a naming standard for pages. For example, you may want to have names reflect where the page is used in the application. To achieve this goal, you can create subdirectories to provide a further layer of organization.

- Connection names: Unlike most JDeveloper and Oracle ADF objects that are created only once per project and by definition have the same name regardless of who sees or uses them, database connection names might be created by individual team members, even though they map to the same connection details. Naming discrepancies may cause unnecessary conflicts. Team members should agree in advance on common, case-sensitive connection names that should be used by every member of the team.

Using a Source Control System

When working in a team environment, you will need to use a source control system. By default, JDeveloper provides support for the Subversion source control system, and others (such as CVS) are available through extensions. You can also create an extension that allows you to work with another system in JDeveloper. For information about using these systems within JDeveloper, see the "Versioning Applications with Source Control" chapter of *Developing Applications with Oracle JDeveloper*.

Following are suggestions for using source control with a Fusion web application:

- Checking out files

Using JDeveloper, you can create a connection to the source control server and use the source control window to check out the source. When you work locally in the files, the pending changes window notifies you of any changed files. You can create a script using Apache Ant (which is integrated into JDeveloper). You can then use the script to build all application workspaces locally. This can ensure that the source files compile before you check the changed files into the source control repository. To find out how to use Apache Ant to create scripts, see the "Building with Apache Ant" section of *Developing Applications with Oracle JDeveloper*.

- Automating builds

  Consider running a continuous integration tool. Once files are checked into the source server, the tool can be used to recognize either that files have changed or to check for changed files at determined intervals. At that point, the tool can run an Ant script on the server that copies the full source (note that this should be a copy, and not a checkout), compiles those files, and if the compilation is successful, creates a zip file for consumers (not developers) of the application to use. The script should then clean up the source directory. Running a continuous integration tool will improve confidence in the quality of the code in the repository, encourage developers to update more often, and lead to smaller updates and fewer conflicts. Hudson (`http://hudson-ci.org/`) is one example of a continuous integration tool.

- Updating and committing files

  When working with Subversion, updates and commits should be done at the Working Copy level, not on individual files. If you attempt to commit and update an individual file, there is a chance you will miss a supporting metadata file and thereby corrupt the committed copy of the application.

- Resolving merge conflicts

  When you add or remove business components in a data model project with ADF Business Components, JDeveloper reflects it in the project file (`.jpr`). When you create (or refactor) a component into a new package, JDeveloper reflects that in the project file and in the ADF Business Components project configuration file (`.jpx`). Although the XML format of these project control files has been optimized to reduce occurrences of merge conflicts, merge conflicts may still arise and you will need to resolve them in JDeveloper using the **Resolve Conflicts** option on the context menu of each affected file.

  After resolving merge conflicts in any ADF Business Components XML component descriptor files, the project file (`.jpr`) for a data model project, or the corresponding ADF Business Components project configuration file (`.jpx`), close and reopen the project to ensure that you're working with latest version of the component definitions. To do this, select the project in the Applications window, choose **File** > **Close** from the JDeveloper main menu, and then expand the project again in the Applications window.

Generation of Complete Web Tier Using Oracle JHeadstart

Oracle JHeadstart application generator is an additional extension for JDeveloper that offers additional productive advantage in creating web-based, JEE business applications. Using Oracle ADF's built-in features, Oracle JHeadstart adds an additional capability to iteratively generate a fully-working web tier using ADF Faces as View layer and ADF Task Flows as Controller layer.

As you'll learn throughout the rest of this guide, JDeveloper and Oracle ADF give you a productive, visual environment for building richly functional, database-centric

Java EE applications with a maximally declarative development experience. However, if you are used to working with tools like Oracle Designer that offer *complete* user interface generation based on a higher-level application structure definition, you may be looking for a similar facility for your Java EE development. If so, then the Oracle JHeadstart application generator may be of interest to you. It is an additional extension for JDeveloper that uses Oracle ADF's built-in features to offer complete web-tier generation for your application modules. Starting with the data model you've designed for your ADF business service, you use the integrated editors that JHeadstart adds to the JDeveloper environment to iteratively refine a higher-level application structure definition. These editors control the functionality and organization of the view objects' information in your generated web user interface. By checking boxes and choosing various options from **dropdown lists**, you describe a logical hierarchy of pages that can include multiple styles of search regions, list of values (LOVs) with validation, shuttle controls, nested tables, and other features. These declarative choices use terminology familiar to Oracle Forms and Designer users, further simplifying web development. Based on the application structure definition, you generate a complete web application that automatically implements the best practices described in this guide, easily leveraging the most sophisticated features that Oracle ADF and JSF have to offer.

Whenever you run the JHeadstart application generator, rather than generating *code*, it creates (or regenerates) all of the declarative view and controller layer artifacts of your Oracle ADF-based web application. These artifacts use ADF Model and work with your ADF application module as their business service. The generated files are the same kinds you produce when using the JDeveloper built-in visual editors. The key difference is that JHeadstart creates them in bulk, based on a higher-level definition that you can iteratively refine until the generated pages match your end users' requirements as closely as possible. The generated files include:

- JSF pages with databound ADF Faces UI components
- ADF Model page definition XML files describing each page's data bindings
- JSF navigation rules to handle page flow
- Resource files containing localizable UI strings

Once you've generated a maximal amount of your application's web user interface, you can spend your time using the productive environment of JDeveloper to tailor the results or to concentrate your effort on additional showcase pages that need special attention. Once you've modified a generated page, you can adjust a setting to avoid regenerating that page on subsequent runs of the application generator. Of course, since both the generated pages and your custom designed ones leverage the same ADF Faces UI components, all of your pages automatically inherit a consistent look and feel. For more information on how to get a fully functional trial of JHeadstart for evaluation, including details on pricing, support, and additional services, see the JHeadstart page on the Oracle Technology Network at`http://www.oracle.com/technetwork/developer-tools/jheadstart/ overview/index.html`

Other Resources for Learning Oracle ADF

To learn how to best use Oracle ADF in applications, you can take the help of many other resources in addition to the developers guide. Some of these resources are step-by-step tutorials, technical papers, pre-recorded sessions, books, trainings, certifications, and so forth.

In addition to this developers guide, Oracle also offers the following resources to help you learn how you can best use Oracle ADF in your applications:

- Tutorials: JDeveloper cue cards provide step-by-step support for the application development process using Oracle ADF. They are designed to be used either with the included examples and a sample schema, or with your own data. The tutorials also include topics that provide more detailed background information, viewlets that demonstrate how to complete the steps in the card, and code samples. Tutorials provide a fast, easy way to become familiar with the basic features of Oracle ADF, and to work through a simple end-to-end task.
- Technical papers, samples, and more on Oracle Technology Network, including ADF Insider, recorded sessions that will help you get up to speed with Oracle ADF.
- Oracle Press and Packt Publishing offer a number of books on JDeveloper and Oracle ADF. See `http://www.oracle.com/technetwork/developer-tools/adf/learnmore/index.html`
- Oracle University offers training and certification for Oracle ADF. See `http://education.oracle.com`.
- Other developer guides for supporting Oracle ADF technology: Oracle provides developer guides for data controls (other than for ADF Business Components), ADF Faces, and **ADF Desktop Integration** to name a few. For information about related guides, see Related Documents, in this book's Preface.

2

Introduction to the ADF Sample Application

This chapter describes how to run the Summit sample applications for Oracle ADF. These samples exist to demonstrate the use of the Fusion web application technology stack to create transaction-based web applications. Details about the schema and features that implement the Summit sample applications for Oracle ADF are also provided.
Before examining the individual components and their source code in depth, you may find it helpful to install and become familiar with the functionality of the Summit sample applications for Oracle ADF. The demonstration application is used as an example throughout this guide to illustrate points and provide code samples.

This chapter includes the following sections:

- About the Summit Sample Applications for Oracle ADF
- Setting Up the Summit Sample Applications for Oracle ADF
- Running the Core Summit ADF Sample Application in the SummitADF Workspace
- Running the Standalone Samples from the SummitADF_Examples Workspace
- Running the Sample Application in the SummitADF_TaskFlows Workspace
- Running the Standalone Samples in the SummitADF_DVT Workspace

About the Summit Sample Applications for Oracle ADF

Summit sample applications are a collection of end-to-end application samples developed by Fusion Middleware Product Management. The Summit samples demonstrate various capabilities of Oracle ADF.

The Summit sample applications for Oracle ADF are a set of applications developed with the purpose of demonstrating common use cases in Fusion web applications, including the integration between the components of the Oracle ADF technology stack (**ADF Business Components**, **ADF Faces**, ADF Data Visualization Tools (DVT), **ADF Controller**, and **ADF Desktop Integration**).

The sample applications are organized into several JDeveloper application workspaces that focus on particular features of the ADF component functionality. Each of the application workspaces uses the same underlying database schema. This approach makes it very easy to access the source code, and also to experience the runtime behavior of the samples in its workspace. Collectively, these sample applications cover the breadth of Oracle ADF functionality for Fusion web applications and are referred to as *Summit sample applications for Oracle ADF*. Chapters of this guide describe sample code drawn from these sample applications.

The core Summit ADF sample application, supported by the underlying Summit ADF schema, serves as a good introduction to Oracle ADF. It provides a runnable application based on a fictional customer management website and demonstrates many user interface design and database transaction features of Oracle ADF. For a detailed description of the core Summit ADF sample application and the features that it implements, see Taking a Look at the Sample Application in the SummitADF Workspace.

A more comprehensive introduction to Oracle ADF functionality is provided by the remaining Summit ADF sample applications, some of which are small standalone samples, while others are extensions of the core sample application. For a list of the Summit ADF sample application workspaces and a description of what functionality each sample provides, see Table 2-1.

In order to view and run any of the samples from its application workspace, you need to install Oracle JDeveloper Studio. You then need to download the ZIP archive files containing the Summit ADF schema and sample applications for this demonstration. Instructions to complete these tasks appear in this chapter. To begin, see Setting Up the Summit Sample Applications for Oracle ADF.

Setting Up the Summit Sample Applications for Oracle ADF

The Summit sample ADF applications contains the database scripts needed to install the summit schema.To set up the Summit sample applications for Oracle ADF, download and extract the sample applications ZIP archive files from Oracle Technology Network (OTN) web site. You will need an Oracle database and JDeveloper, utilizing a shared “summit_adf” schema, which must be installed before any of the samples can be run.

The Summit sample applications for Oracle ADF run using an Oracle database and JDeveloper. The platforms supported are the same as those supported by JDeveloper.

To prepare the environment and run the sample applications, you must download the required application resources. Included among those resources are ZIP archive files that contain the Summit ADF sample application workspaces.

Table 2-1 identifies the sample application archive files that are available for download on the Oracle Technology Network (OTN) website. You may choose to download all of the sample applications in a single archive file; or, when you want to focus on particular Oracle ADF technologies, you may select among the archive files to download just the desired samples.

Table 2-1 The Summit Sample Application Downloadable ZIP Archive Files

Downloadable Samples	ZIP Archive Description
`SummitADF1221.zip`	Contains all the Summit sample applications for Oracle ADF in one convenient download file. It also contains the build file that you will need to install the Summit ADF schema on your database. **Note:** When this ZIP file is expanded, you will have a separate application workspace for each of the sample applications listed below.
`SummitADF_Schema1221.zip`	Contains an application workspace that includes the build file needed to install the Summit ADF schema on your database. The Summit ADF schema is required to run any Summit ADF sample application. For complete details about installing the schema, see How to Install the Summit ADF Schema.

Table 2-1 (Cont.) The Summit Sample Application Downloadable ZIP Archive Files

Downloadable Samples	ZIP Archive Description
`SummitADF_Core1221.zip`	Contains an application workspace that includes the core Summit ADF sample application. This basic application consists of a web user interface and business service layer to demonstrate Oracle ADF technology in Fusion web applications. For a detailed description of this application, see Taking a Look at the Sample Application in the SummitADF Workspace.
`SummitADF_TaskFlows1221.zip`	Contains an application workspace that includes the *Summit sample application for ADF task flows*. This application builds on the core Summit ADF sample application by more fully utilizing **ADF task flows** to demonstrate ADF Security, contextual events, and task flows in Fusion web applications.
`SummitADF_DVT1221.zip`	Contains an application workspace that includes the *Summit sample application for ADF DVT components*. This application workspace consists of a set of standalone sample pages that demonstrate creation and configuration of ADF DVT components.
`SummitADF_Examples1221.zip`	Contains an application workspace that includes the *Summit standalone sample applications for Oracle ADF*. This application workspace consists of small sample applications that allow you to investigate Oracle ADF functionality that does not appear in any of the other sample applications.
`SummitADF_DI1221.zip`	Contains an application workspace that includes the *Summit sample application for ADF Desktop Integration*. This sample application builds on the core Summit sample application by more fully utilizing ADF Desktop Integration functionality. **Note:** This sample is not described in this guide. For complete details about the Summit sample for ADF Desktop Integration, see "Introduction to the ADF Desktop Integration Sample Application" in *Developing Applications with Oracle ADF Desktop Integration*.

How to Download the Application Resources

The Summit sample applications for Oracle ADF require an existing Oracle database for its data. You run the sample applications using JDeveloper.

Do the following before installing the sample applications:

To download the application resources to prepare your environment:

1. Install Oracle JDeveloper Release 12*c*, as described in Installing Oracle JDeveloper Studio in *Installing Oracle JDeveloper*.

   You will need to install the Studio configuration of Oracle JDeveloper Release 12*c* to view the application's projects and run the application in Integrated WebLogic Server. You can download JDeveloper from:

`http://www.oracle.com/technetwork/developer-tools/jdev/overview/index.html`

> **Note:**
>
> When you download and install JDeveloper, ensure that it is the Studio configuration, not the Java configuration. You can verify these details in JDeveloper from the **Help** > **About** menu option.

2. Download the ZIP archive files for the desired Summit ADF sample application workspaces, as identified in Table 2-1. When you want to install all available samples, you can download the single archive file `SummitADF1221.zip`.

   Download the desired Summit ADF sample application archive files from the following web page on the Oracle Technology Network website.

   `http://www.oracle.com/pls/topic/lookup?ctx=E26099_01&id=jdevcodesamples`

3. Install an Oracle database.

   The SQL scripts were written for an Oracle database, so you will need some version of an Oracle RDBMS, such as 11*g*, or Oracle Database Express Edition (Oracle Database XE). The scripts will not install into Oracle Database Lite. If you wish to use Oracle Database Lite or some other database, then you will need to modify the database scripts accordingly. You can download an Oracle database from:

   `http://www.oracle.com/technetwork/database/enterprise-edition/downloads/index.html`

   Specifically, the small footprint of Oracle Database XE is ideally suited for setting up the database on your local machine. You can download it from:

   `http://www.oracle.com/technetwork/database/database-technologies/express-edition/downloads/index.html`

How to Install the Summit ADF Schema

To install the Summit ADF schema, extract the schema files, configure the database connection, and execute the `build_summit_schema.sql` script in JDeveloper.

Before you begin:

It may be helpful to have an understanding of the schema that the Summit sample applications for Oracle ADF use. For more information, see What Happens When You Install the Summit ADF Schema.

You will need to complete these tasks:

- Install Oracle JDeveloper Studio and meet the installation prerequisites. The Summit sample applications for Oracle ADF require an existing Oracle database. For details, see How to Download the Application Resources.
- Download the Summit ADF archive file containing the Summit ADF schema application workspace from the Oracle Technology Network to a local directory. For details, see How to Download the Application Resources.

To install the Summit ADF schema to your database:

1. Navigate to the location where you downloaded the Summit ADF schema archive file and unzip it.
2. Start JDeveloper and in the main menu, choose **File** and then **Open**.
3. In the Open dialog, navigate to where you expanded the ZIP file for the `SummitADF_Schema` directory, select the `Summit_Schema.jws` application workspace and click **Open**.

> **Note:**
>
> If you downloaded the single archive file `SummitADF1221.zip`, the `Summit_Schema.jws` file is in the `Schema` folder. Double-click the `Schema` folder. Select `Summit_Schema.jws` and click Open.

4. In the Applications window, expand the Application Resources panel.
5. Right-click **Connections** and choose **New Connection** and then **Database**.
6. In the Create Database Connection dialog, modify the properties shown in Table 2-2 for your environment. For help with the dialog, press F1 or click **Help.**

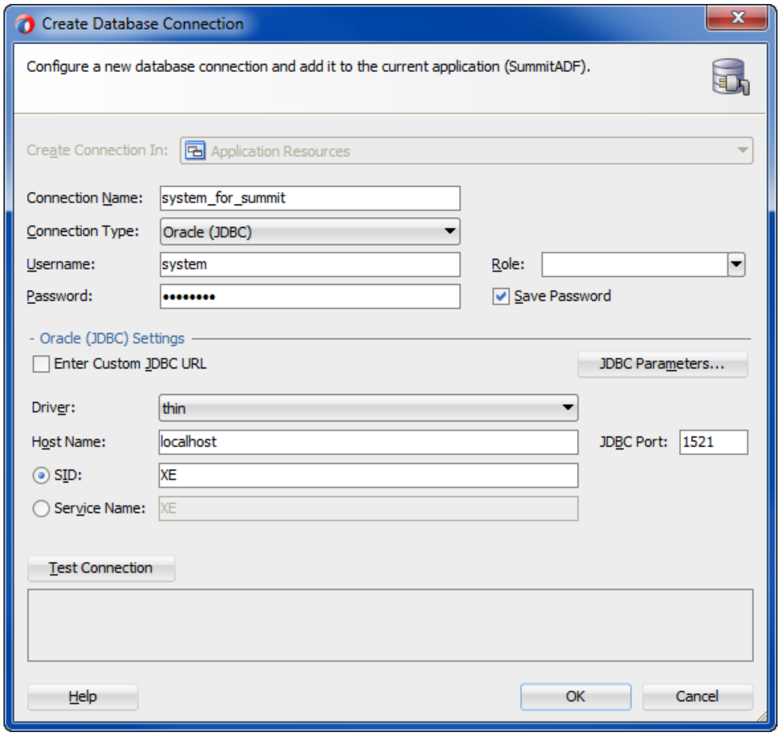
Table 2-2 Properties Required to Install a Summit ADF Sample Application

Property	Description
Connection Name	Descriptive name for the connection. This name must be a valid Java identifier, such as `system_for_summit`.
User Name	The system user for your database. For example: `system`
Password	The password for the system user.
Driver	The JDBC driver for the database. Select a value from the dropdown menu. The default is thin, which is also the default value to specify for Oracle Database XE and any other Oracle database that is not using Oracle Call Interface (OCI).
Host Name	The name of the server running the Oracle database. Use an IP address or a host name that can be resolved by TCP/IP. The default value is `localhost`.
SID	The unique system identifier (SID) of an Oracle Database instance. The default is `XE`, which is also the default value to specify if your database server is running Oracle Database XE. If your server is running another Oracle database, the SID is typically `ORCL`.
JDBC Port	The port of your database. The default value is 1521.

> **Note:**
>
> If your server resides on a remote machine, you may also need to modify the script that builds the schema. To open the script, right-click `build_summit_schema.sql` and choose **Open**.

Figure 2-1 shows the completed Create Database Connection dialog. In this example, the connection is made to an Oracle Database XE instance residing on a local machine.

Figure 2-1 Create Database Connection Dialog for Summit ADF Schema



Create Database Connection

Configure a new database connection and add it to the current application (SummitADF).

Create Connection In: Application Resources

Connection Name: system_for_summit

Connection Type: Oracle (JDBC)

Username: system

Role:

Password: ••••••••

☑ Save Password

Oracle (JDBC) Settings

□ Enter Custom JDBC URL

Driver: thin

Host Name: localhost

JDBC Port: 1521

● SID: XE

○ Service Name: XE

7. Click **Test Connection** to verify that you have a working connection.
8. Click **OK** to create the connection and exit the dialog.
9. In the Applications window, in the Projects panel, expand Database and then Resources.
10. Right-click `build_summit_schema.sql` and choose **Run in Sql*Plus >** *connection name*.

    The connection name displayed is the one you configured in Step 6.
11. In the SQL*Plus Connection dialog, verify that the information matches the configuration you specified in Step 6 and click **OK**.
12. In the SQL*Plus Location dialog, click **Browse** and locate the `sqlplus.exe` executable for your database.

    Typically, the executable is installed in the `BIN` directory under `$ORACLE_HOME`, where `$ORACLE_HOME` represents the path to your Oracle database installation.

13. Click **Open** to select the `sqlplus.exe` executable and then **OK** to exit the dialog.
14. In the SQL*Plus window, enter the password for the system user you specified in Step 6.

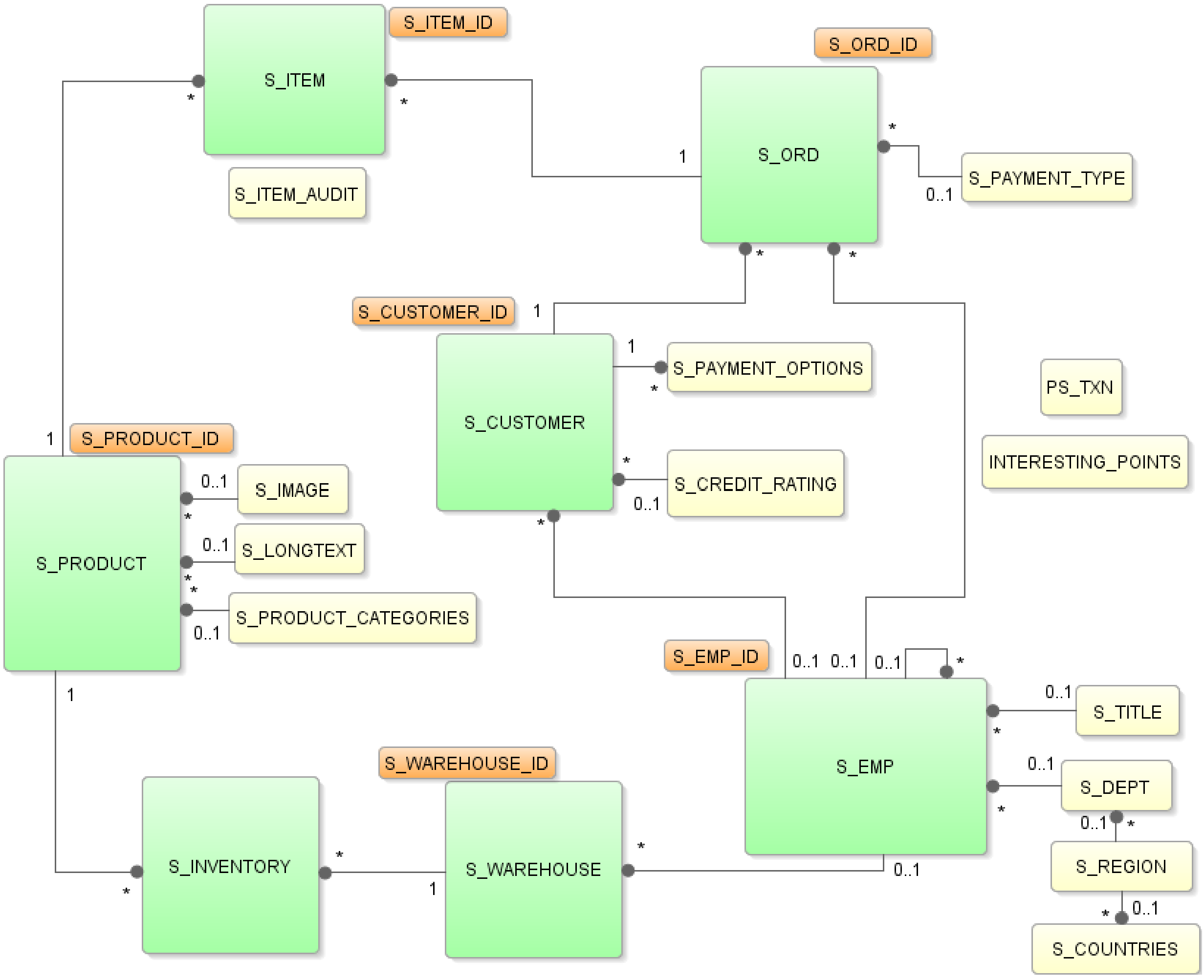
    Once you enter the password, the Ant build script creates the Summit ADF sample application users and populates the tables in the Summit ADF schema. In the Messages - Log window, you will see a series of SQL scripts and finally:

    ```
    Commit complete.

    Commit complete.
    SQL>
    ```
15. At the SQL prompt, enter `quit` to exit the SQL*Plus window.
16. In JDeveloper, in the main menu, choose **Application** and then **Close** to close the Summit ADF schema application.

What Happens When You Install the Summit ADF Schema

Figure 2-2 shows a simplified representation of the schema for the Summit sample applications for Oracle ADF. The green shapes in the diagram represent the seven core tables. The other tables are shown as yellow shapes. Some of the tables use sequences, but only those used by the core tables are shown. To minimize the number of crossed lines, the schema is displayed in a bottom-to-top hierarchy.

Figure 2-2 Schema Diagram for the Summit Sample Applications for Oracle ADF



The core tables represented by the green diagram elements include:

- S_ITEM: For each order, there may be many order items recorded. This table stores the unit price and quantity of each order item. The order line item and its order ID uniquely identify each order item.
- S_ORD: This table represents activity by specific customers. When an order is created, the date of the order, the total amount of the order, the ID of the customer who created it, and the name of the sales representative is recorded. After the order is fulfilled, the order status and order shipped date are updated. All orders are uniquely identified by a sequence-assigned ID.
- S_CUSTOMER: This table stores the name and address of all Summit customers. Each customer is uniquely identified by a sequence-assigned ID. The table also includes a foreign key that associates each customer with a Summit sales representative.
- S_EMP: This table stores all the employees who interact with the system. The first name, last name, email address, and address details of each user is stored. A user is uniquely identified by an ID. Other IDs provide foreign keys to tables with

title and department information, and, in the case of the employee's manager, a self-referential association.

- S_PRODUCT: This table stores all of the products available in the store. For each product, the name and suggested wholesale price are recorded. All products are uniquely identified by a sequence-assigned ID. The image of the product and its description are stored in separate tables, which each reference the product ID.
- S_WAREHOUSE: This table stores the location and manager of each warehouse that carries stock for the Summit products. For each warehouse, the location and manager are recorded. The warehouses are uniquely identified by a sequence-assigned ID.
- S_INVENTORY: This table stores inventory data for the Summit products. For each product, the amount in stock, reorder point, and maximum stock levels are recorded for each warehouse that carries the product.

The sequences that the core tables use include:

- S_ITEM_ID: Populates the ID for each new order item.
- S_ORD_ID: Populates the ID for each new order.
- S_CUSTOMER_ID: Populates the ID for each new customer.
- S_EMP_ID: Populates the ID for each new employee.
- S_PRODUCT_ID: Populates the ID for each new product.

How to Create the Database Connection for Summit Sample Applications for Oracle ADF

The Summit sample applications for Oracle ADF require a connection to the Summit database for data access. By default, each application is configured to use a local connection to an Oracle database that you must verify and modify as needed for your environment. The steps to modify a database connection for any of the Summit sample applications for Oracle ADF are the same for all workspaces and are performed after you have opened the application in JDeveloper.

Before you begin:

It may be helpful to have an understanding of the Summit ADF schema. For more information, see What Happens When You Install the Summit ADF Schema.

You will need to complete these tasks:

- Complete all steps to download the application resources to prepare your environment. For details, see How to Download the Application Resources.
- Install the Summit ADF schema. For details, see How to Install the Summit ADF Schema.
- Open the sample application to which you are connecting using the instructions for the application:
  - How to Run the Core Summit ADF Sample Application
  - How to Run the Summit ADF Standalone Sample Applications
  - How to Run the Summit Sample Application for ADF Task Flows
  - How to Run the Summit ADF DVT Sample Application Pages

To create the database connection for Summit sample applications for Oracle ADF:

1. In the Applications window, expand the Application Resources panel.
2. In the Application Resources panel, expand **Connection** and then **Database**.

   The existing database connection is displayed. For the core Summit ADF application, the database connection is **summit_adf**.
3. Right-click the database connection and choose **Properties**.

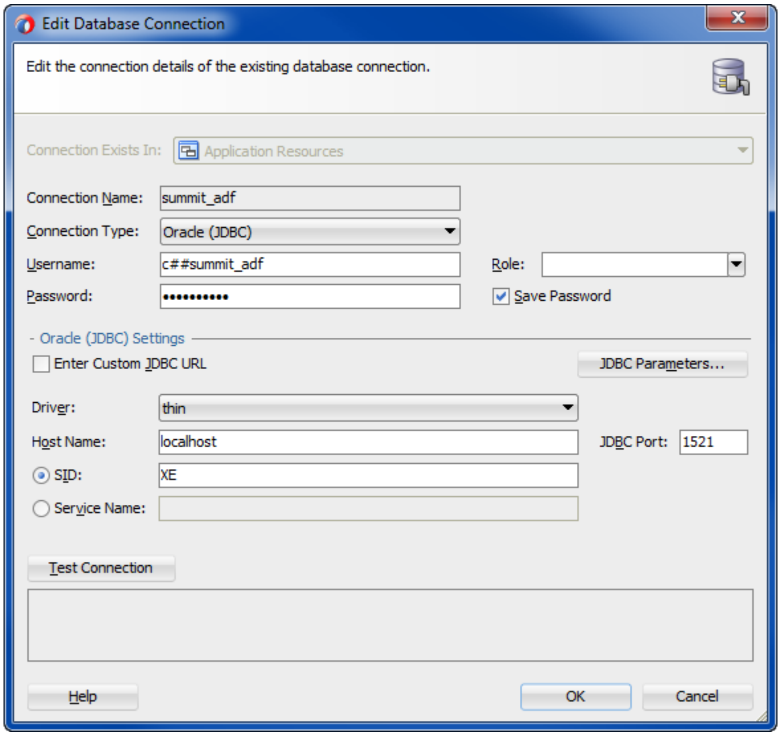
   For example, right-click **summit_adf** and choose **Properties**.
4. In the Edit Database Connection dialog, modify the properties shown in Table 2-3 for your environment. For help with the dialog, press F1 or click **Help**.

Table 2-3 Properties to Create a Database Connection for a Summit ADF Sample Application

Property	Description
User Name	The user name for the Summit ADF sample database. You must use `c##summit_adf` for this property. Note that Oracle Database 12c requires the user name to begin with `c##`.
Password	The password for the Summit ADF sample database. You must use `summit_adf` for this value.
Driver	The JDBC driver for the database. Select a value from the dropdown menu. The default is `thin`, which is also the default value to specify for Oracle Database XE instances and any other Oracle database that is not using Oracle Call Interface (OCI).
Host Name	The host name for your database. For example: `localhost`
JDBC Port	The port for your database. For example: `1521`
SID	The unique system identifier (SID) of an Oracle Database instance. The default is `XE`, which is also the default value to specify if your database server is running Oracle Database XE. If your server is running another Oracle database, the SID is typically `ORCL`.

Figure 2-3 shows the completed Edit Database Connection dialog. In this example, the connection is made to an Oracle Database XE instance residing on a local machine.

Figure 2-3   Edit Database Connection Dialog for a Summit ADF Sample Application



Edit Database Connection

Edit the connection details of the existing database connection.

Connection Exists In: Application Resources

Connection Name: summit_adf

Connection Type: Oracle (JDBC)

Username: c##summit_adf

Role:

Password: ••••••••••

☑ Save Password

Oracle (JDBC) Settings

□ Enter Custom JDBC URL

Driver: thin

Host Name: localhost

JDBC Port: 1521

● SID: XE

○ Service Name:

5. Click **Test Connection** to verify that you have a working connection.
6. Click **OK** to create the connection and exit the dialog.
7. In the Applications window, in the Projects panel, right-click **Model** and choose **Rebuild Model.jpr**.

Running the Core Summit ADF Sample Application in the SummitADF Workspace

The core Summit ADF sample application demonstrates various capabilities of Oracle ADF using a fictional scenario called Summit Sporting Goods. The purpose of the core Summit sample application is to demonstrate common use cases in Fusion web applications.

In the core Summit ADF sample application, sporting goods equipment is sold to international customers by a fictional sporting goods supplier called Summit Management. Employees of Summit Management can visit the website, and manage customer orders by fulfilling orders for products while monitoring the status of product inventory. The functionality in this core sample application has been intentionally restricted to demonstrate a limited number of potential Fusion web application use cases that are suitable for anyone new to Oracle ADF technology.

The core Summit ADF sample application consists of a web user interface and a business service layer. Specifically, the following projects are part of the core Summit ADF sample application:

- `Model`: Provides access to the customer management data and provides transaction support to update data for customer information and orders.

- ViewController: Provides the JSF web page and JSFF page fragments that the Summit user interacts with to browse the customers list, place orders, view order details, check order status, and inspect product inventory.

How to Run the Core Summit ADF Sample Application

You run the core Summit ADF sample application in JDeveloper by running the index.jspx page in the ViewController project. The ViewController project uses JavaServer Faces (JSF) as the view technology, and relies on the **ADF Model** layer to interact with ADF Business Components in the Model project. To learn more about the core Summit ADF sample application and to understand its implementation details, see Taking a Look at the Sample Application in the SummitADF Workspace.

Before you begin:

It may be helpful to have an understanding of the Summit ADF schema. For more information, see What Happens When You Install the Summit ADF Schema.

You will need to complete these tasks:

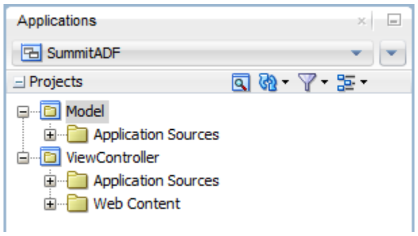
- Install Oracle JDeveloper Studio and meet the installation prerequisites. The core Summit ADF sample application requires an existing Oracle database. For details, see How to Download the Application Resources.
- Download the Summit ADF archive file from the Oracle Technology Network to a local directory and unzip the core Summit ADF sample application. For details, see How to Download the Application Resources.
- Install the Summit ADF schema. For details, see How to Install the Summit ADF Schema.

To run the sample application in the SummitADF application workspace:

1. Open the application in JDeveloper:
   a. In the main menu, choose **File** and then **Open**.
   b. Navigate to the location where you expanded the ZIP file for the SummitADF directory, select the SummitADF.jws application workspace and click **Open**.

      Figure 2-4 shows the Applications window after you open the file for the SummitADF.jws application workspace file in JDeveloper. For a description of each of the projects in the workspace, see Taking a Look at the Sample Application in the SummitADF Workspace.

Figure 2-4 Projects in the SummitADF Application Workspace



2. Create a connection to the Summit database. For details, see How to Create the Database Connection for Summit Sample Applications for Oracle ADF.

3. In the Applications window, expand **ViewController**, right-click `index.jsf` and choose **Run**.

   The Create Default Domain dialog appears the first time you run the application and start a new domain in Integrated WebLogic Server. Use the dialog to define an administrator password for the new domain. Passwords you enter can be eight characters or more and must have a numeric character.

   The `index.jspx` page within the `ViewController` project is the run target. When you run the page, JDeveloper will launch the browser and display the Summit Management web page of the core Summit ADF sample application.

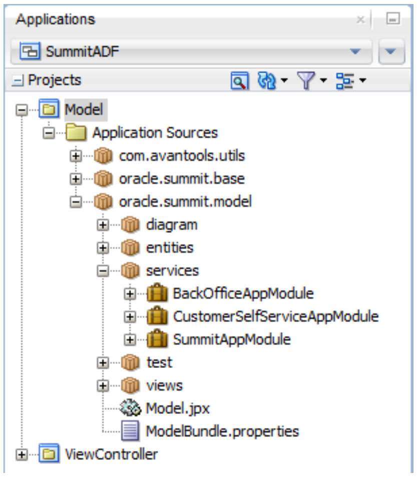
Once the Summit Management web page appears, you can browse the website. Note that the core Summit ADF sample application does not implement security for the Oracle ADF application resources and login is not required.

Taking a Look at the Sample Application in the SummitADF Workspace

Once you have opened the projects in JDeveloper, you can then begin to review the artifacts within each project. The development environment for the core Summit ADF sample application is divided into two projects: the `Model` project and the `ViewController` project.

The `Model` project contains the classes associated with the data model that allow the product data to be displayed in the web application. Figure 2-5 shows the `Model` project and its associated directories.

Figure 2-5 The Data Model Project of the SummitADF Workspace

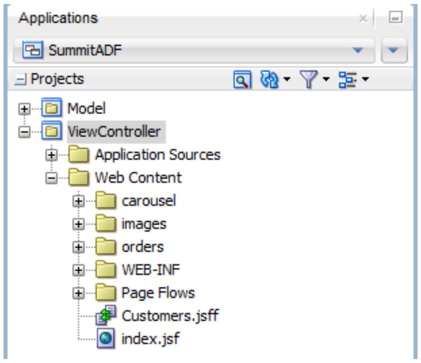


The `ViewController` project contains the following folders associated with the user interface:

- `Application Sources`: Contains the files used to access the product data. Included are the metadata files used by Oracle ADF to bind the data to the view.
- `META-INF`: Contains a file used in deployment.

The ViewController project contains the files for the web interface, including the backing beans, deployment files, and JSF and JSFF files. Figure 2-6 shows the ViewController project and its associated folders.

Figure 2-6 The ViewController Project of the SummitADF Workspace



The ViewController project contains the following directories:

- Application Sources: Contains the code used by the web client, including the managed and backing beans, property files used for internationalization, and the metadata used by Oracle ADF to display bound data.
- Web Content: Contains the web files, including the JSP files, images, skin files, deployment descriptors, and libraries.

Browsing the Application

Start the core Summit ADF sample application by running the index.jspx page in the ViewController project. For details about running the application using the default target, index.jsf, see Running the Core Summit ADF Sample Application in the SummitADF Workspace.

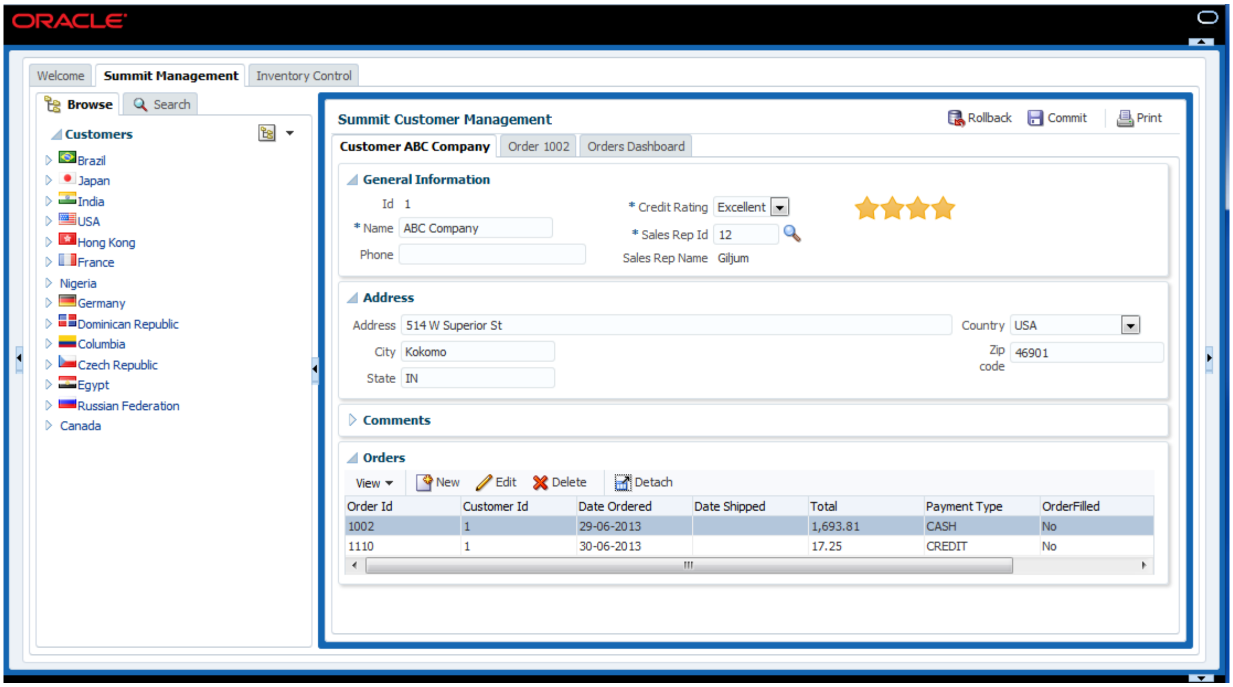
The index.jsf page is organized by an **ADF page** template which provides the structure for the page and uses an af:panelTabbed component to identify the page fragments to display in each of the tab panels.

When you enter the Summit ADF website, the site opens on the index page with three tabs displayed: Welcome, Summit Management, and Inventory Control. Click on each tab to view the separate page fragments referenced as regions by the index page.

The Summit Management region is the main **region** where you can browse the list of customers and their orders. This region is defined by the customers.jsff page fragment and is itself organized by ADF Faces components that provide tabbed panels. The Browse and Search panels are one af:panelTabbed component and the Customer, Order, and Order Dashboard panels belong to another af:panelTabbed component.

Figure 2-7 shows the index page with an order displayed in the Summit Management region.

Figure 2-7 Index Page with Multiple Regions



Id 1
* Name ABC Company
Phone
* Credit Rating Excellent
* Sales Rep Id 12
Sales Rep Name Giljum

Address 514 W Superior St
City Kokomo
State IN
Country USA
Zip code 46901

Order Id	Customer Id	Date Ordered	Date Shipped	Total	Payment Type	OrderFilled
1002	1	29-06-2013		1,693.81	CASH	No
1110	1	30-06-2013		17.25	CREDIT	No

Where to Find Implementation Details

Following are the sections of *Developing Fusion Web Applications with Oracle Application Development Framework* that describe how to create a databound web page:

- Providing the structure for the web page

  The index page separates features of the application into regions that are implemented using a combination of ADF Faces templates and JavaServer Faces (JSF) page fragments. ADF Faces templates and the fragments allow you to add ADF databound components. For information about the steps you perform before adding databound user interface components to a web page, see About Developing a Web Application with ADF Faces.

- Designing the user interface using a variety of components

  JDeveloper allows you to create databound components declaratively for your JSF pages, meaning you can design most aspects of your pages without needing to look at the code. By dragging and dropping items from the **Data Controls panel**, JDeveloper declaratively binds ADF Faces UI components and ADF Data Visualization graph components to attributes on a **data control** using an **ADF binding**. For information about creating databound web pages with ADF Faces components, see About Creating a Basic Databound Page.

- Managing entry points to the application

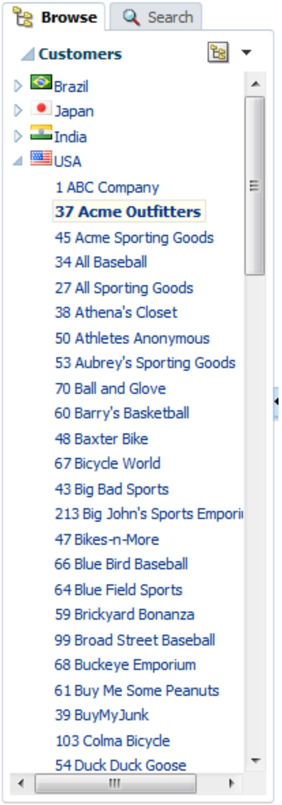
  The page fragments of the index page are supported by individual **ADF task flows**. In general, the Fusion web application relies on this ADF Controller feature to define entry points to the application. When your application uses more than one page, you define an **unbounded task flow** to control the entry point of the application. JDeveloper helps you to create task flows with visual design elements that you drag and drop from the Components window. For more information about specifying the entry points to the application using an **ADF unbounded task flows**, see About ADF Task Flows.

Browsing the Customers List

To view the list of customers, click the **Browse** tab and in the Customers tree, expand a country node to view the list of customers for that country. The customers list is layed out with collapsing nodes organized by an ADF Faces `af:tree` component. The dropdown menu on the tree component lets you choose to organize the branches by country or by sale representative. The nodes of the tree display data from the database corresponding to the name and ID of each customer.

Figure 2-8 shows the tree expanded with a customer node selection.

Figure 2-8   Databound Tree Component Displays Customer Names and ID



Brazil
Japan
India
USA
1 ABC Company
37 Acme Outfitters
45 Acme Sporting Goods
34 All Baseball
27 All Sporting Goods
38 Athena's Closet
50 Athletes Anonymous
53 Aubrey's Sporting Goods
70 Ball and Glove
60 Barry's Basketball
48 Baxter Bike
67 Bicycle World
43 Big Bad Sports
213 Big John's Sports Empori
47 Bikes-n-More
66 Blue Bird Baseball
64 Blue Field Sports
59 Brickyard Bonanza
99 Broad Street Baseball
68 Buckeye Emporium
61 Buy Me Some Peanuts
39 BuyMyJunk
103 Colma Bicycle
54 Duck Duck Goose

Where to Find Implementation Details

Following is the section of *Developing Fusion Web Applications with Oracle Application Development Framework* that describes how to develop the components used to support browsing product details:

- Displaying data in a web page

  To display data from the data model, user interface components in the web page are bound to ADF Model layer binding objects using JSF Expression Language (EL) expressions. For example, when the user expands a country node in the Customers tree to display the list of customers for that country, the JSF runtime evaluates the EL expression for the UI component and pulls the value from the

ADF Model layer. At design time, when you work with the Data Controls panel to drag an attribute for an item of a data collection into your web page, and then choose an ADF Faces component to display the value, JDeveloper creates all the necessary JSF tag and binding code needed to display and update the associated data. For more information about the Data Controls panel and the declarative binding experience, see About Using ADF Model in a Fusion Web Application.

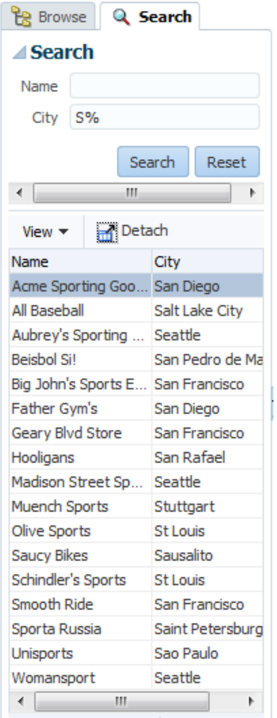
Searching for Customers

To search the customer list, click the Search tab in the Summit Management region. When you click the tab, the Browse panel changes to allow you to enter a search criteria in the query search form and view the search results in the table below. The layout of the search panel is defined by an ADF Faces `af:query` component.

You use the input fields on the Search form to perform a simple keyword search against the attributes common to all customers, such as customer names or cities. For example, you might query the database for the list of customers that begin with a particular letter.

Figure 2-9 shows the search results returned for the customers residing in a city that begins with "S".

Figure 2-9 Query Search Component Filters Customer List



Name

City S%

Name	City
Acme Sporting Goo...	San Diego
All Baseball	Salt Lake City
Aubrey's Sporting ...	Seattle
Beisbol Si!	San Pedro de Ma
Big John's Sports E...	San Francisco
Father Gym's	San Diego
Geary Blvd Store	San Francisco
Hooligans	San Rafael
Madison Street Sp...	Seattle
Muench Sports	Stuttgart
Olive Sports	St Louis
Saucy Bikes	Sausalito
Schindler's Sports	St Louis
Smooth Ride	San Francisco
Sporta Russia	Saint Petersburg
Unisports	Sao Paulo
Womansport	Seattle

Where to Find Implementation Details

Following are the sections of *Developing Fusion Web Applications with Oracle Application Development Framework* that describe how to define queries and create query search forms:

- Defining the query for the search form to display

  A query is associated with an ADF Business Components **view object** that you create for the data model project to define a particular query against the database. In particular, a query component is the visual representation of the **view criteria** defined on that view object. If there are multiple view criteria defined, each of the view criteria can be selected from the **Saved Search** dropdown list. These saved searches are created at design time by the developer. When the query associated with that view object is executed, both view criteria are available for selection. For more information, see About Creating Search Forms.

- Creating a search form

  You create a query search form by dropping a named view criteria item from the Data Controls panel onto a page. You have a choice of dropping only a search panel, dropping a search panel with a results table, or dropping a search panel with a tree table. For more information, see Creating Query Search Forms.

- Displaying the results of a query search

  Normally, you would drop a query search panel with the results table or tree table. JDeveloper will automatically wire up the results table or tree table with the query panel. If you drop a query panel by itself and want a separate results component, you can set the query component's `resultComponentId` attribute to the relative expression of the results component. For more information, see How to Create a Query Search Form and Add a Results Component Later.

Viewing Customer Details

To begin browsing the orders for a customer, click the **Summit Management** tab and make a selection in the expanded Customers tree from the Browse tab or Search tab. This action changes the details displayed in the Summit Customer Management tab to display details about the selected customer.

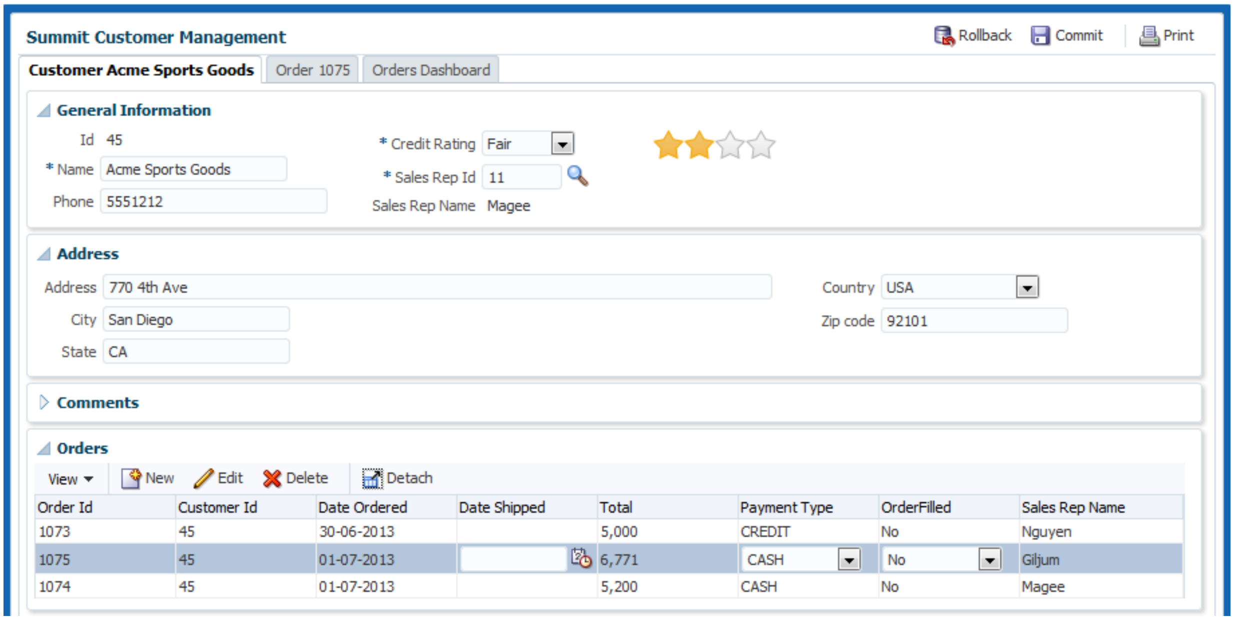
In the Customer details tab you can collapse and expand the panels to view various details of the selected customer. The first three panels display input forms consisting of individual input fields corresponding to the customer details. The customer you select also has one or more orders displayed in the Orders panel as a table, where each row comprises a single order and its details.

This relationship between a customer and their orders represents a business object relationship known as master and detail. In this case, the master, or parent object, can have many detail rows, or child objects. In turn the business objects in the core Summit ADF sample application, correspond to the `S_CUSTOMER` and `S_ORD` tables in the database.

The table of orders is defined by an ADF Faces `af:table` component. You can sort the table rows by clicking the sort buttons in any cell of the table header.

Figure 2-10 shows the customers page fragment with the details of the customer displayed in the top portion and the list of orders specific to the customer in the bottom portion.

Figure 2-10 Customers Page Fragment with Orders List



Id 45
* Name Acme Sports Goods
Phone 5551212
* Credit Rating Fair
* Sales Rep Id 11
Sales Rep Name Magee

Address 770 4th Ave
City San Diego
State CA
Country USA
Zip code 92101

Order Id	Customer Id	Date Ordered	Date Shipped	Total	Payment Type	OrderFilled	Sales Rep Name
1073	45	30-06-2013		5,000	CREDIT	No	Nguyen
1075	45	01-07-2013		6,771	CASH	No	Giljum
1074	45	01-07-2013		5,200	CASH	No	Magee

Where to Find Implementation Details

Following are the sections of *Developing Fusion Web Applications with Oracle Application Development Framework* that describe how to use tables and forms to display master-detail related objects:

- Dragging and dropping master and detail components

  You can create pages that display master-detail data using the Data Controls panel. The Data Controls panel displays master-detail related objects in a hierarchy that mirrors the one you defined in the data model for the ADF **application module**, where the detail objects are children of the master objects. All you have to do is drop the collections on the page and choose the type of component you want to use. For example, in the core Summit ADF sample application, the page fragment `customers.jsff` displays the master list of customer categories in an `af:tree` component and displays the detail list of orders in an `af:table` component. For more information about the data model, see Exposing Application Modules with ADF Data Controls. For more information about various types of pages that display master-detail related data, see About Displaying Master-Detail Data.

- Sorting data that displays in tables

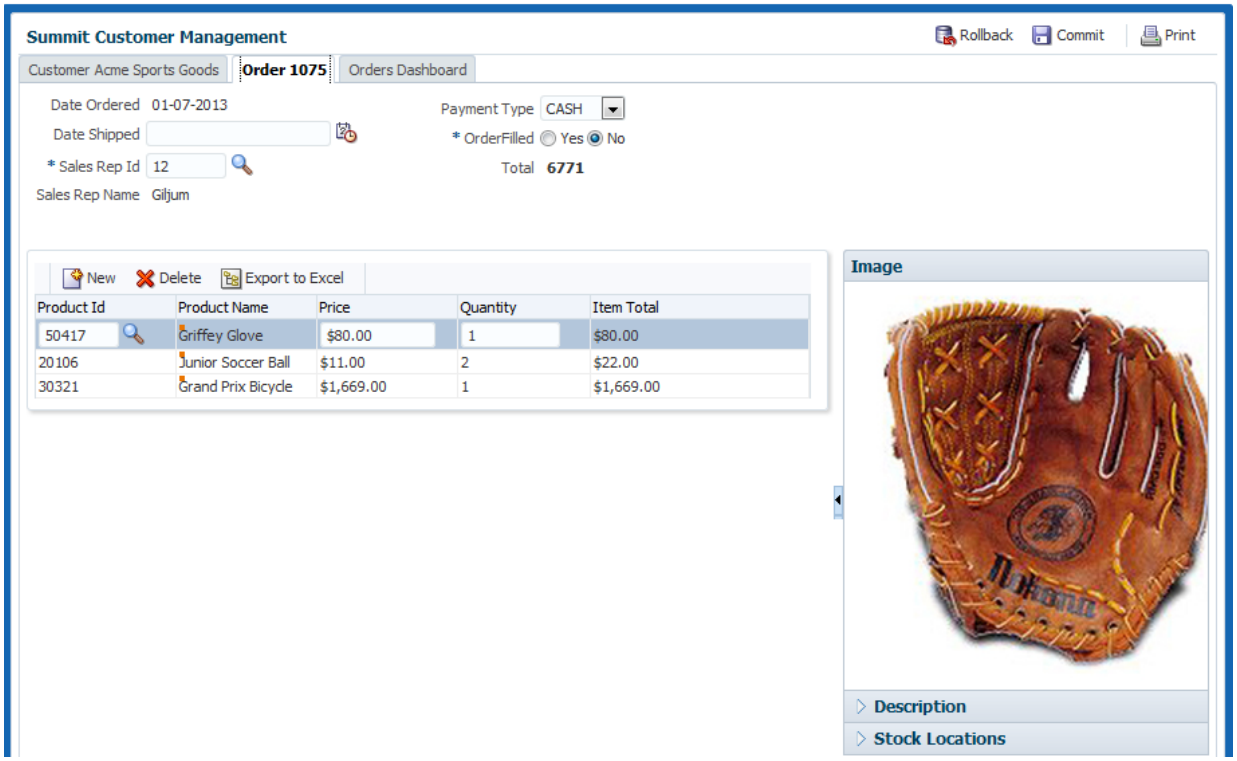
  When you create an ADF Faces table component you bind the table to the complete collection or to a range of data objects from the collection. The specific components that display the data in the columns are then bound to the attributes of the collection. The **iterator binding** handles displaying the correct data for each object, while the table component handles displaying each object in a row. You can set the **Sort** property for any column when you want the iterator to perform an order-by query to determine the order. You can also specify an `ORDER BY` clause for the query that the view object in the data model project defines. For more information about binding table components to a collection, see About Creating ADF Databound Tables. For more information about creating queries that sort data in the data model, see Populating View Object Rows from a Single Database Table.

Viewing Order Details

To update the details of an existing order, select an order from the Orders table displayed in the Customer tab and click the **Order** tab. The Order tab displays the same data that appeared in the row you selected, but organized as an edit form with fields that you can update. To change the order status, for example, you can select a choice from the **OrderFilled** radio button. The selection you make on the ADF Faces `af:selectOneRadio` component updates the corresponding `S_ORD` table column.

Figure 2-11 shows the Order page fragment with Order 1075 selected in the Customer details tab. Data for Order 1075 in both the order form and the Customer details tab are coordinated by the application to reflect changes in the database. Change the status of **OrderFilled** to **Yes**, and return to the Customer details tab to verify that the status is consistent.

Figure 2-11 Orders Page Fragment Displays Order Details

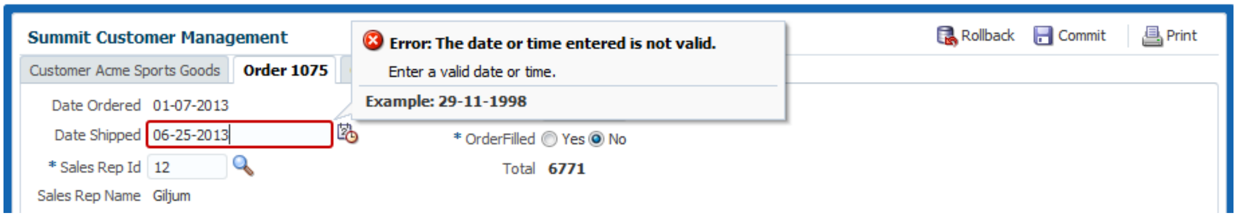


Date Ordered	01-07-2013
Date Shipped	
* Sales Rep Id	12
Sales Rep Name	Giljum
Payment Type	CASH
* OrderFilled	○ Yes ● No
Total	6771

Product Id	Product Name	Price	Quantity	Item Total
50417	Griffey Glove	$80.00	1	$80.00
20106	Junior Soccer Ball	$11.00	2	$22.00
30321	Grand Prix Bicycle	$1,669.00	1	$1,669.00

Other types of data, such as dates and currency, can be displayed using ADF Faces components to handle formatting of the values. Figure 2-12 shows the Order page fragment with a system error that enforces date formats that have been defined on the attribute underlying the **Date Shipped** field.

Figure 2-12 Orders Page Fragment Displays Format Mask Error



Error: The date or time entered is not valid.

Enter a valid date or time.

Example: 29-11-1998

Date Ordered 01-07-2013

Date Shipped 06-25-2013

* Sales Rep Id 12

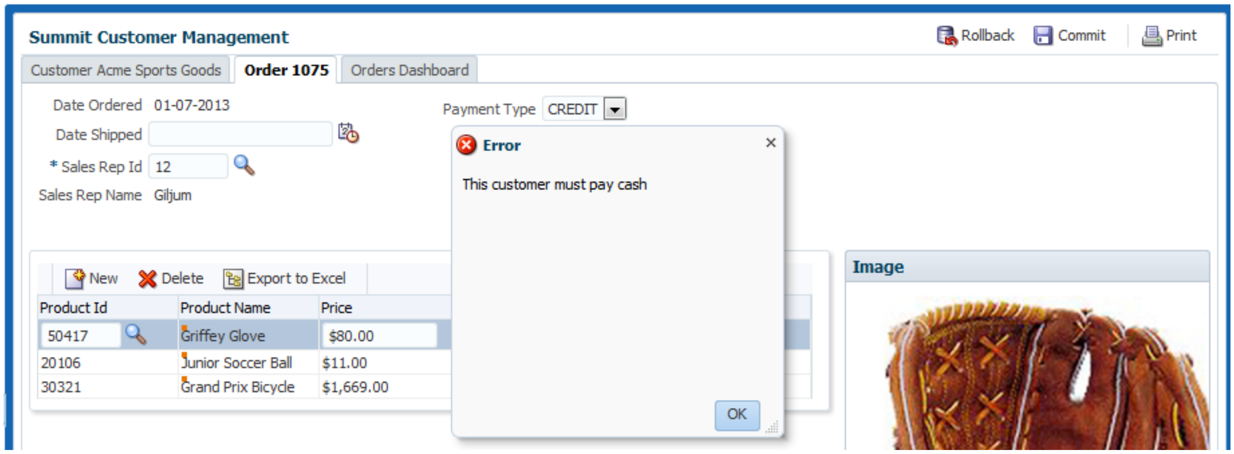
Sales Rep Name Giljum

* OrderFilled ○ Yes ● No

Total 6771

Validation rules based on Groovy calculated expressions can be defined on data updates. Figure 2-13 shows the Order page fragment with a validation error that calculates the customer's credit score before allowing a **Payment Type** of **CREDIT** to be selected.

Figure 2-13 Orders Page Fragment Displays Payment Type Validation Error



Date Ordered 01-07-2013

Date Shipped

* Sales Rep Id 12

Sales Rep Name Giljum

Payment Type CREDIT

Error

This customer must pay cash

Product Id	Product Name	Price
50417	Griffey Glove	$80.00
20106	Junior Soccer Ball	$11.00
30321	Grand Prix Bicycle	$1,669.00

Where to Find Implementation Details

Following are the sections of *Developing Fusion Web Applications with Oracle Application Development Framework* that describe how to create edit forms:

- Creating a databound edit form

  When you want to create a basic form that collects values from the user, instead of having to drop individual attributes, JDeveloper allows you to drop all attributes for an object at once as an input form. You can create forms that display values, forms that allow users to edit values, and forms that collect values. For example, in the core Summit ADF sample application, the page fragment `Orders.jsff` displays a form to display order information. For more information, see Creating a Form to Edit an Existing Record.

- Requiring values to complete a form

  The input form displays attributes of a data collection that you drop from the Data Controls panel. You can set the `required` property of individual components in the form to control whether an attribute value is mandatory. For details about how to customize the `required` property, see Creating Text Fields Using Data Control Attributes. Alternatively, you can set a display **control hint** property directly on the attribute where it is defined by an ADF Business Components **entity object**. The entity object is a data model component that represents a row from a specific table in the database and that simplifies modifying its associated attributes. For details about using control hints to make an attribute mandatory, see Setting Attribute Properties.

- Defining a list of values for selection lists

  Input forms displayed in the user interface can utilize databound ADF Faces selection components to display a **list of values (LOV)** for individual attributes of the data collection. To facilitate this common design task, ADF Business Components provides declarative support to specify the LOV usage for attributes in the data model project. For example, in the core Summit ADF sample application, the `af:selectOneChoice` component for the payment type field displayed in the page fragment `Orders.jsff` is bound to an LOV-enabled attribute configured for the `OrdVO` view object. For more information about configuring attributes for LOV usage, see Working with List of Values (LOV) in View Object Attributes.

- Defining format masks for forms

  Format masks help ensure the user supplies attribute values in the required format. To facilitate this task, ADF Business Components provides declarative support known as **control hints** for attributes in the data model project. For example, in the core Summit ADF sample application, the attribute for the `OrdVO` view object used to specify the date shipped is configured with a format mask hint and enforced in the page fragment `Orders.jsff`. For information on defining format masks for input form components, see Defining UI Hints for View Objects.

- Defining validation rules for data updates

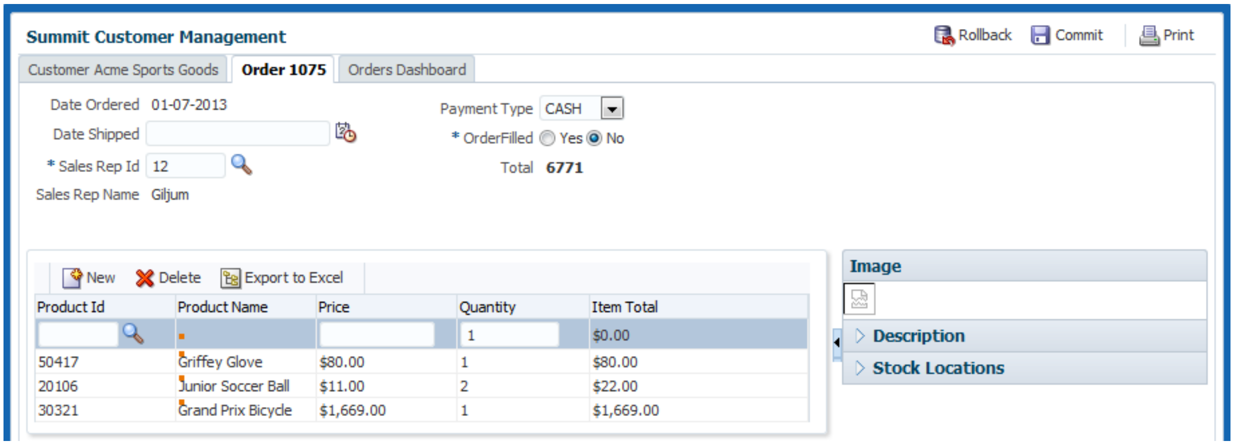
  Validation rules help ensure the user supplies attribute values that meet particular business needs. To facilitate this task, ADF Business Components provides declarative validation for attributes in the data model project. For example, in the core Summit ADF sample application, the attribute for the `OrdEO` entity object used to specify the payment type is configured with a validation rule and enforced in the page fragment `Orders.jsff`. For information on defining validation rules for attributes of entity objects and view objects, see Defining Validation and Business Rules Declaratively.

The Ordering Process

Creating a new customer order requires a database transaction that involves updating the records of the customer's order while decreasing the quantity of products remaining in the Summit inventory. When you click **New** on Customer page above the list of orders, the application creates a new order form with only the date displayed.

Figure 2-14 shows the new order form with no products added.

Figure 2-14 Order Page Fragment Displays New Empty Order



Date Ordered 01-07-2013

Date Shipped

* Sales Rep Id 12

Sales Rep Name Giljum

Payment Type CASH

* OrderFilled ○ Yes ● No

Total 6771

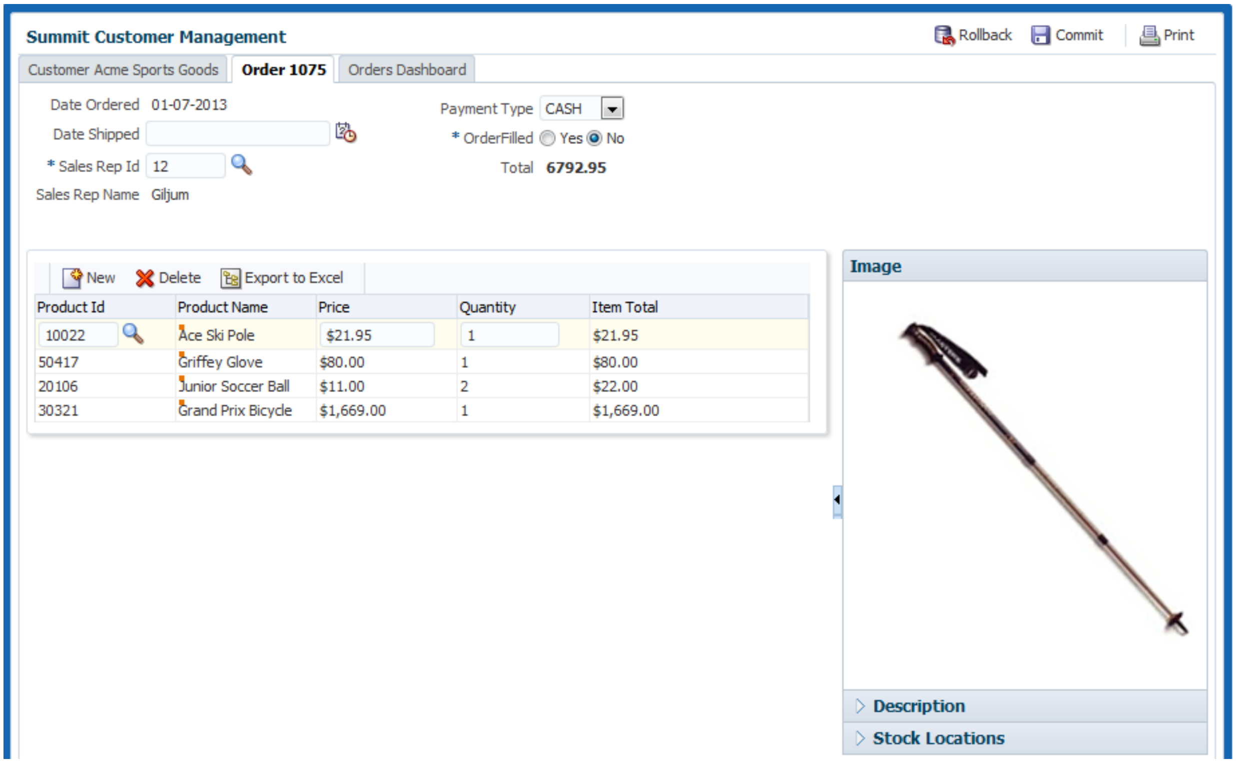
Product Id	Product Name	Price	Quantity	Item Total
			1	$0.00
50417	Griffey Glove	$80.00	1	$80.00
20106	Junior Soccer Ball	$11.00	2	$22.00
30321	Grand Prix Bicycle	$1,669.00	1	$1,669.00

To create the order, click **New** above the empty list of products. Then, next to the **ProductId** field, click the Search icon. The `af:inputListOfValues` component launches the popup window that you can use to search the database for the product. To make searching easier, the popup window displays a limited number of fields from the `S_PRODUCT` table, including product ID and product Name. This type of filtering for display at runtime is performed by the ADF binding that backs the databound `af:inputListOfValues` component.

Select a product in the popup window and click **OK**. The Order tab updates to display the selection in the product table.

Figure 2-15 shows the Order tab with an order comprising the product order.

Figure 2-15   Order Page Fragment Displays Order Entry



Date Ordered 01-07-2013
Date Shipped
* Sales Rep Id 12
Sales Rep Name Giljum

Payment Type CASH
* OrderFilled ○ Yes ● No
Total **6792.95**

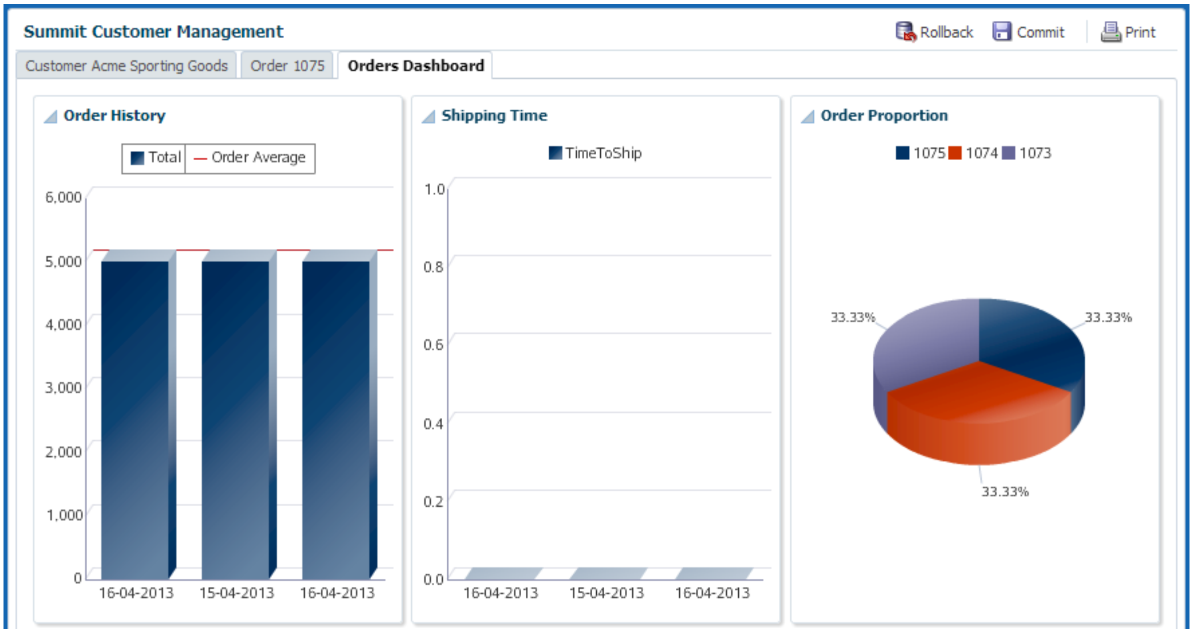
Product Id	Product Name	Price	Quantity	Item Total
10022	Ace Ski Pole	$21.95	1	$21.95
50417	Griffey Glove	$80.00	1	$80.00
20106	Junior Soccer Ball	$11.00	2	$22.00
30321	Grand Prix Bicycle	$1,669.00	1	$1,669.00

To complete the order, enter the ID of a sales representative and then click the **Commit** button in the top right corner of the Summit Customer Management page. This updates the database with the new order detail and reduced product inventory.

To visualize the customer's order history, click the Orders Dashboard tab. The dashboard displays various graphs using ADF Faces DVT components. The **Shipping Time** graph is an `dvt:barChart` component. This component relies on information that is calculated only when the order ship date is updated; the data to display is not obtained from a database table since it must be determined at runtime. To display the **Shipping Time** data the bar chart relies on an ADF Business Components method call to return the number of days for shipping for each order.

Figure 2-16 shows dashboard information for three orders made by the Acme Sporting Goods company.

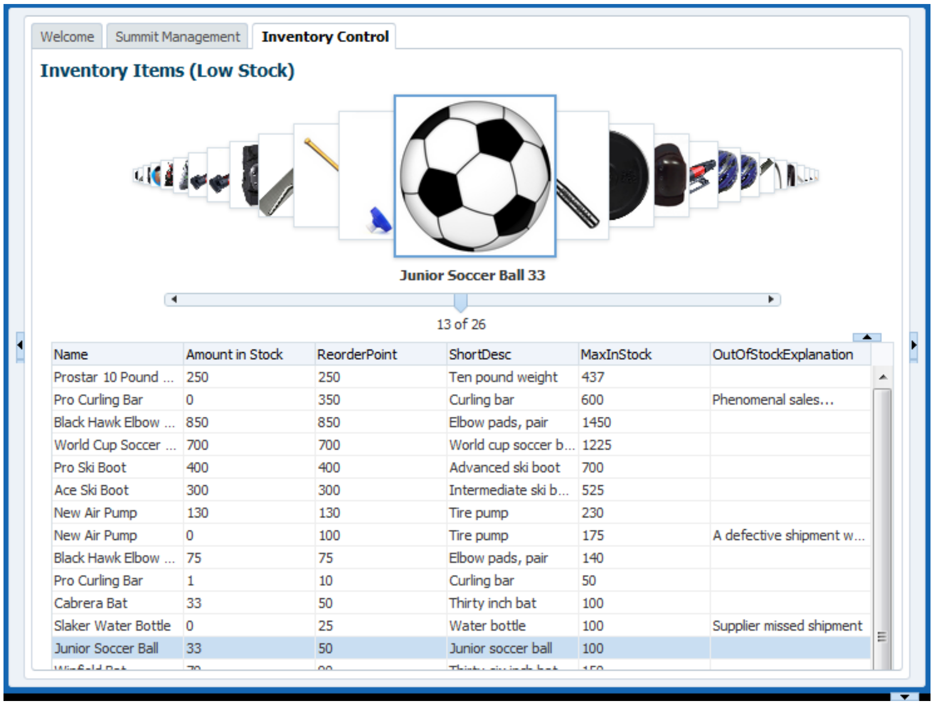
Figure 2-16   Order Page Fragment Displays Dashboard Details



To visualize how orders affect the product inventory, click the Inventory Control tab, next to the Summit Management tab. The Inventory Control page displays only those products that are below the reorder threshold. The low stock is displayed both visually with the `af:carousel` component and in table format with the `af:table` component. Select an item from the low stock carousel and notice that the table updates to make the selected item the current row. The current row in a table is the one that is highlighted. These two components are synchronized through the ADF Model layer with a Java method call that is executed when the carousel selection is made.

Figure 2-17 shows the inventory status of item 13 out of 26. The current row selection in the low stock table reflects the selection displayed in the inventory carousel.

Figure 2-17 Inventory Control Page Fragment Displays Low Stock Items



Inventory Items (Low Stock)

Junior Soccer Ball 33

13 of 26

Name	Amount in Stock	ReorderPoint	ShortDesc	MaxInStock	OutOfStockExplanation
Prostar 10 Pound ...	250	250	Ten pound weight	437	
Pro Curling Bar	0	350	Curling bar	600	Phenomenal sales...
Black Hawk Elbow ...	850	850	Elbow pads, pair	1450	
World Cup Soccer ...	700	700	World cup soccer b...	1225	
Pro Ski Boot	400	400	Advanced ski boot	700	
Ace Ski Boot	300	300	Intermediate ski b...	525	
New Air Pump	130	130	Tire pump	230	
New Air Pump	0	100	Tire pump	175	A defective shipment w...
Black Hawk Elbow ...	75	75	Elbow pads, pair	140	
Pro Curling Bar	1	10	Curling bar	50	
Cabrera Bat	33	50	Thirty inch bat	100	
Slaker Water Bottle	0	25	Water bottle	100	Supplier missed shipment
Junior Soccer Ball	33	50	Junior soccer ball	100	
Winfield Bat	70	90	Thirty six inch bat	150	

Where to Find Implementation Details

Following are the sections of *Developing Fusion Web Applications with Oracle Application Development Framework* that describe how to develop database forms like the ones used to update the database in the Orders and Inventory Control page fragments:

- Triggering data updates with command components

  To support data updating, the Orders page fragment of the Summit Management region uses command components represented as `af:button` components to allow users to execute specific ADF data control operations in the declarative ADF **page definition file**. The built operations of the ADF data control enable you to declaratively handle common form functions such as navigating between records and committing changes to a database. Most operations are available for individual data collections in a data control. The Commit and Rollback operations are available on the whole data control. By dragging and dropping an operation from the Data Controls panel, you are prompted to choose what kind of command component to create, such as a button or a link. For information about triggering built-in operations, see Creating Command Components Using Data Control Operations.

- Triggering backing bean method calls with command components

  To support operations that may require custom logic to handle an interaction with the database, the `InventoryControl` page fragment of the Summit Management region uses the `carouselSpinListener` on the `af:carousel` components to trigger a custom method `handleCarouselSpin()` defined on the backing bean `InventoryControl.java`. When you select an item in the carousel, the action listener triggers the backing bean method which updates the current row for the af:table component iterator binding. For details about executing Java code in backing beans, see How to Create a Databound Carousel Component.

- Keeping track of transient session information and triggering business component method calls

  When you create a data model project that maps attributes to columns in an underlying table, your ADF view objects can include transient attributes that display calculated values (for example, using Java or Groovy expressions) or that are value holders. For example, in the core Summit ADF sample application, the Orders Dashboard tab on the page fragment Orders.jsff displays the value of the TimeToShip attribute calculated by the expression defined on the OrdVO view object. The calculateTimeToShip expression invokes a method on the OrdVORowImpl.java client interface for the view row. For more information about defining transient attributes in the data model project, see Adding Transient and Calculated Attributes to an Entity Object. For more information about implementing custom Java methods in view objects, see How to Generate Client Interfaces for View Objects and View Rows.

Running the Standalone Samples from the SummitADF_Examples Workspace

The standalone sample applications in theSummitADF_Examples application workspace use programmatic test clients to demonstrate concepts related to the ADF Business Components framework. These applications allow you to look into the ADF functionalities that does not appear in the SummitADF workspace.

The Summit standalone sample applications for Oracle ADF include a set of sample packages that allow you to investigate Oracle ADF functionality that does not appear in the SummitADF workspace. Collectively, these sample packages are referred to as Summit ADF standalone sample applications. The standalone applications appear in the Model project of the SummitADF_Examples application workspace, located in the SummitADF_Examples folder where you extracted the Summit ADF standalone sample applications ZIP file.

In general, the standalone applications demonstrate concepts of ADF Business Components and data model projects. References to these standalone applications appear throughout the chapters contained in Building Your Business Services of this developer's guide. As you read sections this guide, you may want to run the corresponding standalone applications to investigate the concepts further. For a brief description of each standalone application and links to the documentation, refer to the table in Overview of the Summit ADF Standalone Sample Applications.

How to Run the Summit ADF Standalone Sample Applications

The approach you use in JDeveloper to run a standalone application depends on the individual package. Some packages are set up to use the interactive testing tool JDeveloper provides for the ADF Business Components data model project (this tool is known as the Oracle ADF Model Tester). Other applications provide Java test clients (with file names like TestClientAaa.java) that use the ADF Business Components API to execute queries and display results. In the case of the Oracle ADF Model Tester, you work entirely within the tool, which essentially provides a convenient user interface for interacting with business components. In the case of the Java clients, the program files output their results and print statements to the JDeveloper Log window.

Running the Standalone Sample Applications From a Java Test Client

When the standalone application has a test client to programmatically exercise the ADF Business Components API, you will use the test client to run the applications.

Before you begin:

It may be helpful to have an understanding of individual Summit ADF standalone sample applications. For more information, see Overview of the Summit ADF Standalone Sample Applications.

You will need to complete these tasks:

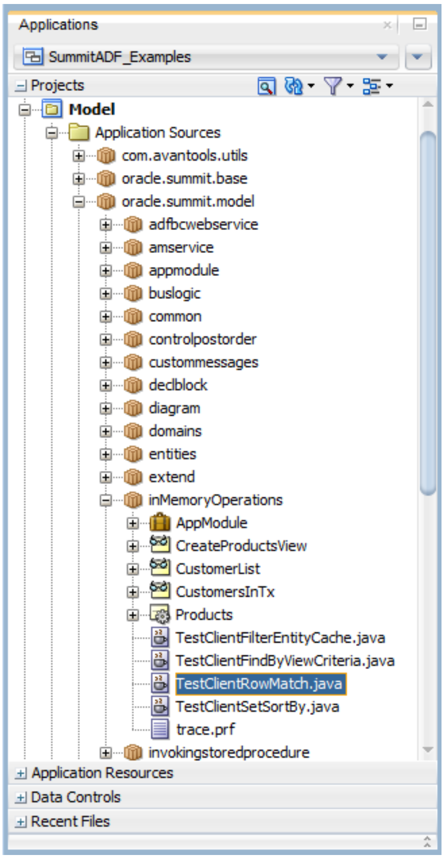
- Install Oracle JDeveloper Studio and meet the installation prerequisites. The standalone applications require an existing Oracle database. For details, see How to Download the Application Resources.
- Download the Summit ADF archive file from the Oracle Technology Network to a local directory and unzip the Summit ADF standalone sample applications. For details, see Setting Up the Summit Sample Applications for Oracle ADF.
- Install the Summit ADF schema. For details, see How to Install the Summit ADF Schema.

To run a standalone application from its provided test client:

1. Open the application in JDeveloper:
   a. In the main menu, choose **File** and then **Open**.
   b. Navigate to the location where you expanded the ZIP file for the **SummitADF_Examples** directory, select the **SummitADF_Examples.jws** application workspace and then click **Open**.
2. Create a database connection for the standalone applications. For details, see How to Create the Database Connection for Summit Sample Applications for Oracle ADF.
3. Run the SQL script in JDeveloper for standalone samples that require it. Some of the standalone applications work with a modified version of the Summit ADF schema. For standalone applications that require schema changes, the application's project contains a SQL script that you must run in JDeveloper. Once you are through running a standalone application, you can use the script to back out the schema changes
4. In the Applications window, expand the project node and locate the test client (`.java`) file node. In some cases, the test client is added to a package located under the **Application Sources** node. In other cases, the **Resources** node contains the test client.

   For example, Figure 2-18 shows the expanded **inMemoryOperations** node with the Java file node **TestClientRowMatch.java** selected.

Figure 2-18 Test Client Selected in Applications Window



5. Right-click the test client and choose **Run**.

   For the names and location of the test clients provided with the standalone applications, see the table in Overview of the Summit ADF Standalone Sample Applications.

   The Create Default Domain dialog appears the first time you run an executable and start a new domain in Integrated WebLogic Server. Use the dialog to define an administrator password for the new domain. Passwords you enter must be eight characters or more and have a numeric character.

6. Examine the JDeveloper Log window for the test client's output.

   Refer to the referenced documentation for details about the expected results.

Running the Standalone Applications Without a Test Client

When the standalone application does not provide a test client to programmatically exercise the ADF Business Components API, you will use the interactive testing tool, known as the Oracle ADF Model Tester.

Before you begin:

It may be helpful to have an understanding of individual Summit ADF standalone sample applications. For more information, see Overview of the Summit ADF Standalone Sample Applications.

It also may be helpful to have an understanding of the Oracle ADF Model Tester. For more information, see Testing View Object Instances Using the Oracle ADF Model Tester.

You will need to complete these tasks:

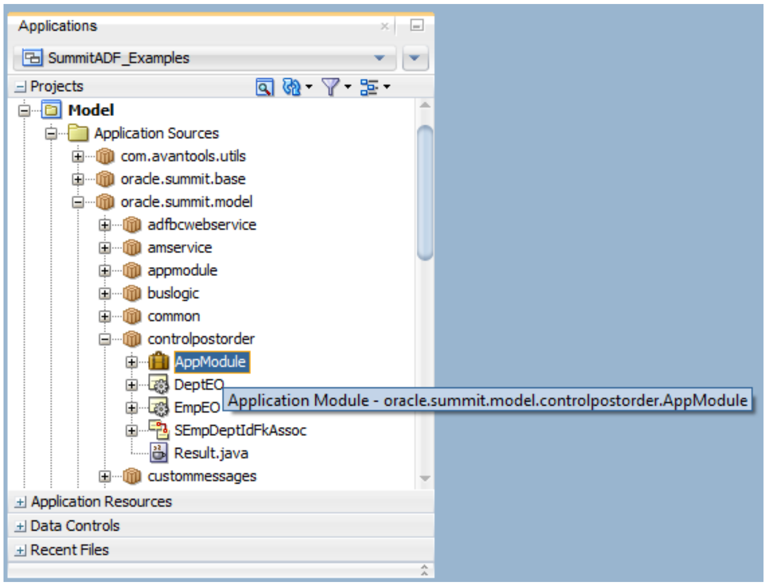
- Install Oracle JDeveloper Studio and meet the installation prerequisites. The standalone applications require an existing Oracle database. For details, see How to Download the Application Resources.
- Download the Summit ADF archive file from the Oracle Technology Network to a local directory and unzip the Summit ADF standalone sample applications. For details, see How to Download the Application Resources.
- Install the Summit ADF schema. For details, see How to Install the Summit ADF Schema.

To run a standalone application in the Oracle ADF Model Tester:

1. Open the application in JDeveloper:
   a. In the main menu, choose **File** and then **Open**.
   b. Navigate to the location where you expanded the ZIP file for the `SummitADF_Examples` directory, select the `SummitADF_Examples.jws` application workspace file and then click **Open**.
2. Create a database connection for the standalone applications. For details, see How to Create the Database Connection for Summit Sample Applications for Oracle ADF.
3. Run the SQL script in JDeveloper for standalone samples that require it. Some of the standalone applications work with a modified version of the Summit ADF schema. For standalone applications that require schema changes, the application's project contains a SQL script that you must run in JDeveloper. Once you are through running a standalone application, you can use the script to back out the schema changes
4. In the Applications window, expand the project node and locate the application module in a package under the **Application Sources** node.

   For example, Figure 2-19 shows the expanded **controlpostorder** project with the application module **AppModule** selected and a tooltip for the node displayed.

Figure 2-19 Application Module Node Selected in Applications Window



5. Right-click the application module node and choose **Run**.

   For the names of the runnable application modules, see the table in Overview of the Summit ADF Standalone Sample Applications.

   The Create Default Domain dialog displays the first time you run an executable and start a new domain in Integrated WebLogic Server. Use the dialog to define an administrator password for the new domain. Passwords you enter must be eight characters or more and have a numeric character.

6. Use the Oracle ADF Model Tester to interact with the view instances of the standalone application.

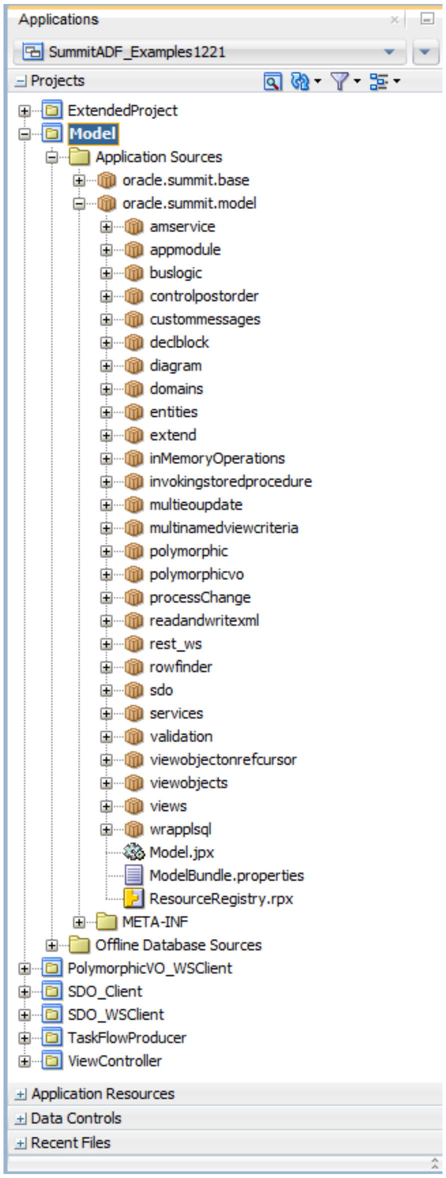
   Refer to the referenced documentation for details about the application. For details about using the tester to interact with the data model, see Testing View Object Instances Using the Oracle ADF Model Tester.

Overview of the Summit ADF Standalone Sample Applications

Some of the standalone applications in the `SummitADF_Examples` application workspace use programmatic test clients to demonstrate concepts related to the ADF Business Components framework. Others demonstrate framework functionality when you run the application in the Oracle ADF Model Tester.

Figure 2-20 shows the Applications window after you open the `SummitADF_Examples` application workspace.

Figure 2-20 Standalone Applications in the SummitADF_Examples Workspace



Applications

SummitADF_Examples1221

Projects

- ExtendedProject
- Model
  - Application Sources
    - oracle.summit.base
    - oracle.summit.model
      - amservice
      - appmodule
      - buslogic
      - controlpostorder
      - custommessages
      - declblock
      - diagram
      - domains
      - entities
      - extend
      - inMemoryOperations
      - invokingstoredprocedure
      - multieoupdate
      - multinamedviewcriteria
      - polymorphic
      - polymorphicvo
      - processChange
      - readandwritexml
      - rest_ws
      - rowfinder
      - sdo
      - services
      - validation
      - viewobjectonrefcursor
      - viewobjects
      - views
      - wrapplsql
      - Model.jpx
      - ModelBundle.properties
      - ResourceRegistry.rpx
    - META-INF
  - Offline Database Sources
- PolymorphicVO_WSClient
- SDO_Client
- SDO_WSClient
- TaskFlowProducer
- ViewController

Application Resources

Data Controls

Recent Files

Note that the test clients for the standalone applications provide a good starting point for understanding how to exercise methods of the ADF Business Components API. They also make good samples for test clients that you may want to create to test business component queries in a data model project. For background on working with test clients, see Testing View Object Instances Programmatically.

Note:

The ADF Business Components API is available when you need to generate custom classes to augment the default runtime behavior of the business components. For background about the ADF Business Components framework, see Getting Started with ADF Business Components.

Table 2-4 describes the standalone applications in the `SummitADF_Examples` application workspace. Examples from these applications appear throughout the chapters contained in Building Your Business Services of this guide.

Table 2-4 Standalone Applications in the SummitADF_Examples Application Workspace

Package Name	Runnable Class or Project Target	Documentation
`amservice`	Run **ServiceAppModule** using the Oracle ADF Model Tester.	For information about the application module service interface, see Publishing Service-Enabled Application Modules.
`appmodule`	Run **TestAM.java** under the **service** node. Run **TestCustomEntity.java** under the **service** node. These test clients exercise custom methods of application module's client interface and print to the JDeveloper Log window to indicate the results.	For details about the test client, see How to Work Programmatically with an Application Module's Client Interface. For details about the methods of the client interface, see the examples in Working Programmatically with Entity Objects and Associations.
`buslogic`	Run **AppModule** using the Oracle ADF Model Tester.	For information about business logic units and groups, see Defining Business Logic Groups.
`controlpostorder`	Run **AppModule** using the Oracle ADF Model Tester.	For information about controlling the posting order resulting from DML operations to save changes to a number of related entity objects, see How to Control Entity Posting Order to Prevent Constraint Violations.
`custommessages`	Run **AppModule** using the Oracle ADF Model Tester.	For information about how to provide an alternative message string for the builtin error codes in a custom message bundle, see Customizing Business Components Error Messages.
`declblock`	Run **AppModule** using the Oracle ADF Model Tester.	For information about how to use custom metadata properties to control insert, update, or delete on a view object, see Declaratively Preventing Insert, Update, and Delete.
`domains`	Run **AppModule** using the Oracle ADF Model Tester.	For information about creating custom data types, see Creating Custom, Validated Data Types Using Domains.
`extend`	Run **TestClient.java**. This client casts the results of the `productByName.first()` method to the `ProductViewRow` interface so that it can make method calls, and prints to the JDeveloper Log window to display the results.	For information about how to extend business components to create a customized versions of the original, see Creating Extended Components Using Inheritance. For information about iterating over a collection using the view row accessor attribute, see Exposing View Row Accessors to Clients.

Table 2-4 (Cont.) Standalone Applications in the SummitADF_Examples Application Workspace

Package Name	Runnable Class or Project Target	Documentation
`inMemoryOperations`	Run the test clients: - **TestClientFilterEntityCache.java** - **TestClientFindByViewCriteria.java** - **TestClientRowMatch.java** - **TestClientSetSortBy.java** These test clients illustrate how to use the in-memory sorting and filtering functionality from the client side using methods on the interfaces in the `oracle.jbo` package, and print to the JDeveloper Log window to display the results.	For information about how to use view objects to perform in-memory searches and sorting to avoid unnecessary trips to the database, see Performing In-Memory Sorting and Filtering of Row Sets.
`invokingstoredprocedure`	Run the **ExampleSQLPackage.sql** script against the `summit_adf` database connection to set up the additional database objects required for the example. Run **TestClient.java**. The test client exercises custom methods of application module's client interface to invoke database stored procedures and functions, and prints to the JDeveloper Log window to indicate the results.	For information about how to code custom Java classes for business components that invoke database stored procedures and functions, see Invoking Stored Procedures and Functions.
`multieoupdate`	Run **AppModule** using the Oracle ADF Model Tester.	For information about creating a view object with multiple updatable entities to support creating new rows, see Creating a View Object with Multiple Updatable Entities.
`multinamedviewcriteria`	Run **MultiNamedViewCriteriaTestClient.java**. The test client exercises custom methods on the view object interface and prints to the JDeveloper Log window to display the results.	For information about how to programmatically filter query results, see Working Programmatically with Multiple Named View Criteria.
`polymorphic`	Run **PolymorphicAppModule** (under the **service** node) using the Oracle ADF Model Tester and display the subtype view instances for US customers (`CountryId=4`). Subtype view usages display the subtype attribute `State`.. This sample demonstrates entity-based polymorphic view objects.	For details about creating an entity object inheritance hierarchy, see Using Inheritance in Your Business Domain Layer and for details about creating view usages for those entity objects, see Working with Polymorphic Entity Usages.

Table 2-4 (Cont.) Standalone Applications in the SummitADF_Examples Application Workspace

Package Name	Runnable Class or Project Target	Documentation
`polymorphicvo`	Run **AppModule** (under the **service** node) using the Oracle ADF Model Tester and display subtype view rows for Sales department employees (`TitleId=2`). Subtype view rows display the subtype attribute `CommissionPct`. This sample demonstrates view row-based polymorphic view objects.	For details about creating a view row subtypes, see Working with Polymorphic View Rows.
`PolymorphicVO_WSClient` (This is a client project of the `polymorphicvo` package.)	First run **AppModuleServiceImpl.java** in the **polymorphicvo** package of the Model project, then run the web service client **AppModuleServiceSoapHttpPortClient** (under the **com.oracle.xmlns.apps.hr.service.types** node). The client file creates a new salesman subtype and prints to the JDeveloper Log. This sample demonstrates how to work with polymorphic collections in SOAP web service clients.	For details about working with subtype view objects in web services, see Accessing Polymorphic Collections in the Consuming Application.
`processChange`	Run **processChangeTestClient.java**. The test client exercises custom methods on the view object interface and prints to the JDeveloper Log window to display the results.	For information about invoking database change notifications, see What You May Need to Know About Programmatically Invoking Database Change Notifications. For details about the test client, see How to Work Programmatically with an Application Module's Client Interface.
`readandwritexml`	Run **TestClientReadXML.java** under the **client** node, then run **TestClientWriteXML.java**.	For details about how to produce XML from queried data, see Reading and Writing XML.
`rest_WS`	This package is reserved for future use.	For details about creating RESTful web services from view instances of an application module, see Creating ADF RESTful Web Services with Application Modules.

Table 2-4 (Cont.) Standalone Applications in the SummitADF_Examples Application Workspace

Package Name	Runnable Class or Project Target	Documentation
`rowfinder`	Run **TestClient.java** to programmatically exercise the row finder `EmpByTitleId` defined on `EmpView`. Row finders may be invoked programmatically within a test client or from the methods of exposed service data object (SDO) operations on a SOAP web service. Alternatively, expand the **serviceinterface** folder of the **EmpService** application module node and run `EmpServiceServiceImpl.java` to test the row finder in the HTTP Analyzer. In the HTTP Analyzer, select the `updateEmpView1` operation and enter `1` for the `TrTitleId` and then enter any First Name to update the Employee record based on a person with the Title Id of "`1`". Row finders may be defined on view objects to support record lookups using any attribute without requiring the view object's key attribute.	For information about defining a row finder and programmatically invoking it, see Working with Row Finders. For information about using row finder operations in SOAP web services, see What You May Need to Know About Row Finders and the ADF Web Service Operations.
`sdo`	This package implements service data objects (SDO) for use with SOAP web services that are based on the `DeptView1` and `EmpView1` view instances of the `HRAppModule` application module. This package exposes a find operation on the `EmpView1` SDO. See `SDO_Client` and `SDO_WSClient` for instruction about running the client projects to test the exposed find operation.	For information about how to create SDO from an application module, see Creating SOAP Web Services with Application Modules.
`SDO_Client`	Run **SDOClientServlet.java** SOAP web service servlet. The test client exercises the `findEmpView1()` method implemented as a SOAP web service view criteria find operation in the services package of the Model project. The servlet `testFindEmps()` method creates a dynamic view criteria passed to `findEmpView1()` that filters employees by a named department and verifies the test result. This is the servlet version of the `SDO_WSClient` project.	For more information about exposing find operations, see How to Expose a Custom Find Method Filtered By a Required Bind Variable.

Table 2-4 (Cont.) Standalone Applications in the SummitADF_Examples Application Workspace

Package Name	Runnable Class or Project Target	Documentation
`SDO_WSClient`	Run **HRAppModuleServiceSoapHttpPortClient.java** SOAP web service HTTP port client. The test client exercises the `findEmpView1()` method implemented as a SOAP web service view criteria find operation in the services package of the Model project. The servlet `testFindEmps()` method creates a dynamic view criteria passed to `findEmpView1()` that filters employees by a named department and verifies the test result. This the web service client version of the `SDO_Client` project.	For more information about exposing find operations in ADF service data objects, see How to Expose a Custom Find Method Filtered By a Required Bind Variable.
`TaskFlowProducer`	Run **RemoteViewer.jsf** page in the **Web Content** folder of this view controller project. The page displays a remote invocable task flow that can be consumed by a consumer application.	For information about creating remote task flows, see How to Configure an Application to Render Remote Regions. Once you deploy this sample application, you can consume the remote invocable task flow, as described in Creating Remote Regions in a Fusion Web Application.
`validation`	Run **AppModule** using the Oracle ADF Model Tester. Click `OrdersView1` and edit the `Data Ordered` field to see an example of attribute level validation using a programmatic validation method.	For information about how to create programmatic validators, see Implementing Validation and Business Rules Programmatically.
`viewobjectonrefcursor`	Run the **CreateRefCursorPackage.sql** script against the `summit_adf` database connection to set up the additional database objects required for the example. Run **AppModule** using the Oracle ADF Model Tester.	For details about how to use PL/SQL to open a cursor to iterate through the results of a query, see How to Create a View Object on a REF CURSOR.
`viewobjects`	Run **TestClient.java**. Programmatically iterates over the customers view instance using methods of the Business Components API `RowIterator` interface and prints to the JDeveloper Log window.	For details about iterating over a collection, see How to Count the Number of Rows in a Row Set. For details about how to create test clients, see Testing View Object Instances Programmatically.
	Run **TestClient2.java**. Programmatically iterates over the customers view instance, accesses the detail collection of orders using a **view link accessor** attribute, and prints to the JDeveloper Log window.	For details about iterating over a detail collection, see How to Access the Detail Collection Using the View Link Accessor. For more details about the test client, see How to Access a Detail Collection Using the View Link Accessor.

Table 2-4 (Cont.) Standalone Applications in the SummitADF_Examples Application Workspace

Package Name	Runnable Class or Project Target	Documentation
	Run **TestClient3.java**. Finds and updates a foreign key value and prints to the JDeveloper Log window.	For details, see How to Find a Row and Update a Foreign Key Value.
	Run **TestClient4.java**. Finds a view instance, creates a new row, inserts it into the row set, and prints to the JDeveloper Log window.	For details, see How to Create a New Row for a View Object Instance.
	Run **TestClient5.java**. Finds a view instance, accesses the row set, iterates over the rows to display the key, and prints to the JDeveloper Log window.	For details, see How to Retrieve the Row Key Identifying a Row.
`wrapplsql`	Run **AppModule** using the Oracle ADF Model Tester.	For details about overriding the default DML processing event for an entity object to invoke methods in a PL/SQL API package that encapsulates insert, update, and delete access to an underlying table, see Basing an Entity Object on a PL/SQL Package API.

Running the Sample Application in the SummitADF_TaskFlows Workspace

The sample Summit application for task flows describes how to use activities in the ADF task flows that you create in your Fusion web application.

The Summit sample application for ADF task flows demonstrates how you can implement security, contextual events, and task flows in your Fusion web application.

How to Run the Summit Sample Application for ADF Task Flows

The easiest way to run the Summit ADF task flow sample application is to expand the ViewController project and run the `index.jsf` page from within this project.

Before you begin:

It may be helpful to have an understanding of the schema that the Summit ADF task flow sample application uses. For more information, see What Happens When You Install the Summit ADF Schema.

You will need to complete these tasks:

- Install Oracle JDeveloper Studio and meet the installation prerequisites. The Summit ADF task flow sample application requires an existing Oracle database. For details, see How to Download the Application Resources.

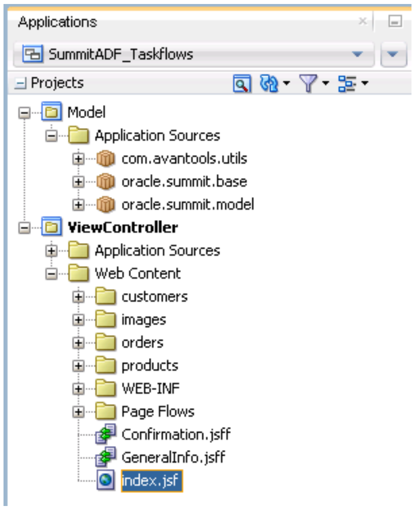
- Download the Summit ADF archive file from the Oracle Technology Network to a local directory and unzip the Summit sample application for ADF task flows. For details, see How to Download the Application Resources.
- Install the Summit ADF schema. For details, see How to Install the Summit ADF Schema.

To run the Summit sample application for ADF task flows:

1. Open the application in JDeveloper:
   a. In the main menu, choose **File** and then **Open**.
   b. Navigate to the location where you expanded the ZIP file for the `SummitADF_Taskflows` directory, select the `SummitADF_Taskflows.jws` application workspace and click **Open**.

      Figure 2-21 shows the Applications window after you open the `SummitADF_Taskflows.jws` application workspace file and expand a number of the nodes. For a description of each of the task flows in the workspace, see Overview of the Summit Sample Application for ADF Task Flows.

      Figure 2-21 Projects in the Summit ADF TaskFlows Application Workspace



2. Create a database connection for the Summit ADF task flow sample application. For details, see How to Create the Database Connection for Summit Sample Applications for Oracle ADF.
3. In the Applications window, expand the **ViewController** and **Web Content** nodes, then right-click `index.jsf` and choose **Run**.

   The Create Default Domain dialog appears the first time you run an application and start a new domain in Integrated WebLogic Server. Use the dialog to define an administrator password for the new domain. Passwords you enter can be eight characters or more and must have a numeric character.
4. Once the `index.jsf` page renders in your browser, you can log in using one of the user accounts listed in Table 2-5 to view the functionality that is configured for an employee or a customer in the application.

Table 2-5 Supplied Users in the Summit ADF Task Flow Sample Application

Username	Password	Application Role	Notes
cmagee	welcome1	Application Employee Role	This is the only user who is preregistered as an employee in the Summit ADF task flow sample application.
214	welcome1	Application Customer Role	This is the only user who is preregistered as a customer in the Summit ADF task flow sample application. The user ID of the customer, Ojibway Retail, serves as the username.

Overview of the Summit Sample Application for ADF Task Flows

The application in the SummitADF_Taskflows workspace demonstrates how you can implement security, contextual events and task flows in your Fusion web application.

The following list shows the filenames of the task flows in the SummitADF_Taskflows workspace. Expand the **Page Flows** node under the ViewController project in the Applications window to view these task flows.

```
|   adfc-config.xml
|
\---flows
    |   customers-task-flow-definition.xml
    |   edit-customer-task-flow-definition.xml
    |   emp-reg-task-flow-definition.xml
    |   product-inventory-task-flow-definition.xml
    |   show-products-task-flow-definition.xml
    |
    \---orders
            create-edit-orders-task-flow-definition.xml
            orders-select-many-items.xml
```

Table 2-6 briefly describes these task flows and provides references to the sections in this guide that use these task flows to demonstrate concepts related to security, contextual events, data rendering and data input.

Apart from these task flows, the SummitADF_Taskflows workspace also contains a jazn-data.xml file where JDeveloper saves all policy store and identity store changes for the application. For more information about this file and its use, see ADF Security Process Overview.

Table 2-6 Task Flows in the Summit ADF Task Flow Sample Application

Task Flow Name	Documentation
adfc-config.xml	Every Fusion web application contains this file. It is the source file for the unbounded task flow that JDeveloper creates by default when you create a Fusion web application. In the Summit ADF task flow sample application, this task flow contains the index view activity that renders the index.jsf page end users see at runtime. This task flow also registers the managed bean (oracle.summit.bean.LoginBean) that handles authentication of users who want to access resources that require authentication. For more information, see How to Create a Login Link Component and Add it to a Public Web Page for Explicit Authentication.
customers-task-flow-definition.xml	This task flow contains a router, a view and a method call activity to navigate an authenticated customer to the row that the customer selects within the Customer.jsff page fragment. For more information, see Working with Task Flow Activities . ADF Security secures this task flow so that only authenticated customers can access the functionality it configures or the web pages it exposes. For more information, see About ADF Security.
edit-customer-task-flow-definition.xml	This task flow renders a number of page fragments in an ADF region that provide controls to authenticated users to edit data. For more information, see Task Flows and ADF Region Use Cases and Examples.
emp-reg-task-flow-definition.xml	The Summit ADF task flow sample application renders this task flow in an ADF dynamic region when an end user clicks the **Register as Employee** link that appears in the index.jsf page at runtime. For more information, see Creating ADF Dynamic Regions. Unlike other task flows (for example, customers-task-flow-definition.xml) users do not have to be authenticated to access this task flow. For this reason, the security policy for this task flow grants view privileges to the builtin anonymous-role application role, as described in What Happens When You Define the Security Policy.
product-inventory-task-flow-definition.xml	This task flow renders the showInventory.jsff page fragment that appears in the Inventory Control tab. This tab is enabled when you log in as an employee. The application uses contextual events to determine what data to display in the showInventory.jsff page fragment in response to the end user selecting data in another ADF region. For more information about this task flow's use of contextual events, see Using Contextual Events.
show-products-task-flow-definition.xml	This task flow renders the showProducts.jsff page fragment that renders an ADF Faces table component containing data about products. At runtime, an end-user's selection of a product broadcasts a contextual event that the product-inventory-task-flow-definition.xml task flow consumes so that it can display the inventory for the product. For more information about this task flow's use of contextual events, see Using Contextual Events.
create-edit-orders-task-flow-definition.xml	This task flow renders the Orders.jsff page fragment where an end user can add a new product to an order. It also calls another task flow (orders-select-many-items), as described in How to Call a Bounded Task Flow Using a Task Flow Call Activity.
orders-select-many-items.xml	This task flow renders a modal dialog, as described in Running a Bounded Task Flow in a Modal Dialog.

Running the Standalone Samples in the SummitADF_DVT Workspace

The standalone Summit sample application for ADF DVT components provide extensive graphical and tabular capabilities for visually displaying and analyzing business data.

The Summit sample application for ADF DVT components includes a set of standalone sample pages that demonstrate creation and configuration of ADF DVT components using ADF Business Components and data binding.

References to these standalone pages appear throughout the following chapters:

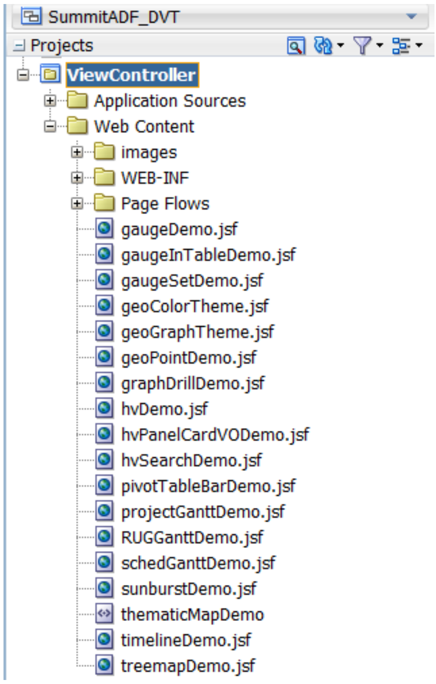
- Creating Databound Chart and Gauge Components
- Creating Databound Pivot Table and Pivot Filter Bar Components
- Creating Databound Geographic and Thematic Map Components
- Creating Databound Gantt Chart and Timeline Components
- Creating Databound Hierarchy Viewer, Treemap, and Sunburst Components

Overview of the Summit ADF DVT Standalone Samples

Each of the standalone pages in the Summit ADF DVT sample application uses the Summit ADF schema to demonstrate creation and configuration of a databound Oracle ADF Data Visualization component.

Figure 2-22 shows the Applications window after you open the `SummitADF_DVT` application workspace.

Figure 2-22 Runnable Pages in the Summit ADF DVT Application Workspace



SummitADF_DVT
Projects
ViewController
Application Sources
Web Content
images
WEB-INF
Page Flows
gaugeDemo.jsf
gaugeInTableDemo.jsf
gaugeSetDemo.jsf
geoColorTheme.jsf
geoGraphTheme.jsf
geoPointDemo.jsf
graphDrillDemo.jsf
hvDemo.jsf
hvPanelCardVODemo.jsf
hvSearchDemo.jsf
pivotTableBarDemo.jsf
projectGanttDemo.jsf
RUGGanttDemo.jsf
schedGanttDemo.jsf
sunburstDemo.jsf
thematicMapDemo
timelineDemo.jsf
treemapDemo.jsf

Table 2-7 describes the standalone pages in the `SummitADF_DVT` application workspace. Each page is runnable on its own.

Table 2-7 Standalone Pages in the Summit ADF DVT Application Workspace

Page Name	Page Description	Documentation
`bubbleChartDemo.jsf`	Demonstrates creation of a DVT bubble chart from a data control	How to Create Databound Bubble and Scatter Charts
`comboChartDemo,jsf`	Demonstrates creation of a DVT combination char from a data control	How to Create an Area, Bar, Combination, Horizontal Bar, or Line Chart Using Data Controls
`gaugeDemo.jsf`	Demonstrates creation of a DVT dial gauge from a data control	Creating Databound Gauges
`gaugeInTableDemo.jsf`	Demonstrates creation of a databound ADF table that stamps a status meter gauge in one of its columns	Including Gauges in Databound ADF Tables
`hvDemo.jsf`	Demonstrates creation of a databound hierarchy viewer rendering an organizational chart of Summit employees	How to Create a Hierarchy Viewer Using ADF Data Controls
`hvPanelCardVODemo.jsf`	Demonstrates creation of a databound hierarchy viewer using an alternate view object for one of its panel cards	How to Configure an Alternate View Object for a Databound Panel Card
`hvSearchDemo.jsf`	Demonstrates hierarchy viewer search configuration	How to Create a Databound Search in a Hierarchy Viewer
`pivotTableBarDemo.jsf`	Demonstrates creation of a pivot table and pivot filter bar using data controls	How to Create a Pivot Table Using ADF Data Controls
`projectGanttDemo.jsf`	Demonstrates creation of a project Gantt chart using data controls	How to Create a Databound Project Gantt Chart
`RUGGanttDemo.jsf`	Demonstrates creation of a resource utilization Gantt chart using data controls	How to Create a Databound Resource Utilization Gantt Chart
`scatterChartDemo.jsf`	Demonstrates creation of a DVT scatter chart from a data control	How to Create Databound Bubble and Scatter Charts
`schedGanttDemo.jsf`	Demonstrates creation of a scheduling Gantt chart using data controls	How to Create a Databound Scheduling Gantt Chart
`sunburstDemo.jsf`	Demonstrates creation of a sunburst using data controls	How to Create Treemaps and Sunbursts Using ADF Data Controls
`thematicMapDemo.jsf`	Demonstrates creation of a thematic map using data controls	How to Create a Thematic Map Using ADF Data Controls
`timelineDemo.jsf`	Demonstrates creation of a timeline component using data controls	How to Create a Timeline Using ADF Data Controls
`treemapDemo.jsf`	Demonstrates creation of a treemap using data controls	How to Create Treemaps and Sunbursts Using ADF Data Controls

How to Run the Summit ADF DVT Sample Application Pages

To run the Summit ADF DVT sample application JSF pages, open the application, expand the View Controller project and run any of the *sample*`.jsf` files.

Before you begin:

It may be helpful to have an understanding of the schema that the Summit ADF DVT sample application uses. For more information, see What Happens When You Install the Summit ADF Schema.

You will need to complete these tasks:

- Install Oracle JDeveloper Studio and meet the installation prerequisites. The Oracle ADF DVT sample application requires an existing Oracle database. For details, see How to Download the Application Resources.
- Download the Summit ADF archive file from the Oracle Technology Network to a local directory and unzip the Summit sample application for ADF DVT. For details, see How to Download the Application Resources.
- Install the Summit ADF schema. For details, see How to Install the Summit ADF Schema.

To run the Summit ADF DVT sample application pages:

1. Open the application in JDeveloper:
   a. In the main menu, choose **File** and then **Open**.
   b. Navigate to the location where you expanded the ZIP file for the `SummitADF_DVT` directory, select the `SummitADF_DVT.jws` application workspace and click **Open**.
2. Create a database connection for the Summit ADF DVT sample application. For details, see How to Create the Database Connection for Summit Sample Applications for Oracle ADF.
3. In the Applications window, expand the **ViewController** and **Web Content** nodes.

   Figure 2-22 shows the Applications window after you open the `SummitADF_DVT.jws` application workspace file for the Summit ADF DVT sample application and expand the **View Controller** and **Web Content** nodes. For a description of each page in the application, see Table 2-7.
4. To run a page, right-click the desired .jsf page and choose **Run**.

   The Create Default Domain dialog appears the first time you run an application and start a new domain in Integrated WebLogic Server. Use the dialog to define an administrator password for the new domain. Passwords you enter can be eight characters or more and must have a numeric character.

Part II
Building Your Business Services

This part describes the tasks developers can perform when creating the business components of the Fusion web application.

Part II contains the following chapters:

3

Getting Started with ADF Business Components

This chapter describes getting started with the ADF Business Components in Oracle ADF applications. It describes key features such as the design-time wizards, overview editors, and ADF Model Tester. It also provides an overview of the process that you follow when creating business components in the data model project.

This chapter includes the following sections:

About ADF Business Components

ADF Business Components is a framework that supports creating the Business Service layer of a Fusion web application in JDeveloper with a minimum of coding. In the Fusion web application, the **business service layer** handles connecting to the database, retrieving data, locking database records, and managing transactions. Consider the following three simple examples of business service layer functionality:

- New data appears in relevant displays without requerying

  A person logs into a customer management application and displays a customer and their existing orders. Then if the person creates a new order, when he views the inventory for the ordered product, the amount of product in stock is updated without the need to requery the database.

- Changes caused by business domain logic automatically reflected

  A back office application causes an update to the order status. Business logic encapsulated in the `Orders` **entity object** in the business domain layer contains a simple rule that updates the last update date whenever the order status attribute is changed. The user interface updates to automatically reflect the last update date that was changed by the logic in the business domain layer.

- Invocation of a business service method by the ADF Model layer binding requeries data and sets current rows

  In a tree display, the end user clicks on a specific node in a tree. This action declaratively invokes a business service method by the ADF tree binding on your **application module** that requeries master-detail information and sets the current rows to an appropriate row in the row set. The display updates to reflect the new master-detail data and current row displayed.

Developers save time building business services using **ADF Business Components** since the JDeveloper design time makes typical development tasks entirely declarative. In particular, JDeveloper supports declarative development with ADF Business Components to:

- Author and test business logic in components which automatically integrate with databases
- Reuse business logic through multiple SQL-based views of data, supporting different application tasks
- Access and update the views from browser, desktop, mobile, and web service clients
- Customize application functionality in layers without requiring modification of the delivered application

ADF Business Components supports the business service layer through the following set of cooperating business components:

- Entity object

  An **entity object** represents a row in a database table and simplifies modifying its data by handling all data manipulation language (DML) operations for you. It can encapsulate business logic for the row to ensure that your business rules are consistently enforced. You associate an entity object with others to reflect relationships in the underlying database schema to create a layer of business domain objects to reuse in multiple applications.

- View object

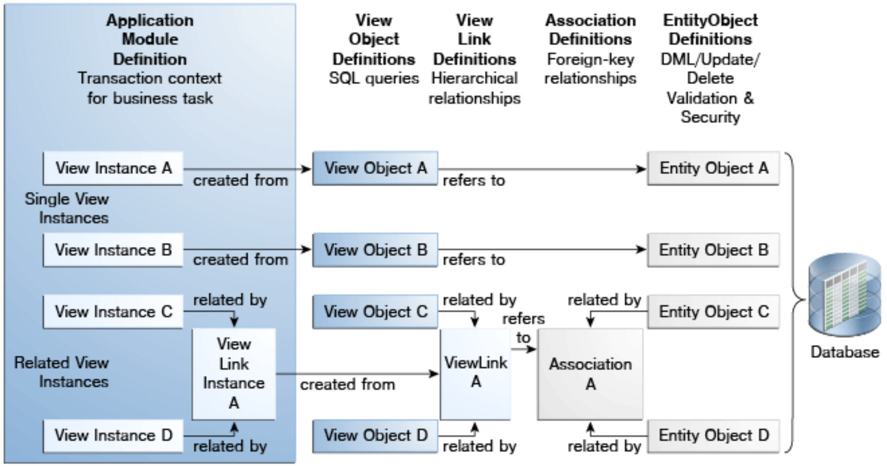
  A **view object** represents a SQL query. You use the full power of the familiar SQL language to join, filter, sort, and aggregate data into exactly the shape required by the end-user task. This includes the ability to link a view object with others to create master-detail hierarchies of any complexity. When end users modify data in the user interface, your view objects collaborate with entity objects to consistently validate and save the changes.

- Application module

  An **application module** is the transactional component that UI clients use to work with application data. It defines an updatable data model and top-level procedures and functions (called **service methods**) related to a logical unit of work related to an end-user task.

Figure 3-1 shows the overall components of the ADF Business Components and their relationships.

Figure 3-1 Overview of ADF Business Components



ADF Business Components Use Cases and Examples

ADF Business Components provides a foundation of Java classes that allow the business service layer of the Fusion web application to leverage the functionality depicted in the following examples:

Simplify Data Access

- Design a data model for client displays, including only necessary data
- Include master-detail hierarchies of any complexity as part of the data model
- Implement end-user Query-by-Example data filtering without code
- Automatically coordinate data model changes with the **business services layer**
- Automatically validate and save any changes to the database

Enforce Business Domain Validation and Business Logic

- Declaratively enforce required fields, primary key uniqueness, data precision-scale, and foreign key references
- Easily capture and enforce both simple and complex business rules, programmatically or declaratively, with multilevel validation support
- Navigate relationships between business domain objects and enforce constraints related to compound components

Support User Interfaces with Multipage Units of Work

- Automatically reflect changes made by business service application logic in the user interface
- Retrieve reference information from related tables, and automatically maintain the information when the end user changes foreign-key values
- Simplify multistep web-based business transactions with automatic web-tier state management

- Handle images, video, sound, and documents without having to use code
- Synchronize pending data changes across multiple views of data
- Consistently apply prompts, tooltips, format masks, and error messages in any application
- Define custom metadata for any business components to support metadata-driven user interface or application functionality
- Add dynamic attributes at runtime to simplify per-row state management

Implement a Service-Oriented Architecture

- Support highly functional web service interfaces for business integration without writing code
- Enforce best-practice interface-based programming style
- Simplify application security with automatic JAAS integration and audit maintenance
- "Write once, run anywhere": use the same business service as plain Java class or web service

Support Application Customization

- Extend component functionality after delivery without modifying source code
- Globally substitute delivered components with extended ones without modifying the application
- Deliver application upgrades without losing or having to reapply downstream customizations manually

Additional Functionality for ADF Business Components

You may find it helpful to understand related **Oracle ADF** features before you start working with ADF Business Components. Following are links to other functionality that may be of interest.

- For details about the Java EE design patterns that ADF Business Components implements, see ADF Business Components Java EE Design Pattern Catalog.
- For details about method calls of the ADF Business Components interfaces and classes, see Most Commonly Used ADF Business Components Methods.
- For details about common Oracle Forms tasks are implemented in Oracle ADF, see Performing Common Oracle Forms Tasks in Oracle ADF.
- For API documentation related to the `oracle.jbo` package, see the following Javadoc reference document:
  - *Java API Reference for Oracle ADF Model*

Creating the ADF Business Components Data Model Project

JDeveloper allows you to create the data model project using ADF view objects and encapsulate them in an ADF Application Module.
JDeveloper includes comprehensive design time support for ADF Business Components. Collectively, these facilities let you create, edit, diagram, test, and refactor the business components.

How to Create a Data Model Project for ADF Business Components

JDeveloper provides an application template that aids in creating databound web applications with Oracle ADF. When you use the **ADF Fusion Web Application** template, your application will consist of one project for the data model (with ADF Business Components) and another project for the view and controller components (**ADF Faces** and **ADF task flows**).

Before you begin:

It may be helpful to have an understanding of the Fusion web application. See Introduction to Building Fusion Web Applications with Oracle ADF.

You may also find it helpful to understand ADF Business Components preferences that can be specified globally across data model projects. For more information, see How to Customize ADF Business Components Preferences.

You may also find it helpful to understand more about the tools and features that JDeveloper provides to create and manage applications. See Getting Started with Developing Applications with Oracle JDeveloperin *Developing Applications with Oracle JDeveloper*.

To create a data model project with the ADF Fusion Web Application template:

1. In the main menu, choose **File** and then **Application > New**.
2. In the New Gallery, in the **Items** list, double-click **ADF Fusion Web Application**.
3. In the Create ADF Fusion Web Application wizard, enter application details like the name and directory. For help with the wizard, press F1.

   The Project 1 Name page of the wizard shows the list of features needed to complete the data model project. The project feature **ADF Business Components** should appear in the list.
4. Click **Finish**.

How to Create a Data Model Project for ADF REST Web Applications

JDeveloper provides an application template that aids in creating databound web applications with Oracle ADF. When you use the **ADF REST Web Application** template, your application will consist of one project for the web components (REST Web Project) and another project for the data model (ADF Business Components).

Before you begin:

It may be helpful to have an understanding of the REST web application. For more information, see About RESTful Web Services and ADF Business Components.

You may also find it helpful to understand ADF Business Components preferences that can be specified globally across data model projects. For more information, see How to Customize ADF Business Components Preferences.

You may also find it helpful to understand more about the tools and features that JDeveloper provides to create and manage applications. For more information, see the Getting Started with Developing Applications with Oracle JDeveloper chapter in *Developing Applications with Oracle JDeveloper*.

To create a data model project with the ADF REST Web Application template:

1. In the main menu, choose **File** and then **Application > New**.
2. In the New Gallery, in the **Items** list, double-click **ADF REST Web Application**.
3. In the Create ADF REST Web Application wizard, enter application details like the name and directory. For help with the wizard, press F1.
4. The Version page of the wizard shows the new version release name and internal name created during the application creation process. Click **Next**.
5. The Project Name page of the wizard shows the list of features needed to complete the data model project. The project feature **ADF Business Components** should appear in the list. Click **Next**.
6. In the Project Java Settings page, enter the package name, the directory of the Java source code in your project, and output directory where output class files will be placed, and click **Finish**.

   > **Note:**
   >
   > REST Web Service project is named by default as `RESTWebService` and contains the `web.xml` file. You cannot edit this project name during the new application setup.

7. You may proceed to create ADF REST resources using the Create Business Components from Tables wizard. For details, see How to Create ADF REST Resources Using the Create Business Components from Tables Wizard.

How to Add an ADF Business Components Data Model Project to an Existing Application

Applications that you create without using a predefined technology template, may be constructed by adding project folders from the New Gallery. When you add a project for ADF Business Components, you will select **ADF Model Project** from the list in the New Gallery. This creates a project that defines a data model for an ADF web application that relies on ADF Business Components.

Before you begin:

It may be helpful to have an understanding of the Fusion web application. For more information, see Introduction to Building Fusion Web Applications with Oracle ADF.

You may also find it helpful to understand more about the tools and features that JDeveloper provides to create and manage applications. See Getting Started with Developing Applications with Oracle JDeveloper in *Developing Applications with Oracle JDeveloper*.

To add a data model project to your application:

1. In the Applications window, select any existing project and in the main menu, choose **File** and then **New > Project**.
2. In the New Gallery, in the **Items** list, double-click **ADF Model Project**.
3. In the Create ADF Model Project wizard, enter project details like the name and directory. For help with the wizard, press F1.

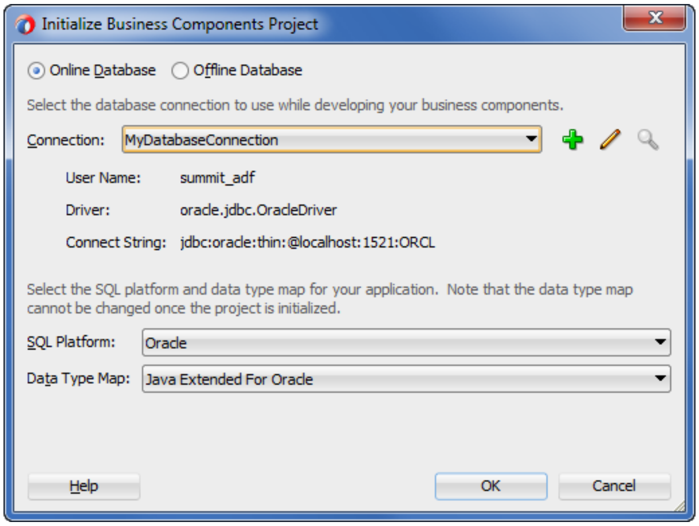
The Project Name page of the wizard shows the list of features needed to complete the model project. The project feature **ADF Business Components** should appear in the list.

4. Click **Finish**.

How to Initialize the Data Model Project With a Database Connection

The first time you create a business component in your ADF Business Components model project, you'll see the Initialize Business Components Project dialog shown in Figure 3-2. You use this dialog to select a design time application resource connection to use while working on your business components in this data model project or to create a new application resource connection by copying an existing IDE-level connection. You can specify whether the database that JDeveloper will use to create business components will be online or offline. The dialog default specifies an online database, for more details about working with an offline database, see How to Change the Data Model Project to Work With Offline Database Objects.

Figure 3-2 Initialize Business Components Project Dialog



● Online Database ○ Offline Database

Select the database connection to use while developing your business components.

Connection: MyDatabaseConnection

User Name: summit_adf

Driver: oracle.jdbc.OracleDriver

Connect String: jdbc:oracle:thin:@localhost:1521:ORCL

Select the SQL platform and data type map for your application. Note that the data type map cannot be changed once the project is initialized.

SQL Platform: Oracle

Data Type Map: Java Extended For Oracle

Since this dialog appears before you create your first business component, you also use it to globally control the SQL platform that the view objects will use to formulate SQL statements. SQL platforms that you can choose include:

- Oracle SQL platform for an Oracle database connection (the default)
- OLite for the Oracle Lite database
- SQLServer for a Microsoft SQLServer database
- DB2 for an IBM DB2 database
- SQL92 for any other supported SQL92- compliant database

> **Note:**
>
> If you plan to have your application run against both Oracle and non-Oracle databases, you should select the **SQL92** SQL platform when you begin building your application, not later. While this sacrifices some of the Oracle-specific optimizations that are inherent in using the Oracle SQL platform, it makes the application portable to both Oracle and non-Oracle databases. Additionally, it is important to understand that support for running ADF applications with multiple SQL platforms chosen at design time is not supported by the ADF runtime. Therefore, you must ensure that ADF applications that you plan to deploy to the same managed server all use a single value for their chosen SQL platform. In the scenario where you will run against both Oracle and non-Oracle databases, you must therefore ensure that all applications have SQL92 chosen as their SQL platform. If this requirement is not observed, SQL validation errors may result at runtime.

Additionally, the dialog lets you determine which set of data types that you want the data model project to use. JDeveloper uses the data type selection to define the data types of attributes when you create entity object and view objects in the data model project. It is therefore important that you make the appropriate selection before you save the settings in the Initialize Business Components Project dialog. The dialog provides these options:

- **Java Extended for Oracle** type map is selected by default if JDeveloper detects you are using an Oracle database driver. The **Java Extended for Oracle** type map uses standard Java types and the optimized types in the `oracle.jbo.domain` package for common data types.

  > **Tip:**
  >
  > New Fusion web applications should use the default **Java Extended for Oracle** type.

- **Java** type map is provided to support applications that will run on a non-Oracle database and that you create using SQL92-compliance. In this case, you should set the data type map to **Java** to globally use only the basic Java data types.
- **Oracle Domains** type map is provided for backward compatibility and for ADF applications that do not use ADF Faces as the view layer technology, as explained in What You May Need to Know About Displaying Numeric Values. Please note that when you migrate an application developed with JDeveloper version 11.1.1.4.0 or earlier, your application will continue to use the **Oracle Domains** type map and will not change to the current default type map **Java Extended for Oracle**

Once you save project selections in the Initialize Business Components Project dialog, the project is considered initialized. You will not be able to change the data type map of the initialized project and it is not recommended to change the SQL platform after you have created business components.

Before you begin:

It may be helpful to have an understanding about database connections as resources in JDeveloper. See Connecting to and Working with Databases in *Developing Applications with Oracle JDeveloper*.

You will need to complete these tasks:

- Create the data model project for ADF Business Components, as described in How to Create a Data Model Project for ADF Business Components.
- Obtain the connection credentials for the database that you will use as a data source for the business components you will create.

To initialize the data model project:

1. In the Application window, right-click the model project in which you want to create business components and choose **New > Business Components from Tables**.

   You can select any other ADF Business Components option, but the option to create business components from tables is particularly useful for generating the first pass of business components that you anticipate needing. If you do not see ADF Business Components options in the context menu, make sure that a model project created for ADF Business Components appears selected in the Applications window.

2. In the Initialize Business Components Project dialog, leave **Online Database** selected and either choose an existing database resource from the dropdown list or click the **Create a New Database Connection** icon.

   If you are creating a database connection, enter the connection information into the Create Database Connection dialog. For help with the dialog, press F1. You can begin working with offline database objects at any time, as described in How to Change the Data Model Project to Work With Offline Database Objects.

3. Select the SQL platform that your business components will use.

   You can select the SQL platform in this dialog only when you first initialize your data model project. After you initialize the project, you can override the SQL platform in the Business Components page of the overview editor for the `adf-config.xml` file, but you must do this before you add business components to the project. You can locate the file in the Application Resources panel by expanding the **Descriptors** and **ADF META-INF** nodes. Specifying the database type in the `adf-config.xml` file supports generating SQL statements during runtime that can require the actual database type of the deployed Fusion web application.

4. Select the data type map that your business components will use. Note that you cannot change the data type map after you have initialized the data model project for ADF Business Components.

   For an application running against an Oracle database, you can usually accept the default **Java Extended for Oracle** which uses standard Java types wherever possible and provides custom Oracle domain types for common data types like numbers and dates. Other supported data type maps include **Oracle Domains** (provided for backward compatibility) and **Java** (select for use with SQL92-compliant applications that will run on a non-Oracle database).

5. Click **OK**.

How to Change the Data Model Project to Work With Offline Database Objects

ADF Business Components supports using offline database as its database object provider. After you have initialized the data model project to specify a database, you can switch to working with the database in offline mode. Working with an offline database to create business components in the model project lets you work in a disconnected fashion and pick up database changes when are ready. This also keeps your project independent of schema changes that might force you to make changes to the business objects.

When you create or select an existing offline database, you still need an online connection; this will be used for testing (using the Oracle ADF Model Tester) and deployment. During design time, the offline database will be used. You can also update an offline database to synchronize with the source database. A database object compare feature lets you obtain a visual mapping that you can use to accept or reject each difference.

Before you begin:

It may be helpful to have an understanding about offline databases in JDeveloper. For more information, see Connecting to and Working with Databases in *Developing Applications with Oracle JDeveloper*.

You will need to complete these tasks:

- Create the data model project for ADF Business Components, as described in How to Create a Data Model Project for ADF Business Components.
- Create the database connection for use when testing business components, as described in How to Initialize the Data Model Project With a Database Connection.

To change the data model project to work with an offline database:

1. In the Application window, right-click the data model project in which you want to create the offline database and choose **Project Properties**.
2. In the Project Properties dialog, select **ADF Business Components**.
3. On the ADF Business Components page, select **Offline Database** and click the **Create a New Offline Database** icon.
4. In the Create Offline Database dialog, enter the database and schema names that you want the data model project to display, select other options to complete the dialog, and then click **OK**.

   The data model project is updated with a new Offline Database Sources folder and contains an empty folder with the name of the offline database.
5. In the **Connection** list, choose the online database that you will connect to when testing business components in the data model project.

   The online database and offline database that you select must have the same schema to ensure database object names and enable testing. For more information about testing business components, see How to Use the Oracle ADF Model Tester.
6. Click **OK**.

To create offline database definitions in the data model project:

1. In the Application window, right-click the folder in which you want to create the offline database and choose **New > Offline Database Objects from Source Database**.
2. In the Reverse Engineer Database Objects wizard, on the Database page, verify that your online database connection is displayed as the **Source Database** and click **Next**.
3. On the Select Objects page, click **Query** and select one or more desired methods from the **Available** list and click the **Add** button to shuttle them into the **Selected** list.
4. Click **Finish**.

What You May Need to Know About Application Server or Database Independence

Applications built using ADF Business Components can be deployed to and run on any application server that Oracle certifies for ADF applications (such as Oracle WebLogic Server or GlassFish). Because business components are implemented using plain Java classes and XML files, you can use them in any runtime environment where a Java Virtual Machine is present. This means that services built using ADF Business Components are easy to use both inside a Java EE server — known as the "container" of your application at runtime — and outside. For more information about deploying ADF applications and the supported application servers, see Deploying Fusion Web Applications.

Customers routinely use application modules in such diverse configurations as command-line batch programs, web services, custom servlets, and JSP pages.

You can also build applications that work with non-Oracle databases, as described in How to Initialize the Data Model Project With a Database Connection. However, applications that target Oracle databases will find numerous optimizations built into ADF Business Components.

> **Note:**
>
> Support for using multiple SQL flavors across ADF applications running on same server is not implemented nor supported by ADF. ADF application developers must therefore ensure that ADF applications they plan to run in the same managed server all use a single value for their SQL flavor. At design time, the SQL flavor can be set either at the project level using the Initialize Business Components dialog or globally in the `adf-config.xml` file. Regardless of the approach used to set the SQLBuilder property, in practice, the value must be the same for all ADF applications running on the same managed server. Failure to implement this requirement can result in runtime conflicts in the binding style that ADF will use to process query statements and this in turn will cause SQL validation errors for the applications. Additionally, it is important to note in the scenario where an ADF application needs to run against both Oracle and non-Oracle databases, all applications on the same managed server must have SQL92 chosen as their SQL flavor.

What You May Need to Know About ADF Business Components Data Types

ADF Business Components is compatible with the built-in data types the Java language provides, including strings, dates, and numeric data. In addition to these Java data types, the `oracle.jbo.domain` packages of ADF Business Components provides types that are optimized for the Oracle database. Table 3-2 shows the ADF Business Components-specific types that you may use in addition to the built-in types provided by the Java language.

Table 3-1 Basic Data Types in the oracle.jbo.domain Packages

Data Type	Package	Represents
`DBSequence`	`oracle.jbo.domain`	Sequential integer assigned by a database trigger
`RowID`	`oracle.jbo.domain`	Oracle database ROWID
`BFileDomain`	`oracle.jbo.domain`	Binary File (BFILE) object
`BlobDomain`	`oracle.jbo.domain`	Binary Large Object (BLOB)
`ClobDomain`	`oracle.jbo.domain`	Character Large Object (CLOB)
`Struct`	`oracle.jbo.domain`	User-defined object type
`Array`	`oracle.jbo.domain`	User-defined collection type (e.g. VARRAY)

For backward compatibility, ADF Business Components still provides the option to select a set of data types in the `oracle.jbo.domain` packages that were available before Java types became the standard. Table 3-2 shows the types that you may use when backward compatibility is desired.

Table 3-2 Backward Compatible Data Types in the oracle.jbo.domain Packages

Data Type	Package	Represents
`Number` (not used by default)	`oracle.jbo.domain`	Any numerical data. If you receive compiler or runtime errors related to "Number is an abstract class" it means you are using `java.lang.Number` instead of `oracle.jbo.domain.Number`. Adding the following line at the top of your class, after the `package` line, prevents these kinds of errors: `import oracle.jbo.domain.Number;`
`Date`	`oracle.jbo.domain`	Date with optional time
`Timestamp`	`oracle.jbo.domain`	Timestamp value
`TimestampTZ`	`oracle.jbo.domain`	Timestamp value with time zone information

Table 3-2 (Cont.) Backward Compatible Data Types in the oracle.jbo.domain Packages

Data Type	Package	Represents
`TimestampLTZ`	`oracle.jbo.domain`	Timestamp value with local time zone information retrieved from JavaVM or from the ADF Context when configured in the application's `adf-config.xml` with an EL expression: `<user-time-zone-config xmlns= "http://xmlns.oracle.com/adf/usertimezone/config"> <user-timezone expression= "EL exp" /> </user-time-zone-config>` The EL expression will be evaluated to determine the time zone of the current end user; otherwise, the value defaults to the time zone of the JavaVM.

What You May Need to Know About Displaying Numeric Values

The **Java Extended for Oracle** type map and the **Oracle Domains** type map handle numeric data differently. When you create a new application the default type map **Java Extended for Oracle** maps numeric data to the `java.math.BigDecimal` class, which inherits from `java.math.Number`. The `java.math.BigDecimal` default matches the way the Fusion web application view layer, consisting of ADF Faces components, preserves alignment of numeric data (such as numeric values displayed by ADF Faces input fields in a web page). Whereas the **Oracle Domains** type map, which maps numeric data to the `oracle.jbo.domain.Number` class, may not display the data with the alignment expected by certain ADF Faces components. Aside from this alignment issue, the **Oracle Domains** type map remains a valid choice and applications without ADF Faces components will function without issue.

How to Customize Model Project Properties for ADF Business Components

Before you begin creating business components, it is a good idea to familiarize yourself with the many project properties that are available to configure ADF Business Components. Selections that you make in the Project Properties dialog will apply to the data model project you are editing. You can specify a variety of properties including suffix naming conventions for generated business components and subpackage names for the data model project.

Before you begin:

It may be helpful to have an understanding about managing projects in JDeveloper. See Getting Started with Developing Applications with Oracle JDeveloper in *Developing Applications with Oracle JDeveloper*.

You may also find it helpful to understand preferences that can be specified globally across data model projects. See How to Customize ADF Business Components Preferences.

You will need to complete this task:

Create the data model project for ADF Business Components, as described in How to Create a Data Model Project for ADF Business Components.

To customize data model project properties:

1. In the Application window, right-click the project in which you want to create the shared application module and choose **Project Properties**.
2. In the Project Properties dialog, expand **ADF Business Components** and select the desired node in the tree. For help with any of the pages of the dialog, press F1.

How to Customize ADF Business Components Preferences

Many design time and runtime options that you can configure for ADF Business Components are specified at the level of JDeveloper preferences. It is a good idea to familiarize yourself with these preferences for ADF Business Components. Selections that you make in the Preferences dialog will apply to each data model project you create.

Certain preferences that you specify globally may be overridden for individual data model projects, as described in How to Customize Model Project Properties for ADF Business Components.

Before you begin:

You may also find it helpful to understand properties that can be specified for individual data model projects. For more information, see How to Customize Model Project Properties for ADF Business Components.

You will need to complete this task:

> Create the data model project for ADF Business Components, as described in How to Create a Data Model Project for ADF Business Components.

To customize ADF Business Components preferences:

1. In JDeveloper, choose **Tools > Preferences**.
2. In the Preferences dialog, expand **ADF Business Components** and select the desired node in the tree. For help with any of the pages of the dialog, press F1.

Creating and Editing Business Components

Use JDeveloper’s design time to work with the ADF Data Model to build databound pages and also create and edit ADF Business Components using dedicated wizards and overview editors.
JDeveloper includes comprehensive design time support to create and customize the business components of the data model project. These facilities let you create the components using dedicated wizards, modify their properties with overview editors, and view their definition files in source editors.

How to Create New Components Using Wizards

In the New Gallery, in the **ADF Business Components** category, JDeveloper offers a wizard to create each kind of business component. Each wizard allows you to specify the component name for the new component and to select the package into which you'd like to organize the component. If the package does not yet exist, the new component becomes the first component in that new package.

Each wizard presents a series of pages that capture the necessary information to create the component type. When you click **Finish**, JDeveloper creates the new component by saving its XML document file. And, when you change the default **Code Generation** option in the ADF Business Components - Options page of the Project Properties dialog to generate custom Java class files, JDeveloper can also create initial custom Java class files in addition to the XML document file.

The Create Business Components from Tables wizard is particularly useful in JDeveloper because it is the only wizard that combines generating the various business component types in a single end to end process. This wizard lets you create a large number of business components quickly and easily. You can create entity objects based on an online or offline database, then you can create either entity-based view objects or query-based view objects, and an application module to contain the view instances of the data model. Finally, the wizard gives you the option to generate a business components diagram to display the relationships between the collaborating business components.

After you use a wizard, you can edit any business components you create and you can add new business components later.

Before you begin:

It may be helpful to have an understanding of the various ADF business components. For more information, see About ADF Business Components.

You will need to complete this task:

Create the data model project that will contain the business components, as described in How to Create a Data Model Project for ADF Business Components.

To create collaborating business components in one pass:

1. In the Applications window, right-click the project in which you want to create the entity objects and choose **New**.
2. In the New Gallery, expand **Business Tier**, select **ADF Business Components** and then **Business Components from Tables**, and click **OK**.

   If this is the first component you're creating in the project, the Initialize Business Components Project dialog appears to allow you to select a database connection. Complete the dialog, as described in How to Initialize the Data Model Project With a Database Connection.
3. In the Create Business Components from Tables wizard, on the Entity Objects page, filter the database schema to display available database objects and select those database objects for which you want to create entity objects. Click **Next**.

   Note that if you change a default package name, you must not use reserved words like `rest` in the package name. For details about package naming, see What You May Need to Know About Package Naming Conventions.
4. On the Entity-based View Objects page, select among the available entity objects to create updatable view object definitions for your entity object definitions. Click **Next**.

   Note that the view object definitions you create in this wizard will contain a single usage of an entity object, and will expose all the attributes in that object.
5. On the Query-based View Objects page, select among the available database objects to create read-only view object definitions. Click **Next**.
6. On the Application Module page, leave the default selections unchanged, and click **Next**.

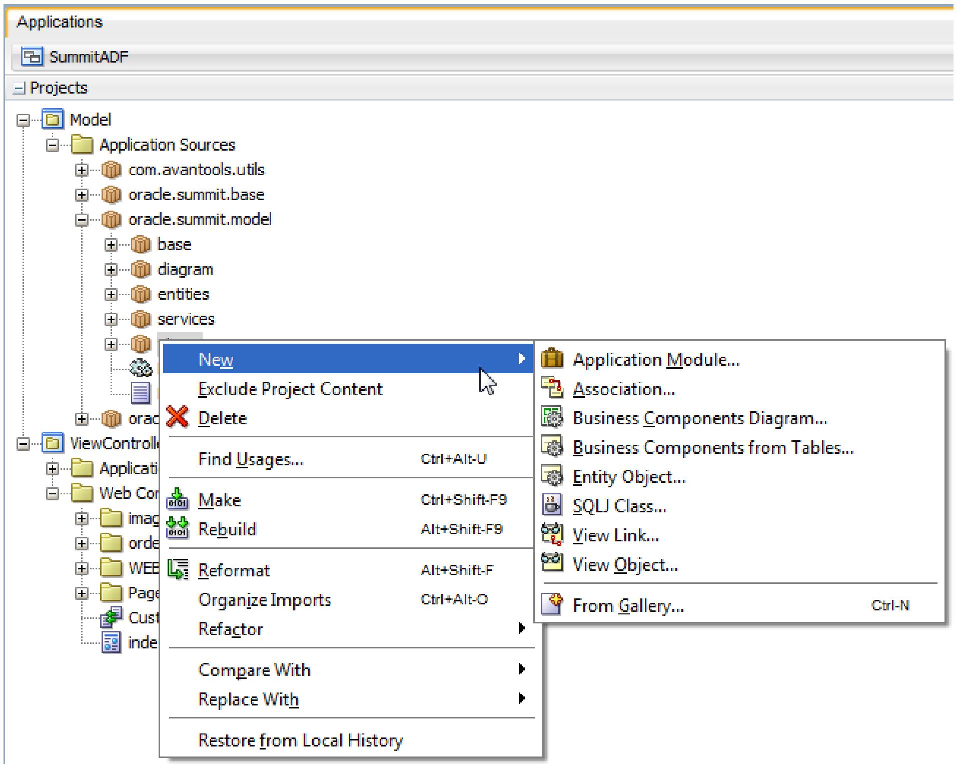
7. On the Diagram page, select **Add to Business Components Diagram**, and click **Next**.
8. On the Summary page, review the list of business components that the wizard will create, and click **Finish**

How to Create New Components Using the Context Menu

Once a package exists in the Applications window, you can quickly create additional business components of any type by selecting the component from the context menu that you display in the Applications window. To add the component, right-click the package in the data model project where you want the component to reside and select the component from the **New** menu. The components displayed in the **New** menu depend on the type of project that you have selected.

Figure 3-3 shows how the Applications window context menu displays the **New** menu after a right-click on a package in the data model project. Because the context menu is displayed on an ADF Business Components data model project, the available components are restricted to just ADF Business Components.

Figure 3-3 Context Menu Options on a Package to Create Any Kind of Business Component



The Applications window context menu displays options that are appropriate for the project that you right-click. If the context menu does not display ADF Business Components options, confirm that the project you right-clicked is an ADF Business Components model project.

Before you begin:

It may be helpful to have an understanding of the various ADF business components. For more information, see About ADF Business Components.

You will need to complete this task:

> Create the data model project that will contain the business components, as described in How to Create a Data Model Project for ADF Business Components.

To add individual business components to the data model project:

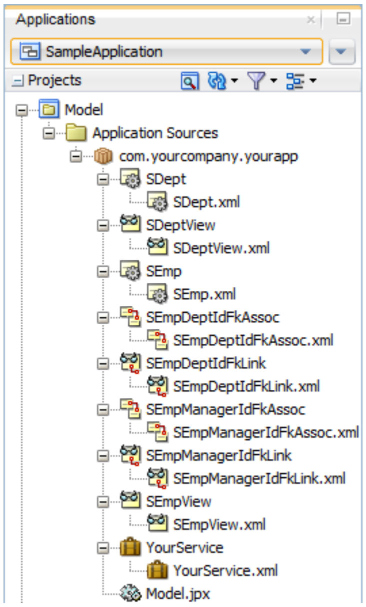
1. In the Applications window, right-click the package in the model project in which you want to create a new business component and choose **New - *business component name***.
2. In the wizard for the business component, complete the pages of the wizard and click **Finish**. For help with any of the pages of the wizard, press F1.

What Happens When You Create Business Components

Each kind of component in ADF Business Components comes with built-in runtime functionality that you control through declarative settings. These settings are stored in an XML document file with the same name as the component that it represents.

Figure 3-4 shows how the Applications window displays the XML document file for each business component in the data model project. To view XML document files in the Applications window, you can expand the node with the name assigned to the business component when it was created in JDeveloper. For example, you can expand the view object node `SDeptView` to view the associated XML document file `SDeptView.xml`. To display the overview editor for the component, you can double-click either the XML document file or its component node.

Figure 3-4   Applications Window Displays Component XML Files

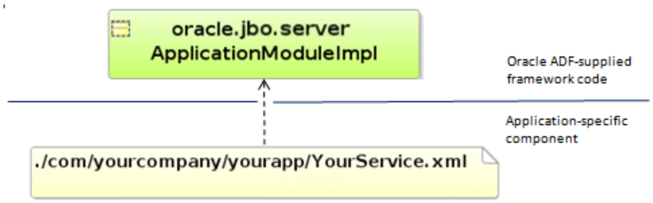


```
Applications
SampleApplication
Projects
Model
  Application Sources
    com.yourcompany.yourapp
      SDept
        SDept.xml
      SDeptView
        SDeptView.xml
      SEmp
        SEmp.xml
      SEmpDeptIdFkAssoc
        SEmpDeptIdFkAssoc.xml
      SEmpDeptIdFkLink
        SEmpDeptIdFkLink.xml
      SEmpManagerIdFkAssoc
        SEmpManagerIdFkAssoc.xml
      SEmpManagerIdFkLink
        SEmpManagerIdFkLink.xml
      SEmpView
        SEmpView.xml
      YourService
        YourService.xml
      Model.jpx
```

Figure 3-5 illustrates the XML document file for an application-specific component like an application module named *`YourService`* that you create in a package named `com.`*`yourcompany.yourapp`*. The corresponding XML document resides in a `./com/`

`yourcompany/yourapp` subdirectory of the data model project's source path root directory. That XML file records the name of the Java class it should use at runtime to provide the application module implementation. In this case, the XML records the name of the base `oracle.jbo.server.ApplicationModuleImpl` class provided by Oracle ADF.

Figure 3-5   XML Document File for an Application Module


oracle.jbo.server
ApplicationModuleImpl
Oracle ADF-supplied framework code
Application-specific component
./com/yourcompany/yourapp/YourService.xml

When used without customization, your component is completely defined by its XML document and it will be fully functional without custom Java code or even a Java class file for the component. If you have no need to extend the built-in functionality of a component in ADF Business Components, and no need to write any custom code to handle its built-in events, you can use the component in this XML-only fashion.

When you need to write custom code for a component, for example to augment the component's behavior, you can enable an optional custom Java class for the component in question, as described in How to Generate Business Component Java Subclasses.

What You May Need to Know About the Model Project Organization

Since ADF Business Components is implemented in Java, its classes and interfaces are organized into packages. Java packages are identified by dot-separated names that developers use to arrange code into a hierarchical naming structure.

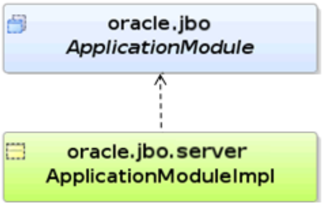
The classes and interfaces that comprise the source code provided by ADF Business Components reside in the `oracle.jbo` package and numerous subpackages. However, in day to day work with ADF Business Components, you'll work typically with classes and interfaces in these two key packages:

- The `oracle.jbo` package, which contains all of the interfaces that are designed for the business service client to work with
- The `oracle.jbo.server` package, which contains the classes that implement these interfaces

> **Note:**
>
> The term *client* here refers to any code in the model, view, or controller layers that accesses the application module component as a business service.

Figure 3-6 shows a concrete example of the application module component. The client interface for the application module is the `ApplicationModule` interface in the oracle.jbo package. This interface defines the names and signatures of methods that clients can use while working with the application module, but it does not include any specifics about the implementation of that functionality. The class that implements the base functionality of the application module component resides in the oracle.jbo.server package and is named ApplicationModuleImpl.

Figure 3-6 ADF Business Components Separate Interface and Implementation



What You May Need to Know About Package Naming Conventions

Since ADF Business Components is implemented in Java, the components of your application (including their classes, interfaces, and metadata files) will also be organized into packages.

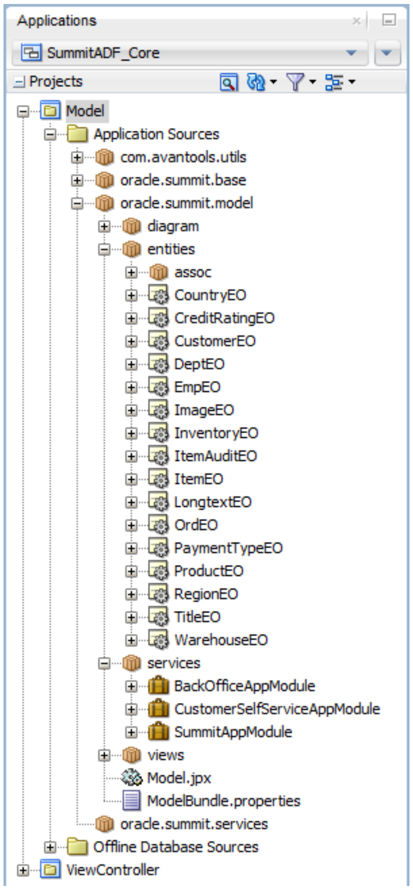
To ensure that your components won't clash with reusable components from other organizations, choose package names that begin with your organization's name or web domain name. So, for example, the Apache organization chose org.apache.tomcat for a package name related to its Tomcat web server, while Oracle picked oracle.xml.parser as a package name for its XML parser. Components you create for your own applications might reside in packages with names like com.*yourcompany*.*yourapp* and subpackages of these.

Package names that you specify must not reference the names of objects that already exist within the ADF Business Components project or include the names of certain technologies, such as rest. When you attempt to modify a default package name in an ADF Business Components wizard and use a reserved word, the wizard prevents this with an alert dialog. For example, naming a package like adf.sample.adfbc.rest is illegal in JDeveloper because rest is a reserved word.

As a specific example, the business components that make up the main business service for the Summit sample application for Oracle ADF are organized into the oracle.summit.model package and its subpackages. As shown in Figure 3-7, these components reside in the Model project in the SummitADF workspace, and are organized as follows:

- oracle.summit.model.diagram contains the business components diagram
- oracle.summit.model.entities contains the entity objects
- oracle.summit.model.services contains the SummitAppModule application module
- oracle.summit.model.views contains the view objects

Figure 3-7 Organization of ADF Business Components in the Core Summit ADF Sample Application



In your own applications, you can choose any package organization that you believe best. In particular, keep in mind that you are not constrained to organize components of the same type into a single package.

There is no optimal number of components in a package. However, the best structure should fall somewhere between the two extremes of placing all components in a single package and placing each component in its own, separate package.

One thing to consider is that the project is the unit of granularity that JDeveloper supports for reuse in other data model projects. So, you might factor this consideration into how you choose to organize components. For more information, see Packaging a Reusable ADF Component into an ADF Library.

What You May Need to Know About Renaming Components

JDeveloper supports component refactoring through specific refactoring actions that help you to incorporate changes as your application evolves. When you use JDeveloper to refactor the business components of your data model project, you can rename components or move them to a different package and JDeveloper will find and update all references to the refactored component. Whenever possible, you should use JDeveloper's refactoring actions to avoid error-prone manual editing of the project. For more information about the usages and limitations of refactoring business components in the data model project, see Refactoring a Fusion Web Application .

How to Edit Components Using the Component Overview Editor

Once a business component exists, you can edit its properties using the respective overview editor that you access either by double-clicking the component in the Applications window or by selecting it and choosing the **Open** option from the context menu.

The overview editor presents the same editing options that you see in the wizard but it may arrange them differently. The overview editor allows you to change any aspect of the component. When you make a change in the component's editor, JDeveloper updates the component's XML document file and, if necessary, any of its related custom Java files. Because the overview editor is a JDeveloper editor window, rather than a modal dialog, you can open and view the overview editor for as many components as you require.

Before you begin:

It may be helpful to have an understanding of the tools available for managing project. For more information, see the Working with Oracle JDeveloper in *Developing Applications with Oracle JDeveloper*.

You will need to complete this task:

> Create the business components in the data model project, as described in How to Create New Components Using Wizards.

To edit properties of a business component in the data model project:

1. In the Applications window, right-click the business component that you want to edit and choose **Open**.
2. In the overview editor for the business component, click the navigation tabs on the left to view and edit related groups of properties. For help with any of the pages of the overview editor, press F1.
3. In the overview editor, click the Source tab to view the changes in the XML definition source.
4. In the overview editor, click the History tab to view the revision history and compare the XML source of what appears in the current overview editor with a past revision.

   For details about managing the versions of files, see Versioning Applications with Source Control in *Developing Applications with Oracle JDeveloper*.
5. In JDeveloper, choose **File > Save** to save the property changes to the business component XML definition file.

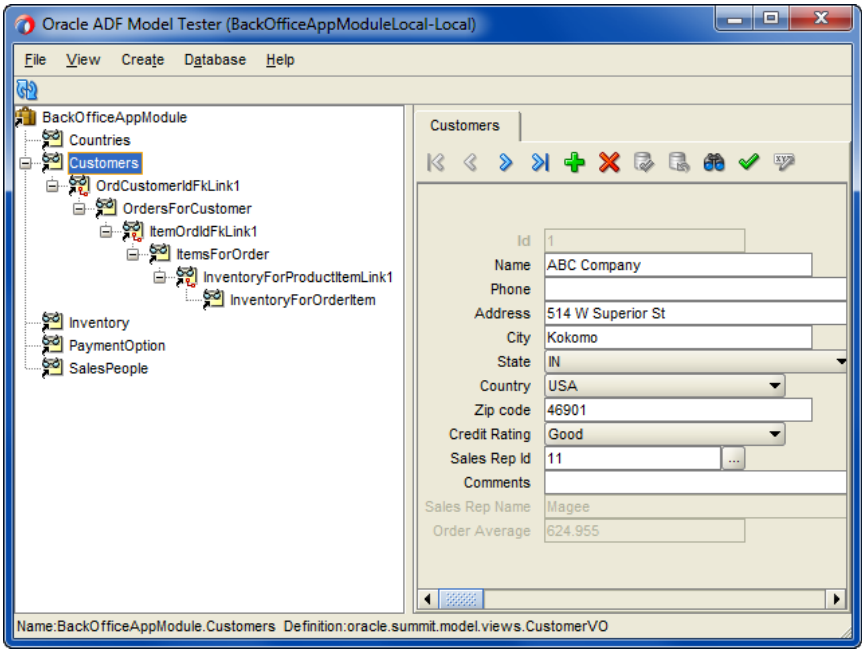
Testing, Refactoring, and Visualizing Business Components

Use the JDeveloper design time to test, refactor and visualize ADF Business Components.
JDeveloper includes comprehensive design time support to manage the growing number of business components that you create in the data model project. These facilities let you test, refactor, and visualize the business components.

How to Use the Oracle ADF Model Tester

Once you have created an application module component, you can test it interactively using the built-in Oracle ADF Model Tester. To launch the Oracle ADF Model Tester, select the application module in the Applications window or in the business components diagram and choose either **Run** or **Debug** from the context menu.

Figure 3-8   Oracle ADF Model Tester



Id: 1
Name: ABC Company
Phone:
Address: 514 W Superior St
City: Kokomo
State: IN
Country: USA
Zip code: 46901
Credit Rating: Good
Sales Rep Id: 11
Comments:
Sales Rep Name: Magee
Order Average: 624.955

Name:BackOfficeAppModule.Customers Definition:oracle.summit.model.views.CustomerVO

As shown in Figure 3-8, Oracle ADF Model Tester presents the view object instances in the application module's data model and allows you to interact with them using a dynamically generated user interface. The tool also provides a list of the application module's client interface methods that you can test interactively by double-clicking the application module node. This tool is invaluable for testing or debugging your business service both before and after you create the web page view layer.

Before you begin:

It may be helpful to have an understanding of the runtime behavior of the data model project. For more information, see Testing View Instance Queries.

You may also find it helpful to understand functionality that can be used to test Oracle ADF applications, including ADF business components. For more information, see Testing and Debugging ADF Components .

You will need to complete this task:

Create the business components in the data model project, as described in How to Create New Components Using Wizards.

To test view instances in an application module configuration:

1. In the Applications window, right-click the application module that you want to test and choose **Run**.

   Alternatively, choose **Debug** when you want to run the application in the Oracle ADF Model Tester with debugging enabled. For more information about this option, see Using the Oracle ADF Model Tester for Testing and Debugging.

2. To execute a view instance in the Oracle ADF Model Tester, expand the data model tree and double-click the desired view instance node.

3. Right-click a node in the data model tree to display the context menu for that node. For example, on a view instance node you can reexecute the query if needed, remove the view instance from the data model tree, and perform other tasks.

4. Right-click the tab of an open data viewer to display the context menu for that tab. For example, you can close the data viewer or open it in a separate window.

5. For help with the Oracle ADF Model Tester, press F1.

What You May Need to Know About Obtaining Oracle ADF Source Code

Since ADF Business Components is often used for business critical applications, it's important to understand that the full source for Oracle ADF, including ADF Business Components, is available to supported customers through Oracle Worldwide Support. The full source code for Oracle ADF can be an important tool to assist you in diagnosing problems, as described in Using the ADF Declarative Debugger. Working with the full source code for Oracle ADF also helps you understand how to correctly extend the base framework functionality to suit your needs, as described in Customizing Framework Behavior with Extension Classes.

How to Find Business Component Usages in the Data Model Project

Before you modify the names of database objects in your offline database or the names of ADF business components, you can view all objects that reference a selected item. You use the Find Usages page in the Log window to display all data model project usages of a selected item.

Before you begin:

It may be helpful to have an understanding of JDeveloper support for refactoring. See Working with Java Code in *Developing Applications with Oracle JDeveloper*.

You may also find it helpful to understand the limitations of refactoring ADF business components. For more information, see Refactoring ADF Business Component Object Attributes.

You will need to complete this task:

> Create the business components in the data model project, as described in How to Create New Components Using Wizards.

To find an item used by ADF business components:

1. In the Application window, in the data model project expand data model project and select the item for which you want to discover usages.

   To find usages of database objects, expand the Offline Database Sources folder and select the desired database object.

2. Right-click the item and choose **Find Usages**.

   To find usages of database columns, display the Structure window and right-click the column in the Structure window.

3. In the Log window, review the business components that reference the selected item.

   The Find Usages page of the Log window links the named items to editors in JDeveloper. Optionally, click an item in the list to view its details in JDeveloper.

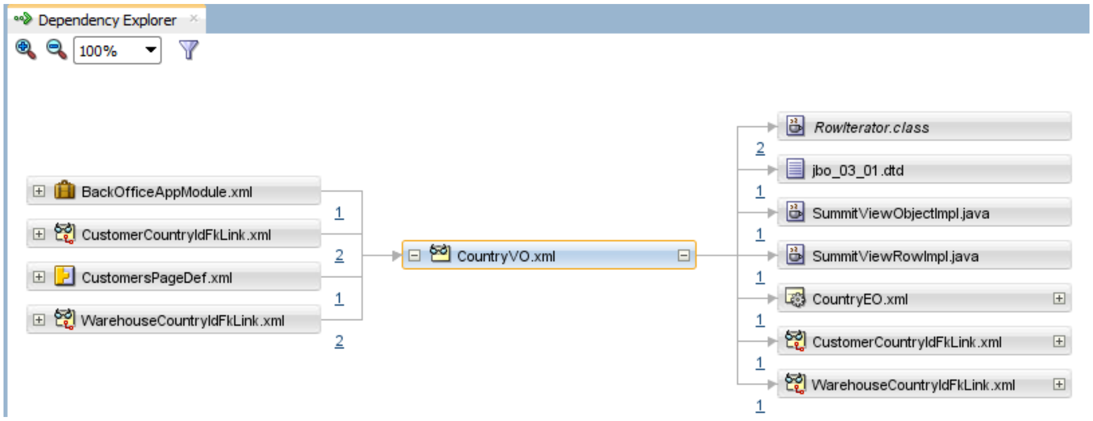
How to Explorer Business Component Dependencies in the Data Model Project

The Dependency Explorer lets you visualize the usage relationships of artifacts in the Data Model project through a diagram that you can expand. As the number of business components in your Data Model project increases, you may want to explore relationships between the business component and other project artifacts. When you expand the diagram you can display artifacts that reference the selected component and artifacts that are referenced by the selected component. In this way, the Dependency Explorer displays more complete relationship information than the usage information returned in the Find Usages log window.

Artifacts that the diagram displays for the Data Model project include the various business component XML definition files (such as, view objects, entity objects, and application module), the project configuration files (including `bc4j.xml` and `Model.jpx`), any resource bundle files (such as `ModelBundle.properties`), any Java implementation files (for example, `SummitViewObjectImpl.java` or `SummitEntityImpl.java`), and ADF Business Component framework source files (including the Oracle ADF Business Component type classes).

Figure 3-9 shows the Dependency Explorer focused on the `CountryVO.xml` file for a view object. The nodes to the left display artifacts that reference `CountryVO` and the nodes to the right are artifacts referenced by `CountryVO`. You can select **Open** from the right-click menu to open any of these artifacts in a JDeveloper source editor and view the detailed reference.

Figure 3-9 Dependency Explorer Displays ADF Business Component Artifacts



Before you begin:

It may be helpful to have an understanding of the common development tools provided by JDeveloper. For more information, see the Working with Oracle JDeveloper in *Developing Applications with Oracle JDeveloper*.

You will need to complete this task:

Create the business components in the data model project, as described in How to Create New Components Using Wizards.

To examine dependencies between ADF business components and other project artifacts:

1. In the Application window, in the data model project expand data model project and select the business component for which you want to examine dependencies.
2. Right-click the item and choose **Explore Dependencies**.
3. In the Dependency Explorer, click the plus icon on a diagram node to expand the diagram. You can expand the diagram to the left to view artifacts that reference the selected component or you can expand to the right to view artifacts referenced by the component.

   When you expand nodes to the left of the selected component you can see where the artifact is used in the application. These are the same items displayed in the Find Usages page of the Log window, when you select Find Usages on the component.
4. Click the linked numeral next to a diagram node to view the detailed list of references, and then click a reference to open the file that contains the reference.

   For more information about the Dependency Explorer, click F1.

How to Refactor Business Components in the Data Model Project

At any time, you can select a component in the Applications window and choose **Refactor > Rename** from the context menu to rename the component. The Structure window also provides a **Rename** context menu option for details of components, such as view object attributes or view instances of the **application module data model**, that do not display in the Applications window. You can also select one or more components in the Applications window by using Ctrl + click and then choosing **Refactor > Move** from the context menu to move the selected components to a new package. References to the old component names or packages in the current data model project are adjusted automatically.

Before you begin:

It may be helpful to have an understanding of JDeveloper support for refactoring. See Working with Java Code in *Developing Applications with Oracle JDeveloper*.

You may also find it helpful to understand the limitations of refactoring ADF business components. For more information, see Refactoring ADF Business Component Object Attributes.

You will need to complete this task:

Create the business components in the data model project, as described in How to Create New Components Using Wizards.

To refactor business components in the data model project:

1. In the Applications window, right-click the business component and choose **Refactor > Rename** or **Refactor > Move**.
2. To delete the object, choose **Delete**.

   If the object is used elsewhere in the application, a dialog appears with the following options:

   - **Ignore**: Unresolved usages will remain in the code as undefined references.
   - **View Usages**: Display a preview of the usages of the element in the Compiler log. You can use the log to inspect and resolve the remaining usages.

How to Refactor Offline Database Objects and Update Business Components

If you are using offline database sources with the data model project, you can rename the database, individual tables, and individual columns. JDeveloper supports refactoring of these database objects in combination with the ADF business components of the data model project, as well as the offline database itself. Changes that you make will automatically be applied to those business components whose definitions reference the database object. For example, if you change the name of a database table in the offline database, any view object queries that reference that table will be modified to use the new table name.

Before you begin:

It may be helpful to have an understanding of JDeveloper support for refactoring. For more information, see Working with Java Code in *Developing Applications with Oracle JDeveloper*.

You may also find it helpful to understand the limitations of refactoring ADF business components. See Refactoring ADF Business Component Object Attributes.

You will need to complete these tasks:

1. Create the business components in the data model project, as described in How to Create New Components Using Wizards.
2. Create the offline database in the data model project, as described in How to Change the Data Model Project to Work With Offline Database Objects.

To refactor offline database table names for ADF business components:

1. In the Application window, in the data model project expand the Offline Database Sources folder and select the database object that you want to edit.
2. In the main menu, choose **Refactor** > **Rename**.

   You can also right-click the database object and choose **Refactor** > **Rename.**
3. In the Rename dialog, enter the new name and click **Preview**.
4. In the Log window, review the business components that reference the database object name.

   The Rename page of the Log window links the named items to editors in JDeveloper. Optionally, click an item in the list to view its details in JDeveloper.
5. Click **Refactor** to apply the new names shown in the Log window.

To refactor offline database column names for ADF business components:

1. In the Application window, in the data model project expand the Offline Database Sources folder and select the database table that contains the column that you want to edit.
2. In the Structure window, double-click the column name.

   You can also right-click the database object and choose **Go to Declaration**.
3. In the overview editor for the offline database table, edit the column name and press Enter.
4. In the Log window, review the business components that reference the column name.

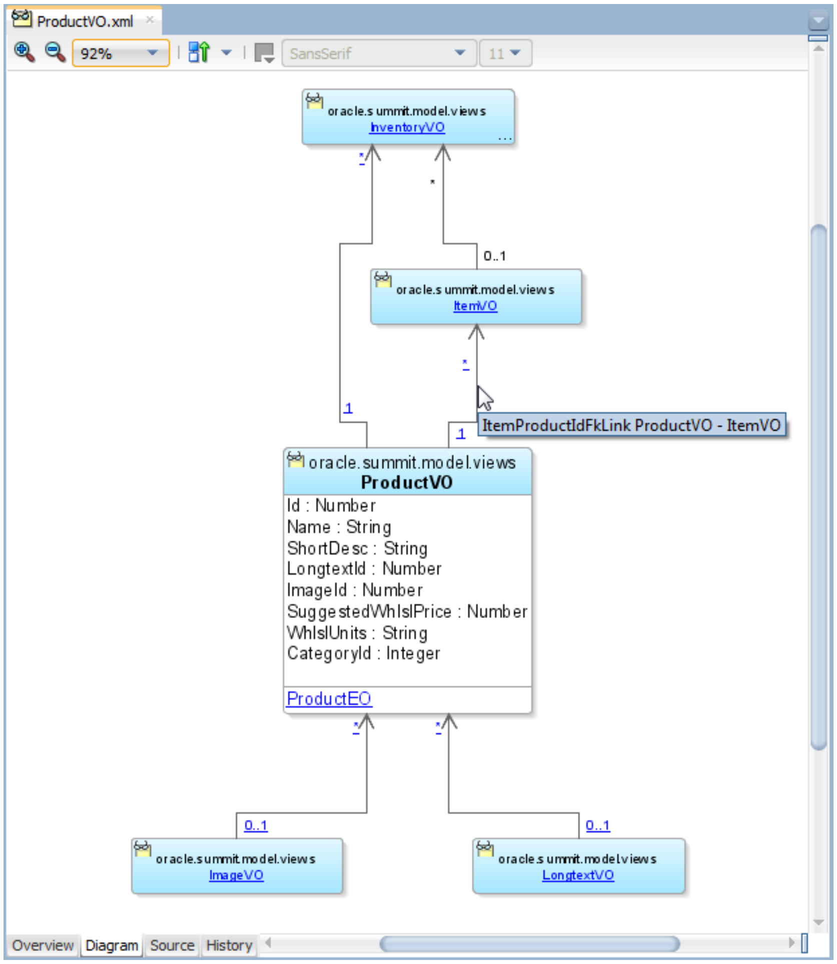
   The Rename page of the Log window links the named items to editors in JDeveloper. Optionally, click an item in the list to view its details in JDeveloper.
5. Click **Refactor** to apply the name changes shown in the Log window.

How to Display Related Business Components Using Diagrams

As the number of business components that your project defines increases, you may decide to refactor components to change the relationships that you originally created. To help you understand the relationship between components in the data model project, open any business component in the overview editor and click the **Diagram** tab. The relationship diagram in the editor identifies the component you are editing in bold text. Related components appear as link text that you can click to display the relationship diagram for the component identified by the link.

For example, Figure 3-10 displays the **Diagram** tab in the editor for the view object `ProductVO`. The diagram identifies the entity objects that `ProductVO` can access (for example, `ProductEO`), the **view link** that defines the view object's relationship to a related view object (for example, mousing over the link between `ProductVO` and `ItemVO`, displays the view link `ItemProductIdFkLink`), and the related view object (in this example, `ItemVO`) named by the view link. Each of these related components displays as a link that you can click to open the component in the **Diagram** tab for its editor. By clicking on related component links, you can use the diagrams to navigate the component relationships that your project defines.

Figure 3-10 Relationship Diagram Displays Main Object and All Related Components



Before you begin:

It may be helpful to have an understanding of JDeveloper support for modeling applications with UML class diagrams. See Developing Applications Using Modeling in *Developing Applications with Oracle JDeveloper*.

You will need to complete this task:

Create the business components in the data model project, as described in How to Create New Components Using Wizards.

To view relationship diagrams for the business components of the data model project:

1. In the Applications window, right-click the business component that you want to view and choose **Open**.
2. In the overview editor, click the Diagram tab to view the component you are editing and all of its related components in a relationship diagram.

   The relationship diagram identifies the component you are editing in bold text. Related components appear as link text that you can click to display the relationship diagram for the component identified by the link. By clicking on related component links, you can use the diagram to navigate the component relationships that your project defines.

What You May Need to Know About Using UML Diagrams

JDeveloper offers extensive UML diagramming support for ADF Business Components. You can drop components that you've already created onto a business components diagram to visualize them. You can also use the diagram to create and modify components. The diagrams are kept in sync with changes you make in the editors.

To create a new business components diagram, use the **Business Components Diagram** item in the **ADF Business Components** category of the JDeveloper New Gallery. This category is part of the **Business Tier** choices.

Customizing Business Components

JDeveloper allows you to extend or customize capabilities of ADF Business Components by creating a custom Java class.
When you need to add custom code to extend the base functionality of a component or to handle events, you must create a custom Java class for any of the key types of ADF Business Components you create, including application modules, entity objects, and view objects.

For example, you can add custom methods to the Java classes of the application module when you want to perform specific business logic. This includes, but is not limited to, code that:

- Configures view object properties to query the correct data to display
- Iterates over view object rows to return an aggregate calculation
- Performs any kind of multistep procedural logic with one or more view objects

By centralizing these implementation details in your application module, you gain the following benefits:

- You make the intent of your code more clear to clients.
- You allow multiple client pages to easily call the same code if needed.
- You simplify regression-testing of your complete business service functionality.
- You keep the option open to improve your implementation without affecting clients.
- You enable *declarative* invocation of logical business functionality in your pages.

How to Generate Business Component Java Subclasses

You can specify company-level subclasses for the entity objects, view objects, and application modules of the data model project. To enable this option so that new business components that you generate are based on these custom classes, you must create Java source files that extend the ADF Business Components implementation classes (in the `oracle.jbo` package) to which extended functionality may eventually be added. Doing this gives you the flexibility to later make changes to the base framework classes that can be picked up by their subclasses in your application.
You can then choose **ADF Business Components > Base Classes** in the Project Properties dialog to specify the names of your subclasses.

You can further enable the generation of custom classes that you wish to generate for the business components for individual components. When you enable this option on the Java page of its respective overview editor, JDeveloper creates a Java source file for a custom class related to the component whose name follows a configurable naming standard. This class, whose name is recorded in the component's XML document, provides a place where you can write further custom Java code required by that component only. Once you've enabled a custom Java class for a component, you edit the class by expanding the component in the Applications window and double-clicking the Java class.

Alternatively, you can change the project-level settings that control how JDeveloper generates Java classes for each component type, choose **Tools > Preferences** and open the **ADF Business Components** page. The settings you choose will apply to all future business components that you create.

> **Best Practice:**
>
> Oracle recommends that developers getting started with ADF Business Components consider overriding the base framework classes to enable all generated custom Java classes to pick up customized behavior. This practice is preferred over creating custom classes for individual entity objects and view objects. You use the **ADF Business Components > Base Classes** page of the Project Properties dialog to specify the framework classes to override when you generate custom component.

Before you begin:

It may be helpful to have an understanding of working programmatically with business components. For more information, see these sections:

- Working Programmatically with an Application Module's Client Interface
- Working Programmatically with Entity Objects and Associations
- Working Programmatically with Multiple Named View Criteria

You will need to complete this task:

Create the business components in the data model project, as described in How to Create New Components Using Wizards.

To override framework business component classes with a custom class:

1. In the Application window, double-click the component for which you want to create the custom class.
2. In the overview editor for the component, select the Java navigation tab.
3. On the Java page, click the **Edit Java options** icon.
4. In the Select Java Options dialog, select generate Java class options to create the desired implementation classes:
   - **Generate Application Module Class**: generates the application module implementation class that may include custom methods to encapsulate functionality that is a logical aspect of the complete business service..

- **Generate Entity Object Class**: generates the entity object implementation class that may include custom methods to act on representations of each row in the underlying database table.
- **Generate View Object Class**: generates the view object implementation class that may include custom methods to act on a particular view object definition within the application module definition.
- **Generate View Row Class**: generates the view row implementation class that may include custom methods to act on representations of a single row returned by the view object.

5. Select the options to generate methods that will support your custom code.

   In the Select Java Options dialog, when you create these classes and select **Include Accessors**, JDeveloper generates accessors for all attributes defined by the business component. Because JDeveloper uses the attribute names in the accessor method names, these attribute names will be difficult to refactor and you should edit the generated class files to remove methods that your custom code will not use.

6. Click **OK**.
7. In the Applications window, expand the business component for which you created the class and double-click the `componentNameImpl.java` file.
8. If you created an entity object class or a view row class, edit the file to remove generated accessor methods for attributes that you do not need to expose for your business use case.

> **Tip:**
>
> Edit the generated class files to remove generated code that is not used by any custom logic to improve overall performance.

How to Expose Business Component Methods to Clients

When you begin adding custom code to your ADF business components that you want clients to be able to call, you can "publish" that functionality to clients for any client-visible component. For each of your components that publishes at least one custom method to clients on its client interface, JDeveloper automatically maintains the related Java interface file. So, assuming you were working with an application module like `SummitAppModule`, you could have custom interfaces like:

- Custom application module interface

  `SummitAppModule extends ApplicationModule`

- Custom view object interface

  `OrderItemsInfo extends ViewObject`

- Custom view row interface

  `OrderItemsInfoRowClient extends Row`

Client code can then cast one of the generic client interfaces to the more specific one that includes the selected set of client-accessible methods you've selected for your particular component.

Before you begin:

You may find it helpful to have an understanding of the methods of the ADF Business Component interfaces that you can override with custom code. For more information, see Most Commonly Used ADF Business Components Methods.

You will need to complete this task:

> Generate the implementation class for the application module or view object and add the custom methods that you want to expose to the client, as described in How to Generate Business Component Java Subclasses.

To expose client methods from the component implementation class:

1. In the Application window, double-click the component for which you want to create the custom class.
2. In the overview editor for the component, select the Java navigation tab.
3. On the Java page, click the **Edit client interface** icon.
4. In the Edit Client Interface dialog, select one or more desired methods from the **Available** list and click the **Add** button to shuttle them into the **Selected** list.
5. Click **OK**.

What Happens When You Generate Custom Classes

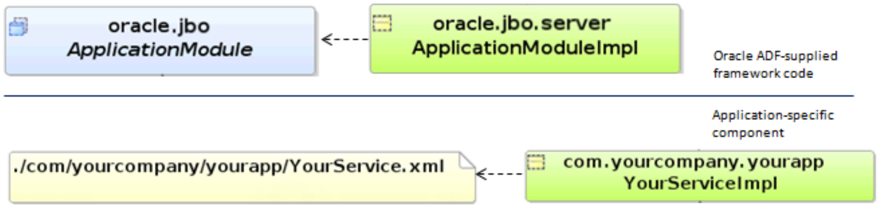
When you select one or more custom Java classes to generate, JDeveloper creates the Java file(s) you've indicated. For example, assuming an application module named `oracle.summit.model.service.SummitAppModule`, the default names for its custom Java files will be `SummitAppModuleImpl.java` for the application module class and `SummitAppModuleDefImpl.java` for the application module definition class. Both files are created in the same `./oracle/summit/model/services` directory as the component's XML document file.

> **Note:**
>
> The examples in this guide use default settings for generated names of custom component classes and interfaces. If you want to change these defaults for your own applications, use the ADF Business Components: Class Naming page of the JDeveloper Preferences dialog. Changes you make only affect newly created components.

The Java generation options for the business components continue to be reflected on subsequent visits to the Java page of the overview editor. Just as with the XML definition file, JDeveloper keeps the generated code in your custom Java classes up to date with any changes you make in the editor. If later you decide you do not require a custom Java file, disabling the relevant options on the Java page removes the corresponding custom Java files.

Figure 3-11 illustrates what occurs when you enable a custom Java class for the *`YourService`* application module. A *`YourServiceImpl`*`.java` source code file is created in the same source path directory as your component's XML document file. The *`YourServiceImpl`*`.xml` file is updated to reflect the fact that at runtime the component should use the com.*yourcompany*.*yourapp*.*YourServiceImpl* class instead of the base ApplicationModuleImpl class.

Figure 3-11 Component with Custom Java Class


oracle.jbo
ApplicationModule
oracle.jbo.server
ApplicationModuleImpl
Oracle ADF-supplied framework code
Application-specific component
./com/yourcompany/yourapp/YourService.xml
com.yourcompany.yourapp
YourServiceImpl

What You May Need to Know About Custom Interface Support

The Java interfaces of the oracle.jbo package provide a client-accessible API for your business service. Your client code works with interfaces like:

- ApplicationModule, to work with the application module
- ViewObject, to work with the view objects
- Row, to work with the view rows

Only these components of the business service are visible to the client:

- Application module, representing the service itself
- View objects, representing the query components
- View rows, representing each row in a given query component's results

This package intentionally does not contain an Entity interface, or any methods that would allow clients to directly work with entity objects. The entity objects in the business service implementation are intentionally not designed to be referenced directly by clients. Instead, clients work with the data queried by view objects as part of an application module's data model. Behind the scenes, the view object cooperates automatically with entity objects in the business services layer to coordinate validating and saving data that the end user changes. For more information about this runtime interaction, see What Happens at Runtime: How View Objects and Entity Objects Cooperate.

What You May Need to Know About Generic Versus Strongly Typed APIs

When working with application modules, view objects, and entity objects, you can choose to use a set of generic APIs or you can have JDeveloper generate code into a custom Java class to enable a strongly typed API for that component. For example, when working with an view object, if you wanted to access the value of an attribute in any row of its result, the generic API would look like this:

```
Row row = ordersVO.getCurrentRow();
Date shippedDate = (Date)row.getAttribute("OrderShippedDate");
```

Notice that using the generic APIs, you pass string names for parameters to the accessor, and you have to cast the return type to the expected type, as with `Date` shown in the example.

Alternatively, when you enable the strongly typed style of working you can write code like this:

```
OrdersRow row = (OrdersRow)ordersVO.getCurrentRow();
Date shippedDate = row.getOrderShippedDate();
```

In this case, you work with generated method names whose return type is known at compile time, instead of passing string names and having to cast the results. Typically, it is necessary to use strongly typed accessors when you need to invoke the methods from the business logic code without sacrificing compile-time safety. This can also be useful when you are writing custom validation logic in setter methods, although in this case, you may want to consider using Groovy expressions instead of generating entity and view row implementation classes for Business Components. Subsequent chapters explain how to enable this strongly typed style of working by generating Java classes for business logic that you choose to implement using Java.

Using Groovy Scripting Language With Business Components

Groovy is a scripting language with Java-like syntax for the Java platform. The Groovy scripting language simplifies the authoring of code by employing dot-separated notation, yet still supports syntax to manipulate collections, Strings, and JavaBeans.

Currently, Groovy expressions are supported on entity object and view object business components. The user can utilize Groovy expressions in ADF Business Components to specify runtime-determined values for view criteria, bind variables, default values of entity or view object attributes, business rules, triggers and more. Additionally, ADF Business Components provides a limited set of built-in keywords that can be used in Groovy expressions. See What You May Need to Know About Where Groovy Expressions May Be Used in the Model Project for details about what you can do using Groovy scripting language.

Generation of the Business Components Script (.bcs) file

When you enter Groovy expressions for an entity object or view object, all of these Groovy expressions populate into a `.bcs` (Business Components Script) file for this business component. View the `.bcs` file for a business component as one among a list of artifacts (such as the Java class, XML file, Operations XML file) belonging to a particular entity object or view object. See How to Enter Groovy Expressions on Business Component to learn how to enter Groovy expressions at various UI contexts in JDeveloper, depending on the context and purpose. The `.bcs` file offers debugging capabilities, and functions like a regular Java editor. Set breakpoints, import annotation packages, compile the Groovy syntax, and fix errors displayed in the Log window.

Type-checking support for Groovy expressions

JDeveloper supports type-checking for Groovy expressions. Type-checking scrutinizes the syntax of Groovy expressions that you create at design time and identifies errors and displays these errors in the JDeveloper Log window. Rectify these errors in your ADF Model project to successfully execute these Groovy expressions at runtime without errors. By default, type-checking is enabled. Type-checking can be disabled.

If type-checking is disabled, type-checking is performed only during runtime. However, it is good practice to type-check Groovy expressions before runtime. You can enable or disable Groovy type-checking for all projects that are created, and/or for specific projects. See What You May Need to Know About Groovy Project Settings for details about application-wide and project-specific Groovy type-checking settings.

> **Note:**
>
> In order to minimize the execution overhead of the groovy script engine, ADF may use an optimized execution path for simple scripts that directly return String or Boolean literals or use a path that directly return a single attribute value. No developer intervention is required to take advantage of this feature. However, the script developer should keep scripts simple, whenever possible.

Script Class File Support

The application allows you to create a custom script class file. Use Script Classes to define a class that can be used by other script classes. A script class file allows you to maintain common Groovy script methods that can be called from within any business component class. This is unlike a regular `.bcs` file that also maintains Groovy scripts that can only be called by the component that this `.bcs` file is associated. See How to Create a Script Expression Class File for details.

How to Enter Groovy Expressions on Business Components

JDeveloper allows you to define Groovy expressions for entity objects, view objects, and view object usages (on an application module). Enter Groovy expressions in the Edit Expression Editor dialog. You invoke this editor in various UI contexts of the application. ADF Business Components supports the use of the Groovy scripting language in places where access to entity object and view object attributes is useful, including attribute validators (for entity objects), attribute default values (for either entity objects or view objects), transient attribute value calculations (for either entity objects or view objects), bind variable default values (in view object query statements and **view criteria** filters), and placeholders for error messages (in entity object validation rules).

Specifically, ADF Business Components provides support for the use of Groovy language expressions to perform the following tasks:

- Add a Script Expression validator or Compare validator (see Using Groovy Expressions For Business Rules and Triggers)
- Define error message tokens for handling validation failure (see How to Embed a Groovy Expression in an Error Message)
- Handle conditional execution of validators (see How to Conditionally Raise Error Messages Using Groovy)
- Set the default value of a bind variable in the view object query statement (see Working with Bind Variables)
- Set the default value of a bind variable that specifies a criteria item in the view criteria statement (see Working with Named View Criteria).

- Define the default value and optional recalculate condition for an entity object attribute (see How to Define a Static Default Value)
- Determine the value of a transient attribute of an entity object or view object (see Adding Transient and Calculated Attributes to an Entity Object and Adding Calculated and Transient Attributes to a View Object)

> **Note:**
>
> These instructions to define the default value of an attribute using Groovy expressions serve as an example to demonstrate how to enter Groovy expressions in the Edit Expression Editor dialog. Given this premise, there is a mention of the Refresh Expression Value, which is defined by you also using the Edit Expression Editor dialog. Now the Refresh Expression Value is specific to defining the default value of an attribute. Therefore, any activity that involves entering Groovy expressions in the Edit Expression Editor dialog may or may not have the associated activity of defining a Refresh Expression Value.

To define the default value of an attribute using Groovy expressions:

1. Open the XML editor view of the entity object or view object. For example, for an entity object named Country, double-click on `CountryEO.xml` to open the Overview Editor of this entity object.
2. Click the **Attributes** tab.
3. Navigate to the **Details** page of the **Attributes** tab and to the **Default Value** section.
4. Click the **Expression** radio-button.
5. Click the icon beside to open the Edit Expression Editor dialog to define the Groovy expression that calculates the default value and click **OK**.
6. If required, in the field immediately below the default value expression as described in the aforementioned step, enter a Refresh Expression Value. Use the same process in the aforementioned step to define the Refresh Expression Value.

What Happens When You Enter Expressions

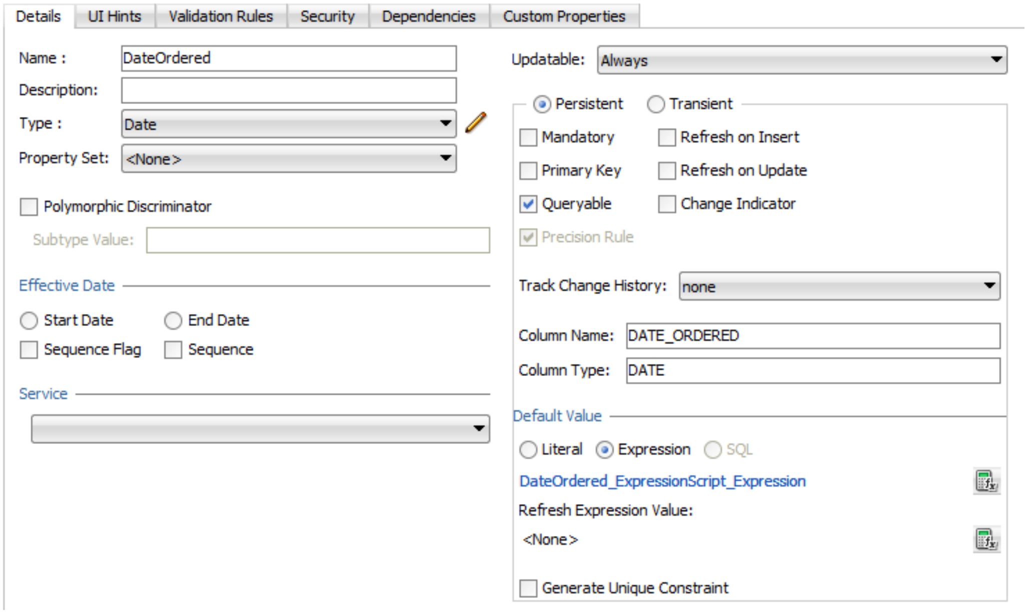
When you enter Groovy expressions for a business component such as an entity object or view object and save the project, the entered Groovy expressions populate in the `.bcs` file associated with the component. Note that to ensure this happens, you must not disable the setting that controls Groovy file generation, as described in What You May Need to Know About Groovy Project Settings. The application provides you a mechanism to enter Groovy expressions in various UI contexts for an entity object or view object. View these added Groovy expressions in the `.bcs` file associated with the business component.

A Groovy expression is represented as an hyperlink when you add Groovy expressions for a business component in the following UI contexts.

- After defining a Groovy expression for an attribute’s default value and/or setting the refresh expression value.
- After defining UI hints such as label, tooltip and so forth for an attribute.

Figure 3-12 shows a Groovy expression added on a view object, to set the default value of this view object's attribute. The expression is represented as a hyperlink. The hyperlink links to the expression script in the `.bcs` file. Click this hyperlink to open the `.bcs` file associated with this business component.

Figure 3-12 Groovy Expression Represented as a Hyperlink

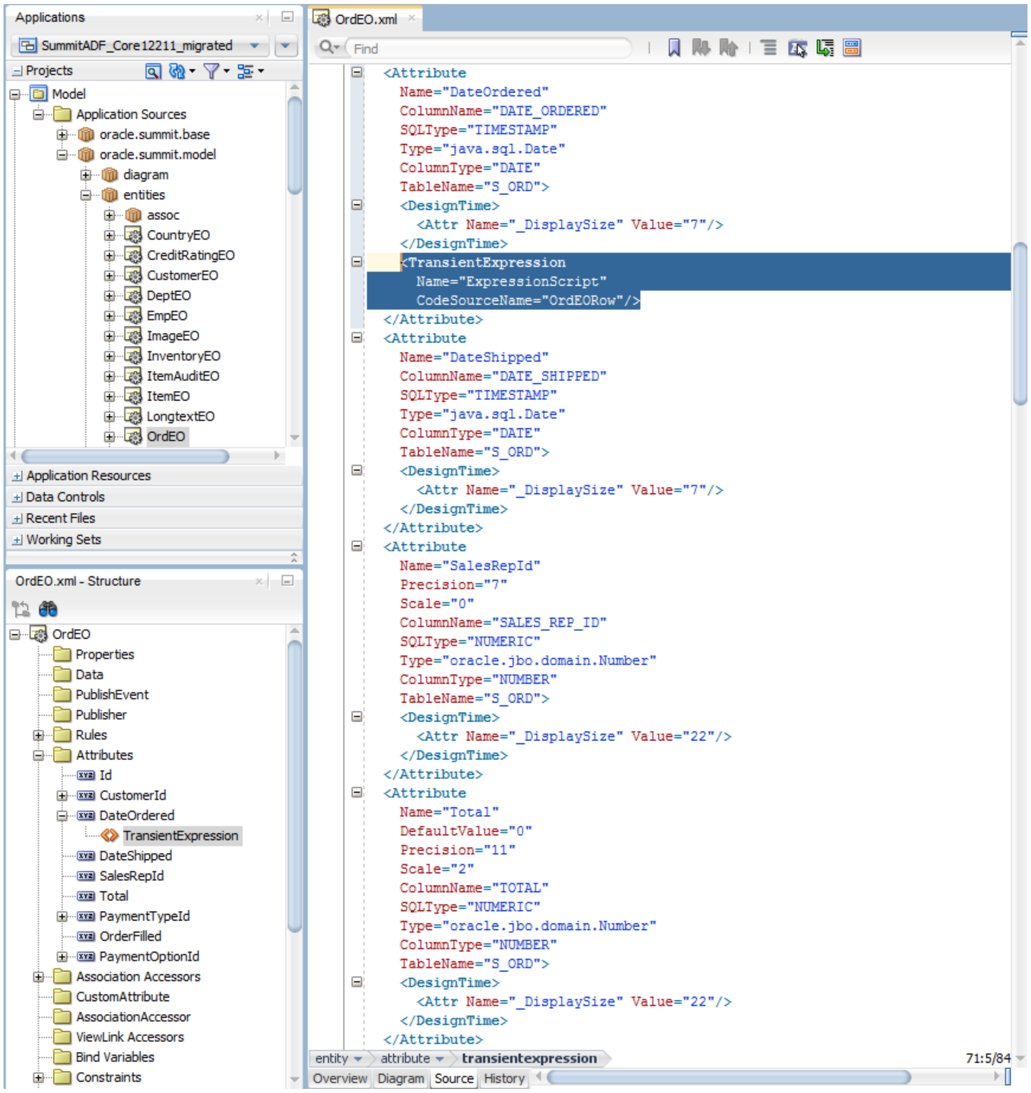


Name : DateOrdered
Description:
Type : Date
Property Set: <None>
□ Polymorphic Discriminator
Subtype Value:
Effective Date
○ Start Date ○ End Date
□ Sequence Flag □ Sequence
Service

Updatable: Always
● Persistent ○ Transient
□ Mandatory □ Refresh on Insert
□ Primary Key □ Refresh on Update
☑ Queryable □ Change Indicator
☑ Precision Rule
Track Change History: none
Column Name: DATE_ORDERED
Column Type: DATE
Default Value
○ Literal ● Expression ○ SQL
DateOrdered_ExpressionScript_Expression
Refresh Expression Value:
<None>
□ Generate Unique Constraint

You can also view Groovy expressions added for an entity object or view object in the component's Structure window. Figure 3-13 shows a Groovy expression added for the `DateOrdered` attribute of the **OrdEO** component in the Structure window in the left-hand bottom side of the screen.

Figure 3-13 Groovy Expression Represented in the Component's Structure Window



```
<Attribute
  Name="DateOrdered"
  ColumnName="DATE_ORDERED"
  SQLType="TIMESTAMP"
  Type="java.sql.Date"
  ColumnType="DATE"
  TableName="S_ORD">
  <DesignTime>
    <Attr Name="_DisplaySize" Value="7"/>
  </DesignTime>
  <TransientExpression
    Name="ExpressionScript"
    CodeSourceName="OrdEORow"/>
</Attribute>
<Attribute
  Name="DateShipped"
  ColumnName="DATE_SHIPPED"
  SQLType="TIMESTAMP"
  Type="java.sql.Date"
  ColumnType="DATE"
  TableName="S_ORD">
  <DesignTime>
    <Attr Name="_DisplaySize" Value="7"/>
  </DesignTime>
</Attribute>
<Attribute
  Name="SalesRepId"
  Precision="7"
  Scale="0"
  ColumnName="SALES_REP_ID"
  SQLType="NUMERIC"
  Type="oracle.jbo.domain.Number"
  ColumnType="NUMBER"
  TableName="S_ORD">
  <DesignTime>
    <Attr Name="_DisplaySize" Value="22"/>
  </DesignTime>
</Attribute>
<Attribute
  Name="Total"
  DefaultValue="0"
  Precision="11"
  Scale="2"
  ColumnName="TOTAL"
  SQLType="NUMERIC"
  Type="oracle.jbo.domain.Number"
  ColumnType="NUMBER"
  TableName="S_ORD">
  <DesignTime>
    <Attr Name="_DisplaySize" Value="22"/>
  </DesignTime>
</Attribute>
```

JDeveloper represents Groovy expressions in an entity object or view object XML source (visible in the Source tab in the overview editor of the business component), as well as the `.bcs` file associated with this component. In the XML source view of the business component, XML structures define references to Groovy expressions, where the Groovy expressions are represented as scripts in the `.bcs` file for this business component. Annotations in the `.bcs` file provide information about the script, such as the business component on which the Groovy expression is defined, attribute name, and so forth. Therefore, some of the information enclosed as part of the XML definition of a Groovy expression is utilized in the annotations. Importantly, an XML definition of a Groovy expression uses the `CodeSourceName` value to identify the `.bcs` file where this Groovy expression resides as a script. The `CodeSourceName` value is assigned the value of the `operations.xml` file. The `operations.xml` file is another XML file that is associated with a business component that contains the URI of the `.bcs` file.

The following sample shows the resulting XML source of the `DateOrdered` attribute of the `OrdEO` entity object that displays the XML definition referencing a Groovy expression.

```
<Attribute
  Name="DateOrdered"
  ColumnName="DATE_ORDERED"
  SQLType="TIMESTAMP"
  Type="java.sql.Date"
  ColumnType="DATE"
  TableName="S_ORD">
  <DesignTime>
    <Attr Name="_DisplaySize" Value="7"/>
  </DesignTime>
  <TransientExpression
    Name="ExpressionScript"
    CodeSourceName="OrdEORow"/>
</Attribute>
```

And, the following sample shows the `.bcs` file that displays the Groovy script definition for the `DateOrdered` attribute.

```
@TransientValueExpression(attributeName="DateOrdered")
def DateOrdered_ExpressionScript_Expression()
{
  adf.currentDate
}
```

As the samples for the XML source and `.bcs` file show, the `name="DateOrdered` in the XML source corresponds to the method `nameDateOrdered_ExpressionScript_Expression()` in the annotation of the `.bcs` file. These values identify the expression script method name which defines the Groovy expression for this `DateOrdered` attribute. The `CodeSourceName="OrdEORow"` in the XML source indicates the `operations.xml` file, which for the `OrdEO` entity object is `OrdEOOperations.xml`. The `operations.xml` file has a URI which points to the `.bcs` file.

What You May Need to Know About Groovy Project Settings

Enable Groovy type-checking to allow the application to type-check Groovy expressions in the `.bcs` file. Enable Groovy file generation to allow the application to populate all Groovy expressions for a component in its associated `.bcs` file. Groovy type-checking and/or Groovy file generation can be enabled or disabled for a specific Model project or across all projects.

Groovy Expression Type Validation

In JDeveloper from the main menu, navigate to **Application** > **Project Properties** then select **ADF Business Components** from the **Project Properties** dialog and then click the plus icon to expand and select **Options**. On the Options page, select the **Groovy Expression Type Validation** checkbox to enable JDeveloper to perform Groovy expression type validation for this Model project. To set this property for all projects across the application, navigate to **Tools** > **Preferences** and select **ADF Business Components** from the **Preferences** dialog box, and then click the plus icon to expand and select **General**. On the General page, select **Groovy Expression Type Validation** checkbox. When you compile the project, Groovy expressions in the entity object and view object-specific Business Components Script (`.bcs`) files are compiled and errors are reported in the JDeveloper Log window. Deselect this option when you want to prevent type validation from occurring on the Groovy expression scripts defined in the .bcs files; type-checking is disabled for any new projects created from this point onwards, whereas existing projects will continue to receive type-checking. It is recommended that you leave this option enabled (default) to support debugging Groovy expression scripts contained in the component's `.bcs` files. While disabling type checking does not disable debugging of Groovy scripts, leaving type checking enabled does surface problems at compile time instead of at runtime, and will thus make Groovy script debugging easier.

> **Note:**
>
> It is possible to disable type validation at the script level using `.bcs` file annotations. In the `.bcs` file, enter the annotation `@TypeChecked(TypeCheckingMode.SKIP)` above the Groovy expression. In the header section of the `.bcs` file, these import statements-`import groovy.transform.TypeCheckingMode` and `import groovy.transform.TypeChecked` allow the application to comply with the annotations you specify. When you compile Groovy expressions in the `.bcs` file, type-checking is skipped or omitted for the expression for which this annotation is supplied.

Enable Groovy File Generation

In JDeveloper from the main menu, navigate to **Application** > **Project Properties** and select **ADF Business Components** from the **Project Properties** dialog, and then click the plus icon to expand and select **Options** . In the Options page, select the **Enable Groovy File Generation** checkbox to enable JDeveloper to generate Business Components Script (`.bcs`) files in this Model project for the entity objects and view objects with Groovy expressions (default). The `.bcs` file is created on the component, when you define a Groovy expression in the component overview editor (for example, when you enter a default value expression on an entity object attribute or when you define validation rule expressions). Deselect this option when you do not want JDeveloper to aggregate Groovy expressions in `.bcs` files.

If you have already entered Groovy expressions with this checkbox selected, and later deselect the checkbox, the generated `.bcs` files retain all previously added Groovy expressions, but Groovy expressions that you define going forward will not populate in the `.bcs` files and will be saved in the component's XML definition (`.xml`) file. It is recommended that you leave this option enabled (default) to aggregate all expressions into entity object and view object-specific `.bcs` files, where you can edit Groovy expression scripts and compile them for errors. Also, aggregating expressions into Business Components Script files ensures the best runtime performance.

> **Note:**
>
> Users migrating ADF applications to JDeveloper release 12.2.1 and later can retain expressions in the Model project component `.xml` files by clearing this checkbox. However, it is recommended that users leave this checkbox selected and make use of the JDeveloper-provided migration facilities to move all Groovy expressions to `.bcs` files, where script editing and compilation can be performed in the Java source editor.

What You May Need to Know About Where Groovy Expressions Can Be Used

Table 3-3 Groovy Expression Usage in ADF Business Components

Expression Usage	Component Usage	XML Element/Parent XML Element	Description
Transient Attribute Value	entity object / view object	TransientExpression/ Attribute	Calculates a transient attribute value at the row level. Scripts are defined at the row level.
Attribute Default Value	entity object / view object	TransientExpression/ Attribute	Calculates the attribute default value. Scripts are defined at row level.
Attribute Default Value Recalculation	entity object / view object	RecalcCondition/Attribute	When this value evaluates to be true, the default value is recalculated. Scripts are defined at row level.
Variable Default Value	entity object / view object	TransientExpression/ Variable	Calculates the variable default value. Scripts are defined at the object level.
Expression Validator	entity object / view object	TransientExpression/ ExpressionValidationBean (Attribute, ViewAttribute, or Entity)	Executes this expression to validate the current contents of the individual attribute or the whole entity. Scripts are defined at row level.
Compare Validator	entity object / view object	TransientExpression/ CompareValidationBean (under Attribute, ViewAttribute, Entity)	Calculates a value to compare with the new attribute value. Scripts are defined at row level.
Validator OnCondition	entity object / view object	OnCondition/ ExpressionValidationBean, CompareValidationBean, etc. (under Attribute, ViewAttribute, Entity)	Calculates a value to compare with the new attribute value. Scripts are defined at row level.

Table 3-3 (Cont.) Groovy Expression Usage in ADF Business Components

Expression Usage	Component Usage	XML Element/Parent XML Element	Description
Validator Message Parameter	entity object / view object	Expression/ ResExpressions (under all validation types)	In the validator, you can have a text resource to provide an error message when the validator detects an error. Inside the text resource string, you can specify the name of a variable within curly braces {}. This variable refers to a Groovy expression that calculates a value to embed in the text resource. Scripts are defined at row level.
UI Hints	entity object / view object	TransientExpression/ LABEL	In each attribute there are a number of SchemaBasedProperties that are intended to give hints to UI developers as to how the UI should appear for this attribute. Some of these are text resources, such as LABEL and some are just values, such as DISPLAYWIDTH. Any of these UI Hints could be a Groovy expression, calculating the value for the UI Hint. Scripts are defined at row level.
Entity Triggers	entity object	TransientExpression/ ExpressionValidationBean (under Trigger)	The name property under the trigger element identifies the type of this trigger. This expression is fired when the trigger event occurs. Scripts are defined at row level.
View Accessor Parameter	entity object / view object	TransientExpression/PIMap (under ParaeterMap which is under ViewAccessor)	If the view accessor utilizes a view criteria and inside the view criteria an item uses a Bind Variable; this expression provides the value for the Bind Variable. Scripts are defined at the object level.
View Usage Parameter	application module	TransientExpression/PIMap (under ParaeterMap which is under ViewUsage)	If the view usage utilizes a view criteria and inside the view criteria an item uses a Bind Variable; this expression provides the value for the Bind Variable. Scripts are defined at the object level.

Table 3-3 (Cont.) Groovy Expression Usage in ADF Business Components

Expression Usage	Component Usage	XML Element/Parent XML Element	Description
RowFinder Parameter	view object	TransientExpression/ PIMap (under ParaeterMap which is under ViewUsage)	If the rowfinder utilizes a view criteria and inside the view criteria an item uses a Bind Variable; this expression provides the value for the Bind Variable. Scripts are defined at the object level.
View Criteria CompOper	view object	TransientExpression/ CompOper	If the view criteria implements a custom operator, this expression can be used to execute the custom operator. Scripts are defined at the object level.

What You May Need to Know About Groovy Expression Syntax

The following sections provide important details about the syntax of the Groovy expression language in ADF Model projects. Groovy expressions work interactively with ADF Business Components to let you perform numerous activities. There's a lot you can do using Groovy scripts: such as, using `adf.context` in a Groovy expression in order to reference the `ADFContext` object, invoking either custom or ADF Business Components Java APIs, and manipulating business component attribute values using built-in aggregate functions.

What You May Need to Know About Referencing Business Components in Groovy Expressions

There is one top-level object named `adf` that allows you access to objects that the framework makes available to the Groovy script. When you reference an Oracle ADF object in a Groovy expression, the Oracle ADF runtime returns wrapper objects that do not correspond to the actual concrete type of the classes. These wrapper objects support all of the method and field types of the wrapped object. Your expressions can use wrapped objects as if they were the actual object. Note, however, any attempt to cast wrappered objects to its concrete type will fail with a `ClassCastException`. In general, when working with the Groovy language it is not necessary to use explicit casting, and in the case of these wrapped ADF Business Components objects, doing so will cause an exception.

The accessible Oracle ADF objects consist of the following:

- `adf.context` - to reference the `ADFContext` object
- `adf.object` - to reference the object on which the expression is being applied (which can also be referenced using the keyword `object`, without the `adf` prefix). Other accessible member names come from the context in which the Groovy script is applied.

  - Entity object attributes: The context is an instance of the entity implementation class. Through this object you can reference custom methods of the custom entity implementation class, any methods defined by the base implementation class as specified by the Javadoc for `EntityImpl`, and you can reference the attributes of the entity instance.
  - Entity object script validation rules: The context is the validator object (`JboValidatorContext`) merged with the entity on which the validator is applied. For details about keywords that you can use in this context, see Referencing Members of the Same Business Component.
  - View object attributes: The context is an instance of the view row implementation class. Through this object, you can reference custom methods of the custom view row implementation class, any methods defined by the base implementation class as specified by the Javadoc for `ViewRowImpl`, and you can reference the attributes of the view row instance as defined by the query row set. The
  - Bind variable in view object query statements: The context is the variable object itself not the view row. You can reference the `structureDef` property to access other information as well as the `viewObject` property to access the view object in which the bind variable participates. Note that access to view object attributes through the `viewObject` property is not supported.
  - Bind variable in view criteria: The context is the current view criteria row. The view criteria with bind variable may be used to create a query search form component in which a bind variable expression can provide the list of search criteria values. You must reference the `structureDef` property to access the view criteria row, and you can then reference the `findAttributeDef()` method to derive values from a specified attribute. You cannot use the `viewObject` property to access the view object on which the view criteria is defined.
  - Bind variable in view accessors: The context is the current view row. The **view accessor** with bind variable is used to create a cascading List of Value (LOV). The view accessor can derive Groovy-driven values from the current view row in the view accessor view object used to formulate the list of valid choices. You must reference the `structureDef` property to access the view row, and you can then reference the `findAttributeDef()` method to derive values from a specified attribute. You cannot use the `viewObject` property to access the view object for which the view accessor is defined.
  - Transient attributes: The context is the current entity or view row. You can reference attributes by name in the entity or view row in which the attribute appears, as well as public methods on that entity or view row. To access methods on the current object, you must use the `object` keyword to reference the current object (for example, `object.methodName( )`). The `object` keyword is equivalent to the `this` keyword in Java. Without it, in transient expressions, the method will be assumed to exist on the dynamically compiled Groovy script object itself.
- `adf.error` - in validation rules, to access the error handler that allows the validation expression to generate exceptions or warnings
- `adf.userSession` - returns a reference to the ADF Business Components user session (which you can use to reference values in the `userData` hashmap that is part of the session)

You can also reference the current date (time truncated) or current date and time using the following expressions:

- `adf.currentDate`
- `adf.currentDateTime`

What You May Need to Know About Untrusted Groovy Expressions

The ADF Groovy engine restricts Groovy access to general purpose Java APIs. This is done to prevent illegal and/or insecure method calls and property access. An application developer may enhance the security policy to include custom methods defined in the application Java API with the `@AllowUntrustedScriptAccess` annotation.

Custom Java may be used to access Java libraries that are not otherwise supported in ADF scripts. For example, if one wanted to read from a file, in order to support that case, you can create a custom Java class, mark it with the `@AllowUntrustedScriptAccess` annotation, and then invoke the custom class from an ADF script.

In the class file, you need to:

1. Import `oracle.adf.share.security.AllowUntrustedScriptAccess`.
2. Add the annotation `@AllowUntrustedScriptAccess(methodNames={"`*`xyz`*`", "`*`abc`*`"})` at the top of the view object implementation class.

   For example:

   ```
   import oracle.adf.share.security.AllowUntrustedScriptAccess;
   @AllowUntrustedScriptAccess(methodNames = {"testAccessObject"})

   public class ActivityVOImpl extends CRMViewObjectImpl implements
   ActivityVO
   {
       ......

       public boolean testAccessObject(String objectType, Long
   objectKey)
       {
         // your logic here
         return true;
       }
   ```

After you have updated the class file, you can directly access the Java method from the `VOImpl` class inside the Groovy script, just as you would in any source code. For example, to access the `testAccessObject()` method, you would need to invoke the method on the object that you create, as illustrated by the following example.

```
def voActivity = newView('Activity')
def objectType = 'OPPTY'
def objectKey = 100010025532672
voActivity.testAccessObject(objectType, objectKey)
println('Done test');
```

What You May Need to Know About Referencing Custom Business Components Methods and Attributes in Groovy Expressions

Groovy script language simplifies the authoring of code that you might write to access methods and attributes of your entity object and view objects.

Referencing Members of the Same Business Component

The simplest example of referencing business component members, including methods and attributes that the entity object and view object define, is to reference attributes that exist in the same entity object or view object as the attribute that you apply the expression.

For example, you could define a Groovy expression to calculate the value of a transient attribute `AnnualSalary` on an entity object with an attribute `Sal` that specifies the employee's monthly salary:

```
Sal * 12
```

Or, with Groovy you can write a simple validation rule to compare the attributes of a single view object using syntax like:

```
PromotionDate > HireDate
```

Using Java, this same comparison would look like:

```
((Date)getAttribute("PromotionDate")).compareTo((Date)getAttribute("HireDate")) > 0
```

Note that the current object is passed in to the script as the `this` object, so you can reference an attribute in the current object by simply using the attribute name. For example, in an attribute-level or entity-level Script Expression validator, to refer to an attribute named "HireDate", the script can simply reference `HireDate`.

Similar to referencing attributes, when you define custom methods in an entity implementation class, you can invoke those methods as part of your expression. For example, to define an attribute default value:

```
adf.object.getDefaultSalaryForGrade()
```

A method reference requires the prefix `adf.object` which allows you to reference the same entity that defines the attribute on which the expression is applied. This same prefix also allows you to reference the methods of the base class of the entity implementation class (`EntityImpl.java`) that your custom implementation class extends.

Note that when you want to reference the method of an entity implementation class in a validation rule, you use the `source` prefix:

```
source.getDefaultSalaryForGrade()
```

Use of the `source` prefix is necessary in validators because the `object` keyword implies the validation rule object instead of the entity object (where the method is defined).

To allow you to reference members of the validator object (`JboValidatorContext`), you can use these keywords in your validation rule expression:

- `newValue`: in an attribute-level validator, to access the attribute value being set
- `oldValue`: in an attribute-level validator, to access the current value of the attribute being set

For example, you might use the following expression to specify a dynamic validation rule check of the salary for a salesman.

```
if (Job == "SALESMAN")
{
  return newValue < source.getMaxSalaryForGrade(Job)
}
else
return true
```

Referencing Members of Other Business Components

You can also reference the methods and attributes that entity objects and view objects defines in the expressions you apply to a different entity object attribute or validation rule. This is accomplished by referencing the accessor in the **entity association**.

For example, if you define an entity with a master-detail association for `Dept` and `Emp`, by default the accessor for the entity association will be named `Dept` and `Emp`, to identity the source and destination data source. Using that accessor in a Groovy expression to set the default value for a new employee's salary based on the location of their department:

```
adf.object.getDefaultSalaryForGrade(Dept.Loc)
```

This expression does not reference the entity even though it has the same name (`Dept`) as the accessor for the association. Instead, assuming a **master-detail relationship** between departments and employees, referencing the accessor allows the Groovy expression for the employee entity object to walk back to the master department entity and pass in the value of `Loc` from that master.

What You May Need to Know About Manipulating Business Component Attribute Values in Groovy Expressions

You can use the following built-in aggregate functions on Oracle Business Components `RowSet` objects:

- `*rowSetAttr*.sum(*GroovyExpr*)`
- `*rowSetAttr*.count(*GroovyExpr*)`
- `*rowSetAttr*.avg(*GroovyExpr*)`
- `*rowSetAttr*.min(*GroovyExpr*)`
- `*rowSetAttr*.max(*GroovyExpr*)`

These aggregate functions accept a string-value argument that is interpreted as a Groovy expression that is evaluated in the context of each row in the row set as the aggregate is being computed. The Groovy expression must return a numeric value (or number domain).

For example, in a `Dept` entity object you could add a transient attribute that displays the sum of all employee salaries that is calculated by this expression:

```
EmployeesInDept.sum("Sal")
```

To reference the employees of a specific department, the expression supplies the name of the master-detail association's accessor for the destination `Emp` entity. In this case, the accessor is `EmployeesInDept` and salary is interpreted for each record of the `Emp` entity object.

Or, assume that you want the calculation of the salary total for specific departments to include each employee's benefits package, which varies with job role:

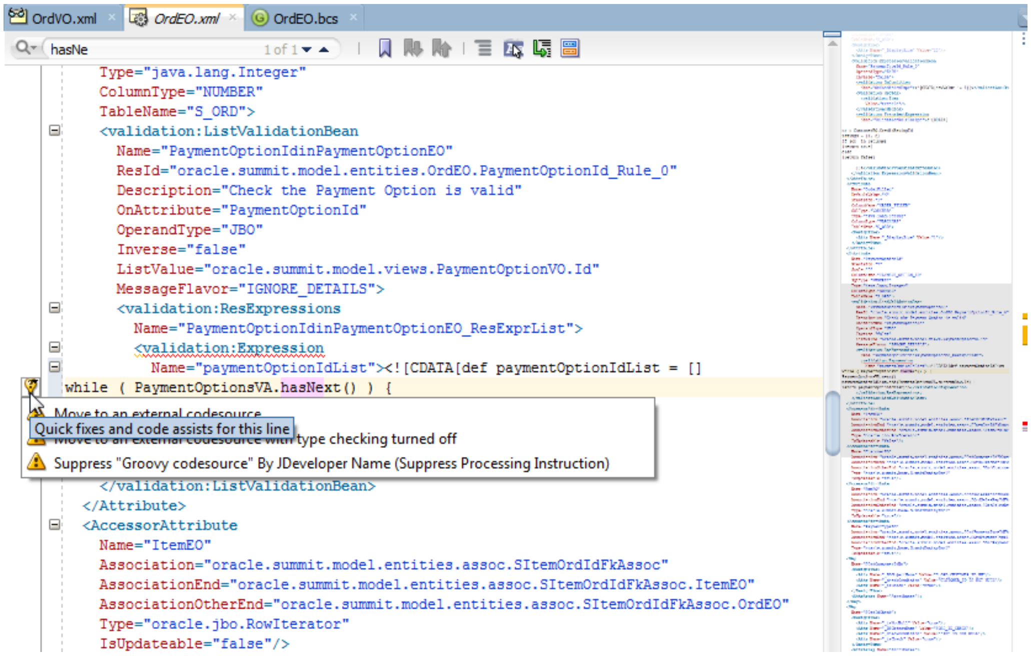
```
EmployeesInDept.sum("Sal + adf.object.getBenefitsValue(Job)")
```

What You May Need to Know About Migrating Groovy Snippets

When migrating to JDeveloper 12.2.1 or later from a previous release, the application allows you to migrate Groovy snippets to the `.bcs` file. In releases prior to JDeveloper 12.2.1, Groovy expressions defined on a business component, such as an entity object or view object, reside in the XML source of this business component. From JDeveloper 12.2.1 onwards, Groovy expressions defined on a business component aggregate into the `.bcs` file associated with this business component, if Groovy file generation is enabled. Type-checking if enabled for the current project or for all projects, is performed by the application in the `.bcs` file, upon compilation. See What You May Need to Know About Groovy Project Settings for details on how to enable/ disable Groovy type-checking and/or Groovy file generation.

Starting in JDeveloper 12.2.1, the settings that enable Groovy file generation into the `.bcs` file and type-checking of Groovy expressions in the `.bcs` file are enabled by default. So when you migrate an ADF application to JDeveloper 12.2.1 from a previous release, the application displays prompts in the gutter of the XML source of the business component. Click the prompts to display options that allow you to move Groovy expressions into the `.bcs` file. Figure 3-14 displays the prompts that JDeveloper displays for a Groovy snippet in the XML source (visible in the Source tab of the overview editor for the business component).

Figure 3-14   Code Assist Performs Groovy Snippet Migration



```
    Type="java.lang.Integer"
    ColumnType="NUMBER"
    TableName="S_ORD">
    <validation:ListValidationBean
      Name="PaymentOptionIdinPaymentOptionEO"
      ResId="oracle.summit.model.entities.OrdEO.PaymentOptionId_Rule_0"
      Description="Check the Payment Option is valid"
      OnAttribute="PaymentOptionId"
      OperandType="JBO"
      Inverse="false"
      ListValue="oracle.summit.model.views.PaymentOptionVO.Id"
      MessageFlavor="IGNORE_DETAILS">
      <validation:ResExpressions
        Name="PaymentOptionIdinPaymentOptionEO_ResExprList">
        <validation:Expression
          Name="paymentOptionIdList"><![CDATA[def paymentOptionIdList = []
while ( PaymentOptionsVA.hasNext() ) {
```

Move to an external codesource
Quick fixes and code assists for this line
Move to an external codesource with type checking turned off
Suppress "Groovy codesource" By JDeveloper Name (Suppress Processing Instruction)

```
    </validation:ListValidationBean>
  </Attribute>
  <AccessorAttribute
    Name="ItemEO"
    Association="oracle.summit.model.entities.assoc.SItemOrdIdFkAssoc"
    AssociationEnd="oracle.summit.model.entities.assoc.SItemOrdIdFkAssoc.ItemEO"
    AssociationOtherEnd="oracle.summit.model.entities.assoc.SItemOrdIdFkAssoc.OrdEO"
    Type="oracle.jbo.RowIterator"
    IsUpdateable="false"/>
```

- Select **Move to an external codesource** to move this Groovy snippet for which this prompt displays, into the `.bcs` file for this business component.
- Select **Move to an external codesource with type checking turned off** to move this Groovy snippet for which this prompt displays, into the `.bcs` file for this business component, with type-checking turned off.
- Select **Suppress "Groovy codesource" By JDeveloper Name (Suppress Processing Instruction)** to turn-off these prompts. Once turned-off, prompts do not display in the XML source for any other migrated business component having Groovy expressions.

After a Groovy snippet is migrated from the XML source to the `.bcs` file, the XML definition in the XML source displays a reference to the `.bcs` file where this migrated Groovy snippet resides. A migrated Groovy expression (as with any Groovy expression) has common information in the XML definition of the XML source that overlaps with the information contained in the annotations in the `.bcs` file. This common information helps identify a Groovy expression across the XML source and `.bcs` file. See What Happens When You Enter Expressions for details on the common information about a Groovy expression across the XML source and `.bcs` file contexts.

> **Note:**
>
> JDeveloper facilitates migration capabilities with audit rules that execute when you open the application. Audit rules are part of an audit profile — which is a collection of audit rules. A set of pre-enabled Groovy-specific audit rules achieve migration. To view these flags, from the main menu of JDeveloper navigate to **Tools** > select **Preferences**. In the Preferences dialog, select **Audit**. Click **Manage Profiles** to view the list of audit profiles. From this list, select and expand the **Application Development Framework (ADF)** audit profile to view the list of sub-rules. Select and expand **ADFbc Audit Rules** to view the list of sub-rules within this rule. Select and expand **Common** to view the Groovy-specific Audit rules that are pre-enabled, and which facilitate migration of Groovy snippets. See Understanding Audit Rules of *Developing Applications with Oracle JDeveloper*.

How to Create a Script Expression Class File

ADF Business Components allows you to create a script expression class file that you use to maintain common Groovy script methods that can be called by from anywhere within the application.

To create a script expression class file:

1. From the **File** menu, click **New**, select **From Gallery**, select **ADF Business Components** in the Business Tier section of the New Gallery dialog and select **Script Class** in the right-hand pane of this dialog.
2. In the Create Script Expression Class dialog, enter the class name of this script expression class in the **Class Name** field.
3. Specify the package that this script expression class resides in the **Package Name** field.

4. Optionally, specify the script expression classes that this script expression class extends in **Extends**.
5. Click **OK**.

The script expression class file is created and can be opened in JDeveloper from the package it resides.

4

Creating a Business Domain Layer Using Entity Objects

This chapter describes how to use ADF entity objects to create a reusable business layer of Java objects that describe the business domain in an Oracle ADF application. This chapter includes the following sections:

About Entity Objects

You use ADF entity objects to represent rows of a data source. ADF entity objects allow you to modify row attributes, encapsulate domain business logic and implement business policies and rules.

An entity object is the ADF business component that represents a row in the specified data source (generally a single database table, view, or synonym) and simplifies modifying its associated attributes. Importantly, it allows you to encapsulate domain business logic to ensure that your business policies and rules are consistently validated.

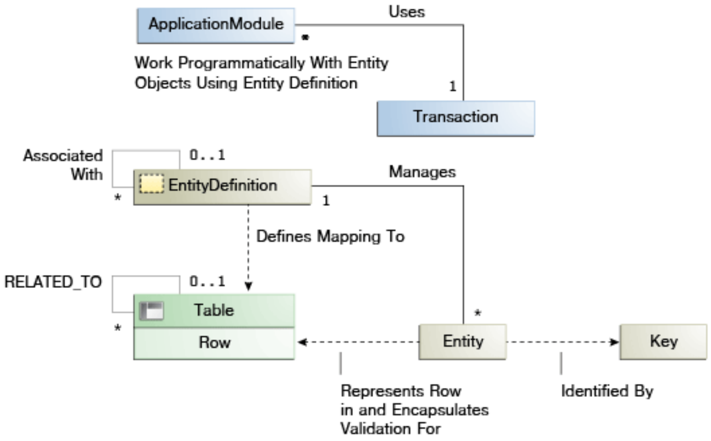
Entity Object Use Cases and Examples

Entity objects support numerous declarative business logic features to enforce the validity of your data. You will typically complement declarative validation with additional custom application logic and business rules to cleanly encapsulate a maximum amount of domain business logic into each entity object. Your associated set of entity objects forms a reusable business domain layer that you can exploit in multiple applications.

The key concepts of entity objects (as illustrated in Figure 4-1) are the following:

- You define an entity object by specifying the database table whose rows it will represent.
- You can create associations to reflect relationships between entity objects.
- At runtime, entity rows are managed by the entity definition object.
- Each entity row is identified by a related row key.
- You retrieve and modify entity rows in the context of an **application module** that provides the database transaction.

Figure 4-1 Entity Object Encapsulates Business Logic for a Table



Additional Functionality for Entity Objects

You may find it helpful to understand other **Oracle ADF** features before you start working with entity objects. Following are links to other functionality that may be of interest.

- For information about using declarative validation in entity objects, see Defining Validation and Business Rules Declaratively.
- For API documentation related to the `oracle.jbo` package, see the following Javadoc reference document:
  - *Java API Reference for Oracle ADF Model*

Creating Entity Objects and Associations

Create ADF entity objects and associations from existing tables. You can also create ADF entity objects from scratch and generate tables.

If you already have a database schema to work from, the simplest way to create entity objects and associations is to reverse-engineer them from existing tables. When needed, you can also create an entity object from scratch, and then generate a table for it later.

How to Create Multiple Entity Objects and Associations from Existing Tables

To create one or more entity objects, use the Business Components from Tables wizard, which is available from the New Gallery.

Before you begin:

It may be helpful to have an understanding of the options you have for creating entity objects. For more information, see Creating Entity Objects and Associations.

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Additional Functionality for Entity Objects.

To create one or more entity objects and associations from existing tables:

1. In the Applications window, right-click the project in which you want to create the entity objects and choose **New > From Gallery**.
2. In the New Gallery, expand **Business Tier**, select **ADF Business Components** and then **Business Components from Tables**, and click **OK**.

   If this is the first component you're creating in the project, the Initialize Business Components Project dialog appears to allow you to select a database connection.
3. In the Initialize Business Components Project dialog, select the database connection or click the **Create a new database connection** icon to create a connection. Click **OK**.
4. On the Entity Objects page, do the following to create the entity objects:
   - Enter the name of the package in which all of the entity objects will be created.
   - Select the tables from the **Available** list for which you want to create entity objects.

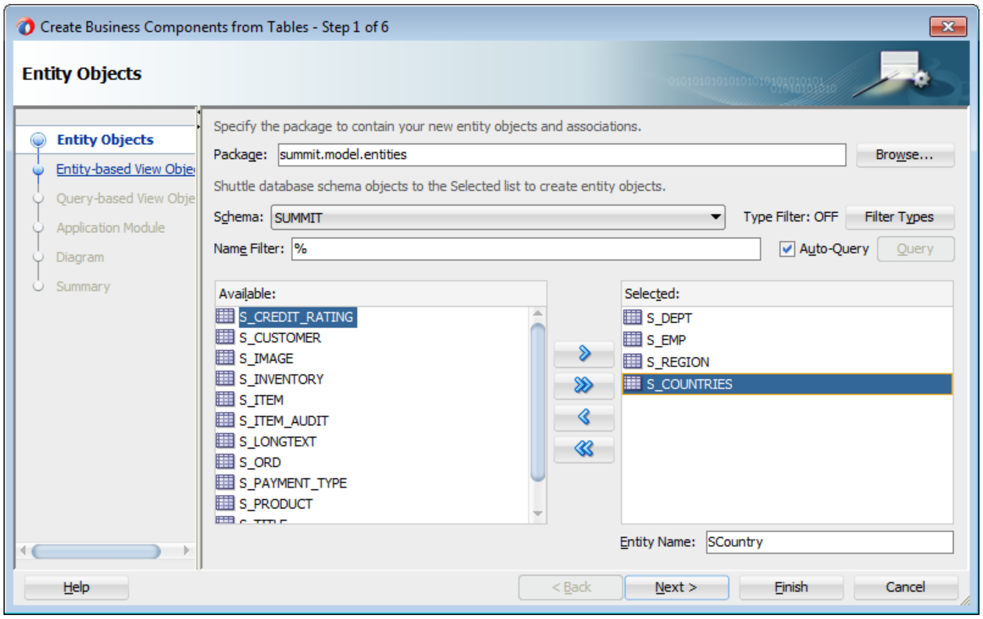
     If the **Auto-Query** checkbox is selected, then the list of available tables appears immediately. In the **Name Filter** field, you can optionally enter a full or partial table name to filter the available tables list in real time. As an alternative to the auto-query feature, click the **Query** button to retrieve the list based on an optional table name filter. When no name filter is entered, JDeveloper retrieves all table objects for the chosen schema.
   - Click **Filter Types** if you want to see only a subset of the database objects available. You can filter out tables, views, or synonyms.

Once you have selected a table from the **Available** list, the proposed entity object name for that table appears in the **Selected** list with the related table name in parenthesis.

- Select an entity object name in the **Selected** list and use the **Entity Name** field to change the default entity object name.

> **Best Practice:**
>
> Because each entity object instance represents a *single* row in a particular table, name the entity objects with a singular noun (like Address, Order, and Person), instead of their plural counterparts. Figure 4-2 shows what the wizard page looks like after selecting the `S_COUNTRIES` table in the Summit ADF schema, setting a package name of `summit.model.entities`, and renaming the entity object in the singular.

Figure 4-2 Create Business Components from Tables Wizard, Entity Objects Page



Create Business Components from Tables - Step 1 of 6

Entity Objects

Specify the package to contain your new entity objects and associations.

Package: summit.model.entities

Shuttle database schema objects to the Selected list to create entity objects.

Schema: SUMMIT

Type Filter: OFF

Name Filter: %

☑ Auto-Query

Available:
S_CREDIT_RATING
S_CUSTOMER
S_IMAGE
S_INVENTORY
S_ITEM
S_ITEM_AUDIT
S_LONGTEXT
S_ORD
S_PAYMENT_TYPE
S_PRODUCT
S_TITLE

Selected:
S_DEPT
S_EMP
S_REGION
S_COUNTRIES

Entity Name: SCountry

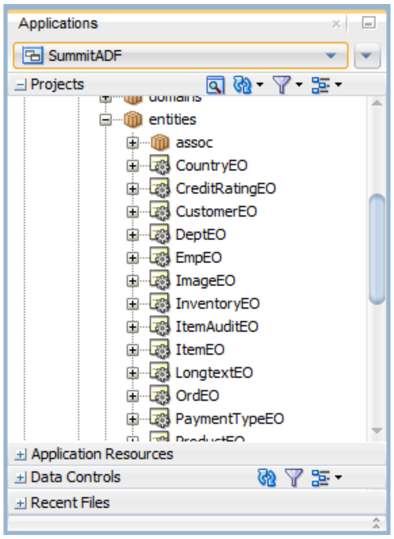
5. When you are satisfied with the selected table objects and their corresponding entity object names, click **Finish**.

The Applications window displays the entity objects in the package you specified.

Best Practice:

After you create associations, move all of your associations to a separate package so that you can view and manage them separately from the entity objects. In Figure 4-3, the associations have been moved to a subpackage (`assoc`) and do not appear in the `entities` package in the Applications window. For more information, see How to Rename and Move Associations to a Different Package.

Figure 4-3 New Entity Objects in Applications window



How to Create Single Entity Objects Using the Create Entity Wizard

To create a single entity object, you can use the Create Entity Object wizard, which is available in the New Gallery.

Note:

Associations are not generated when you use the Create Entity Object wizard. However, the Business Components from Tables wizard does generate associations. If you use the Create Entity Object wizard to create entity objects, you will need to create the corresponding associations manually.

Before you begin:

It may be helpful to have an understanding of the options you have for creating entity objects. For more information, see Creating Entity Objects and Associations.

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Additional Functionality for Entity Objects.

To create a single entity object:

1. In the Applications window, right-click the project in which you want to create the entity object and choose **New > From Gallery**.
2. In the New Gallery, expand **Business Tier**, select **ADF Business Components** and then **Entity Object**, and click **OK**.

   If this is the first component you're creating in the project, the Initialize Business Components Project dialog appears to allow you to select a database connection.
3. In the Initialize Business Components Project dialog, select the database connection or choose **New** to create a connection. Click **OK**.
4. On the Name page, do the following to create the entity object:
   - Enter a name for the entity object.
   - Enter the package name in which the entity object will be created.
   - Click **Browse** (next to the **Schema Object** field) to select the table for which you want to create the entity object.

     Or, if you plan to create the table later, you can enter a name of a table that does not exist.
5. If you manually entered a table name in the **Schema Object** field, you will need to define each attribute on the Attributes page of the wizard. Click **Next**.

   You can create the table manually or generate it, as described in How to Create Database Tables from Entity Objects.
6. When you are satisfied with the table object and its corresponding entity object name, click **Finish**.

What Happens When You Create Entity Objects and Associations from Existing Tables

When you create an entity object from an existing table, first JDeveloper interrogates the data dictionary to infer the following information:

- The Java-friendly entity attribute names from the names of the table's columns (for example, `USER_ID` -> `UserId`)
- The SQL and Java data types of each attribute based on those of the underlying column
- The length and precision of each attribute
- The primary and unique key attributes
- The mandatory flag on attributes, based on `NOT NULL` constraints
- The relationships between the new entity object and other entities based on foreign key constraints

Note:

Since an entity object represents a database row, it seems natural to call it an **entity row**. Alternatively, since at runtime the entity row is an instance of a Java object that encapsulates business logic for that database row, the more object-oriented term **entity instance** is also appropriate. Therefore, these two terms are interchangeable.

JDeveloper then creates the XML document file that represents its declarative settings and saves it in the directory that corresponds to the name of its package. For example, when an entity named `Order` appears in the `genericbcmodel.entities` package, JDeveloper will create the XML file `genericbcmodel/entities/Order.xml` under the project's source path. This XML file contains the name of the table, the names and data types of each entity attribute, and the column name for each attribute.

You can inspect the XML description for the entity object by opening the object in the overview editor and clicking the **Source** tab.

Note:

If your IDE-level Business Components Java generation preferences so indicate, the wizard may also create an optional custom entity object class (for example, `OrderImpl.java`).

Foreign Key Associations Generated When Entity Objects Are Derived from Tables

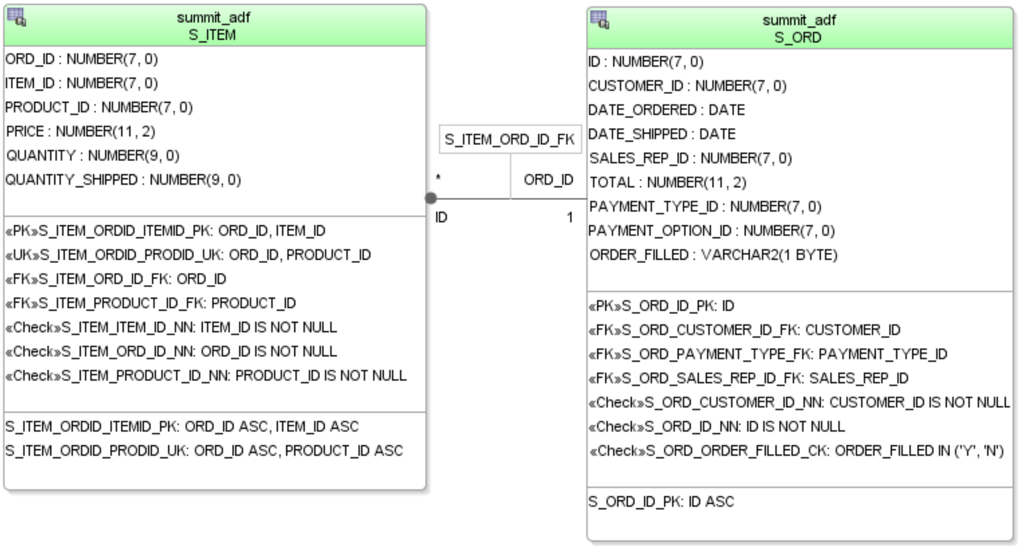
In addition to the entity objects, the Business Components from Tables wizard also generates named association components that capture information about the relationships between entity objects. For example, the database diagram in Figure 4-4 shows that JDeveloper derives default association names like `SItemOrdIdFkAssoc` by converting the foreign key constraint names to a Java-friendly name and adding the `Assoc` suffix. For each association created, JDeveloper creates an appropriate XML document file and saves it in the directory that corresponds to the name of its package.

Note:

Associations are generated when you use the Business Components from Tables wizard. However, the Create Entity Object wizard does not generate associations. If you use the Create Entity Object wizard to create entity objects, you will need to create the corresponding associations manually.

By default the associations reverse-engineered from foreign keys are created in the same package as the entities. For example, for the association `SItemOrdIdFkAssoc` with entities in the `summit.model.entities` package, JDeveloper creates the association XML file named `./summit/model/entities/SItemOrdIdFkAssoc.xml`.

Figure 4-4 S_ITEM and S_ORD Tables Related by Foreign Key


summit_adf
S_ITEM
ORD_ID : NUMBER(7, 0)
ITEM_ID : NUMBER(7, 0)
PRODUCT_ID : NUMBER(7, 0)
PRICE : NUMBER(11, 2)
QUANTITY : NUMBER(9, 0)
QUANTITY_SHIPPED : NUMBER(9, 0)
«PK»S_ITEM_ORDID_ITEMID_PK: ORD_ID, ITEM_ID
«UK»S_ITEM_ORDID_PRODID_UK: ORD_ID, PRODUCT_ID
«FK»S_ITEM_ORD_ID_FK: ORD_ID
«FK»S_ITEM_PRODUCT_ID_FK: PRODUCT_ID
«Check»S_ITEM_ITEM_ID_NN: ITEM_ID IS NOT NULL
«Check»S_ITEM_ORD_ID_NN: ORD_ID IS NOT NULL
«Check»S_ITEM_PRODUCT_ID_NN: PRODUCT_ID IS NOT NULL
S_ITEM_ORDID_ITEMID_PK: ORD_ID ASC, ITEM_ID ASC
S_ITEM_ORDID_PRODID_UK: ORD_ID ASC, PRODUCT_ID ASC
S_ITEM_ORD_ID_FK
*
ORD_ID
ID
1
summit_adf
S_ORD
ID : NUMBER(7, 0)
CUSTOMER_ID : NUMBER(7, 0)
DATE_ORDERED : DATE
DATE_SHIPPED : DATE
SALES_REP_ID : NUMBER(7, 0)
TOTAL : NUMBER(11, 2)
PAYMENT_TYPE_ID : NUMBER(7, 0)
PAYMENT_OPTION_ID : NUMBER(7, 0)
ORDER_FILLED : VARCHAR2(1 BYTE)
«PK»S_ORD_ID_PK: ID
«FK»S_ORD_CUSTOMER_ID_FK: CUSTOMER_ID
«FK»S_ORD_PAYMENT_TYPE_FK: PAYMENT_TYPE_ID
«FK»S_ORD_SALES_REP_ID_FK: SALES_REP_ID
«Check»S_ORD_CUSTOMER_ID_NN: CUSTOMER_ID IS NOT NULL
«Check»S_ORD_ID_NN: ID IS NOT NULL
«Check»S_ORD_ORDER_FILLED_CK: ORDER_FILLED IN ('Y', 'N')
S_ORD_ID_PK: ID ASC

Entity Object Key Generated When a Table Has No Primary Key

If a table has no primary key constraint, then JDeveloper cannot infer the primary key for the entity object. Because every entity object must have at least one attribute marked as a primary key, the wizard marks all columns as part of the primary key for the entity. If appropriate, you can edit the entity object later to mark a different attribute as a primary key and remove the setting from other attributes. When you use the Create Entity Object wizard and you have not set any other attribute as primary key, you will be prompted to use all columns as the primary key.

What Happens When You Create an Entity Object for a Synonym or View

When you create an entity object using the Business Components from Tables wizard or the Create Entity Object wizard, the object can represent an underlying table, synonym, or view. The framework can infer the primary key and related associations for a table or synonym by inspecting database primary and foreign key constraints in the data dictionary.

However, when your selected schema object is a database view, then neither the primary key nor associations can be inferred since database views do not have database constraints. In this case, if you use the Business Components from Tables wizard, the primary key defaults to `RowID`. If you use the Create Entity Object wizard, you'll need to specify the primary key manually by marking at least one of its attributes as a primary key. For more information, see RowID Generated When a Table Has No Primary Key.

When your selected schema object is a synonym, there are two possible outcomes. If the synonym is a synonym for a table, then the wizard and editor behave as if you had specified a table. If instead the synonym refers to a database view, then they behave as if you had specified a view.

How to Edit an Existing Entity Object or Association

After you've created a new entity object or association, you can edit any of its settings in the overview editor. To launch the editor, choose **Open** from the context menu for the entity object or association in the Applications window or double-click the object. By clicking the different tabs of the editor, you can adjust the settings that define the object and govern its runtime behavior.

How to Create Database Tables from Entity Objects

To create database tables based on entity objects, right-click the package in the Applications window that contains the entity objects and choose **Create Database Objects** from the context menu. A dialog appears to let you select the entities whose tables you'd like to create. This tool can be used to generate a table for an entity object you created from scratch, or to drop and re-create an existing table.

> **Caution:**
>
> This feature performs its operations directly against the database and will drop existing tables. It does *not* generate a script to run later. A dialog appears to confirm that you want to do this before proceeding. For entities based on existing tables, use with caution.

In the overview editor for an association, the **Use Database Key Constraints** checkbox on the Association Properties page controls whether the related foreign key constraint will be generated when creating the tables for entity objects. Selecting this option does not have any runtime implications.

How to Synchronize an Entity with Changes to Its Database Table

Inevitably you (or your DBA) might alter a table for which you've already created an entity object. Your existing entity will not be disturbed by the presence of additional attributes in its underlying table; however, if you want to access the new column in the table in your Java EE application, you'll need to synchronize the entity object with the database table.

For example, suppose you had done the following at the SQL*Plus command prompt to add a new `SECURITY_QUESTION` column to the `PERSONS` table:

```
ALTER TABLE PERSONS ADD (security_question VARCHAR2(60));
```

Then you can use the synchronization feature to add the new column as an attribute on the entity object.

Before you begin:

It may be helpful to have an understanding of the options you have for creating entity objects. For more information, see Creating Entity Objects and Associations.

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Additional Functionality for Entity Objects.

To synchronize an entity with changes to its database table:

1. In the Applications window, right-click the entity object you want to synchronize and choose **Synchronize with Database**.

   The Synchronize with Database dialog shows the list of the actions that can be taken to synchronize the business logic tier with the database.

2. Select the action you want to take:

   - Select one or more actions from the list, and click **Synchronize** to synchronize the selected items.
   - Click **Synchronize All** to perform all actions in the list.
   - Click **Write to File** to save the action list to a text file. This feature helps you keep track of the changes you make.

3. Click **OK**.

Removing an Attribute Associated with a Dropped Column

The synchronize feature does not handle dropped columns. When a column is dropped from the underlying database after an entity object has been created, you can delete the corresponding attribute from the entity object. If the attribute is used in other parts of your application, you must remove those usages as well.

Before you begin:

It may be helpful to have an understanding of the options you have for creating entity objects. For more information, see Creating Entity Objects and Associations.

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Additional Functionality for Entity Objects.

To remove an entity attribute:

1. In the Applications window, double-click the entity object that contains the attribute you want to remove.
2. In the overview editor, click the **Attributes** navigation tab.
3. On the Attributes page, right-click the attribute and choose **Delete**.

   If there are other usages, the Confirm Delete dialog displays the message "Usages were found."

4. If usages were found, click **Show Usages**.

   The dialog shows all usages of the attribute, which you can click to display the usage in the source editor.

5. Click **Preview** to display usages of the attribute in the Log window, and then work through the list to delete all usages of the entity attribute.

Addressing a Data Type Change in the Underlying Table

The synchronize feature does not handle changed data types. For a data type change in the underlying table (for example, precision increased), you must locate all usages of the attribute and manually make changes, as necessary.

Before you begin:

It may be helpful to have an understanding of the options you have for creating entity objects. For more information, see Creating Entity Objects and Associations.

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Additional Functionality for Entity Objects.

To locate all usages of an entity attribute:

1. In the Applications window, double-click the entity object.
2. In the overview editor, click the **Attributes** navigation tab.
3. On the Attributes page, right-click the attribute and choose **Find Usages**.

   If there are other usages, they are displayed in the Log window.

How to Store Data Pertaining to a Specific Point in Time

Effective dated tables are used to provide a view into the data set pertaining to a specific point in time. Effective dated tables are widely used in applications like HRMS and Payroll to answer queries like:

- What was the tax rate for an employee on August 31st, 2005?
- What are the employee's benefits as of October 2004?

In either case, the employee's data may have changed to a different value since then.

The primary difference between the effective dated entity type and the dated entity type is that the dated entity does not cause row splits during update and delete.

> **Note:**
>
> Do not use effective dating on a parent entity object in a **master-detail relationship**. Because child objects can raise events on parent objects, and effective dated entity objects cause row splits during update and delete operations, it is possible for the child object to raise an event on the wrong parent object if both the parent and child are updated during the same transaction.

When you create an effective dated entity object, you identify the entity as effective dated and specify the attributes of the entity that represent the start and end dates. The start date and end date attributes must be of the Date type.

Additionally, you can specify an attribute that represents the sequence for the effective dated entity and an attribute that represents a flag for the sequence. These attributes allow for tracking of multiple changes in a single day.

Before you begin:

It may be helpful to have an understanding of the options you have for creating entity objects. For more information, see Creating Entity Objects and Associations.

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Additional Functionality for Entity Objects.

To create an effective dated entity object

1. In the Applications window, double-click the entity object on which you want enable effective dating.
2. In the Properties window, expand the **Type** category.

   If necessary, choose **Properties** from the **Window** menu to display the Properties window.

   If the **Type** category is not displayed in the Properties window, click the **General** tab in the overview editor to set the proper focus.
3. From the property menu for **Effective Date Type**, choose **Edit**.

   To display the property menu, click the down arrow next to the property field.
4. In the Edit Property dialog, specify the following settings:
   - For **Effective Date Type**, select **EffectiveDated**.
   - For **Start Date Attribute**, select the attribute that corresponds to the start date.
   - For **End Date Attribute**, select the attribute that corresponds to the end date.
5. You can optionally specify attributes that allow for tracking of multiple changes in a single day.
   - For **Effective Date Sequence**, select the attribute that stores the sequence of changes.
   - For **Effective Date Sequence Flag**, select the attribute that stores a flag indicating the most recent change in the sequence.

   Without specifying the **Effective Date Sequence** and **Effective Date Sequence Flag** attributes, the default granularity of effective dating is one day. For this reason, multiple changes in a single day are not allowed. An attempt to update the entity a second time in a single day will result in an exception being thrown. After these two attributes are specified, the framework inserts and updates their values as necessary to track multiple changes in a single day.
6. Click **OK**.

> **Note:**
>
> You can also identify the start and end date attributes using the Properties window for the appropriate attributes. To do so, select the appropriate attribute in the overview editor and set the **Start Date** or **End Date** property to **true** in the Properties window.

What Happens When You Create Effective Dated Entity Objects

When you create an effective dated entity object, JDeveloper creates a transient attribute called SysEffectiveDate to store the effective date for the row. Typically the Insert, Update, and Delete operations modify the transient attribute while the **ADF Business Components** framework decides the appropriate values for the effective start date and the effective end date.

The following example shows some sample XML entries that are generated when you create an effective dated entity. For more information about working with effective dated objects, see Limiting View Object Rows Using Effective Date Ranges.

```
// In the effective dated entity
<Entity
  ...
  EffectiveDateType="EffectiveDated">

// In the attribute identified as the start date
  <Attribute
    ...
    IsEffectiveStartDate="true">

// In the attribute identified as the end date
  <Attribute
    ...
    IsEffectiveEndDate="true">

// The SysEffectiveDate transient attribute
  <Attribute
    Name="SysEffectiveDate"
    IsQueriable="false"
    IsPersistent="false"
    ColumnName="$none$"
    Type="oracle.jbo.domain.Date"
    ColumnType="$none$"
    SQLType="DATE"/>
```

What You May Need to Know About Creating Entities from Tables

The Business Components from Tables wizard makes it easy to quickly generate many business components at the same time. In practice, this does not mean that you should use it to immediately create entity objects for every table in your database schema just because it is possible to do so. If your application requires all of the tables, then that strategy might be appropriate. But because you can use the wizard whenever needed, you should create the entity objects for the tables that you know will be involved in the application.

Defining Nested Application Modules, describes a use case-driven design approach for your business services that can assist you in understanding which entity objects are required to support your application's business logic needs. You can always add more entity objects later as necessary.

Creating and Configuring Associations

Create entity associations manually if foreign key constraints don’t exist so that JDeveloper is able to infer associations between ADF entity objects.

If your database tables have no foreign key constraints defined, JDeveloper won't be able to infer the associations between the entity objects that you create. Since several ADF Business Components runtime features depend on the presence of **entity associations**, create them manually if the foreign key constraints don't exist.

How to Create an Association

To create an association, use the Create New Association wizard, which is available in the New Gallery.

Before you begin:

It may be helpful to have an understanding of why you create associations. For more information, see Creating and Configuring Associations.

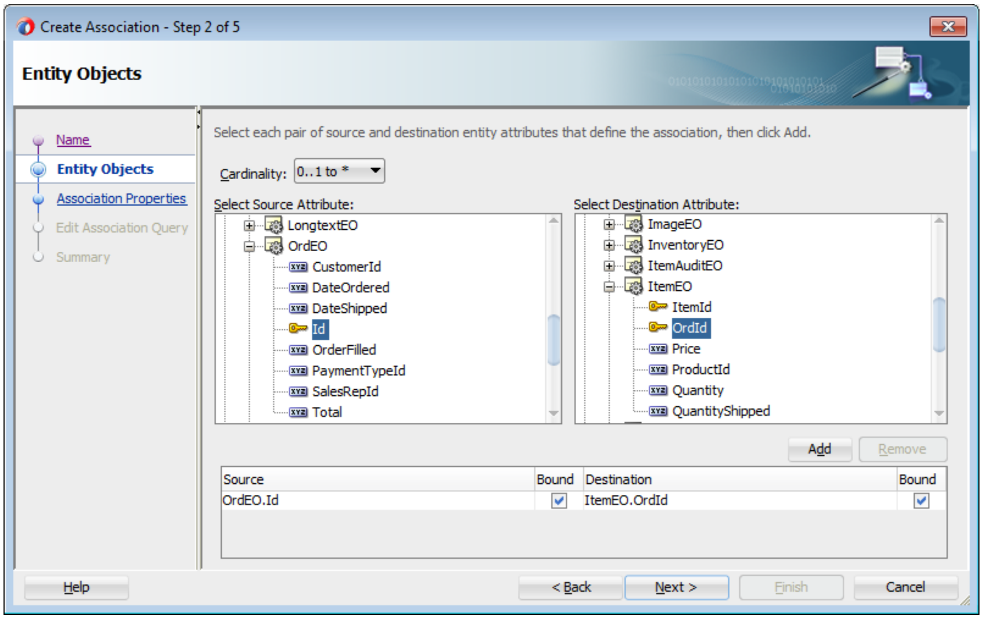
You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Additional Functionality for Entity Objects.

To create an association:

1. In the Applications window, right-click the project in which you want to create the association and choose **New > From Gallery**.
2. In the New Gallery, expand **Business Tier**, select **ADF Business Components** and then **Association**, and click **OK**.
3. In the Create Association wizard, on the Name page, do the following:
   - Enter the package name in which the association will be created.
   - Enter the name of the association component.
   - Click **Next**.
4. On the Entity Objects page, select the source and destination entity attributes:
   - Select a source attribute from one of the entity objects that is involved in the association to act as the master.
   - Select a corresponding destination attribute from the other entity object involved in the association.

   For example, Figure 4-5 shows the selected `Id` attribute from the `OrdEO` entity object as the source entity attribute. Because the `ItemEO` rows contain an order ID that relates them to a specific `OrdEO` row, you would select this `OrdId` foreign key attribute in the `ItemEO` entity object as the destination attribute.

Figure 4-5 Create Association Wizard, Attribute Pairs That Relate Two Entity Objects Defined


Create Association - Step 2 of 5

Entity Objects

Select each pair of source and destination entity attributes that define the association, then click Add.

Cardinality: 0..1 to *

Select Source Attribute:

- LongtextEO
- OrdEO
  - CustomerId
  - DateOrdered
  - DateShipped
  - Id
  - OrderFilled
  - PaymentTypeId
  - SalesRepId
  - Total

Select Destination Attribute:

- ImageEO
- InventoryEO
- ItemAuditEO
- ItemEO
  - ItemId
  - OrdId
  - Price
  - ProductId
  - Quantity
  - QuantityShipped

Source	Bound	Destination	Bound
OrdEO.Id	☑	ItemEO.OrdId	☑

5. Click **Add** to add the matching attribute pair to the table of source and destination attribute pairs below.

   By default, the **Bound** checkbox is selected for both the source and destination attribute. This checkbox allows you to specify whether or not the value will be bound into the association SQL statement that is created internally when navigating from source entity to target entity or from target entity to source entity (depending on which side you select).

   Typically, you would deselect the checkbox for an attribute in the relationship that is a transient entity attribute whose value is a constant and therefore should not participate in the association SQL statement to retrieve the entity.

6. If the association requires multiple attribute pairs to define it, you can repeat the preceding steps to add additional source/target attribute pairs.

7. Finally, ensure that the **Cardinality** dropdown correctly reflects the cardinality of the association. The default is a one-to-many relationship. Click **Next**.

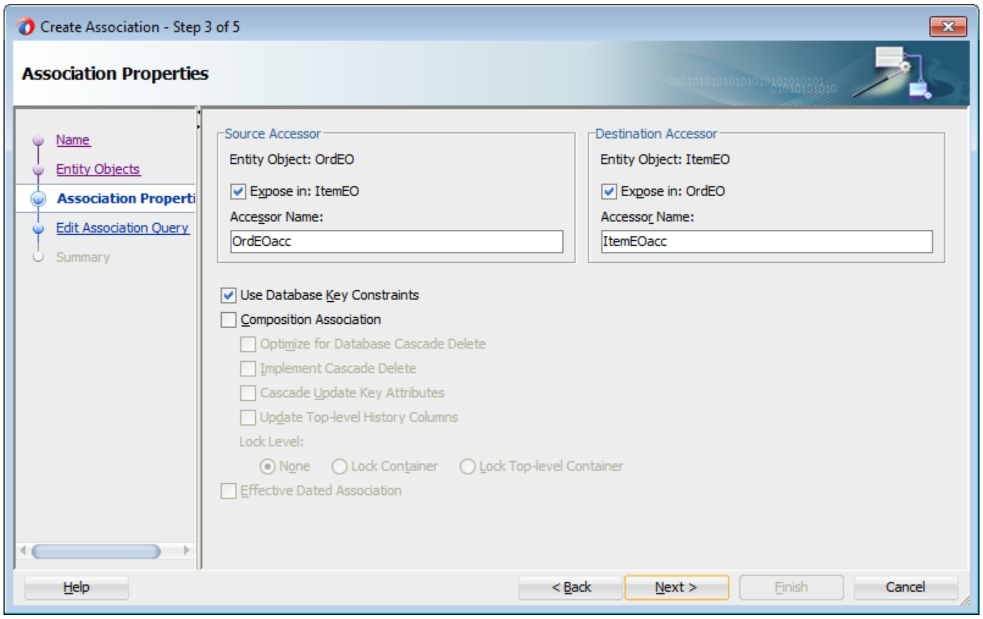
   For example, since the relationship between an `OrdEO` row and its related `ItemEO` rows is one-to-many, you can leave the default setting.

8. On the Association SQL page, you can preview the association SQL predicate that will be used at runtime to access the related destination entity objects for a given instance of the source entity object.

9. On the Association Properties page, disable the **Expose Accessor** checkbox on either the **Source** or the **Destination** entity object when you want to create an association that represents a one-way relationship. The default, bidirectional navigation is more convenient for writing business validation logic, so in practice, you typically leave these default checkbox settings.

   For example, Figure 4-6 shows an association that represents a bidirectional relationship, permitting either entity object to access the related entity row(s) on the other side when needed. In this example, this means that if you are working with an instance of an `OrdEO` entity object, you can easily access the collection of its related `OrderItemEO` rows. With any instance of a `OrderItemEO` entity object, you can also easily access the `Order` to which it belongs.

Figure 4-6   Association Properties Control Runtime Behavior



Source Accessor
Entity Object: OrdEO
☑ Expose in: ItemEO
Accessor Name: OrdEOacc

Destination Accessor
Entity Object: ItemEO
☑ Expose in: OrdEO
Accessor Name: ItemEOacc

☑ Use Database Key Constraints
□ Composition Association
□ Optimize for Database Cascade Delete
□ Implement Cascade Delete
□ Cascade Update Key Attributes
□ Update Top-level History Columns
Lock Level:
● None ○ Lock Container ○ Lock Top-level Container
□ Effective Dated Association

10. When you are satisfied with the association definition, click **Finish**.

What Happens When You Create an Association

When you create an association, JDeveloper creates an appropriate XML document file and saves it in the directory that corresponds to the name of its package. For example, if you created an association named `SItemOrderIdFkAssoc` in the `oracle.summit.model.entities.assoc` subpackage, then the association XML file would be created in the `./oracle/summit/model/entities/assoc` directory with the name `SItemOrderIdFkAssoc.xml`. At runtime, the entity object uses the association information to automate working with related sets of entities.

How to Change Entity Association Accessor Names

You should consider the default settings for the accessor names on the Association Properties page and decide whether changing the names to something more intuitive is appropriate. The default settings define the names of the accessor attributes you will use at runtime to programmatically access the entities on the other side of the relationship. By default, the accessor names will be the names of the entity object on the other side. Since the accessor names on an entity must be unique among entity object attributes and other accessors, if one entity is related to another entity in *multiple* ways, then the default accessor names are modified with a numeric suffix to make the name unique.

In an existing association, you can rename the accessor using the Association Properties dialog.

Before you begin:

It may be helpful to have an understanding of why you create associations. For more information, see Creating and Configuring Associations.

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Additional Functionality for Entity Objects.

To rename the entity accessor in an association:

1. In the Applications window, double-click the association that contains the entity accessor you want to rename.
2. In the overview editor, click the **Relationships** navigation tab.
3. On the Relationships page, expand the **Accessors** section and click the **Edit** icon.

   The Association Properties dialog displays the current settings for the association's accessors.
4. In the Association Properties dialog, modify the name as necessary, and click **OK**.

How to Rename and Move Associations to a Different Package

Since associations are a component that you typically configure at the outset of your project and don't change frequently thereafter, you might want to move the associations to a different package so that your entity objects are easier to see.

Both renaming components and moving them to a different package is straightforward using JDeveloper's refactoring functionality. However, it is not advisable to rename or move ADF Business Components objects manually, as missed references can break your application. For more information about JDeveloper's refactoring functionality, see Refactoring a Fusion Web Application .

Before you begin:

It may be helpful to have an understanding of why you create associations. For more information, see Creating and Configuring Associations.

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Additional Functionality for Entity Objects.

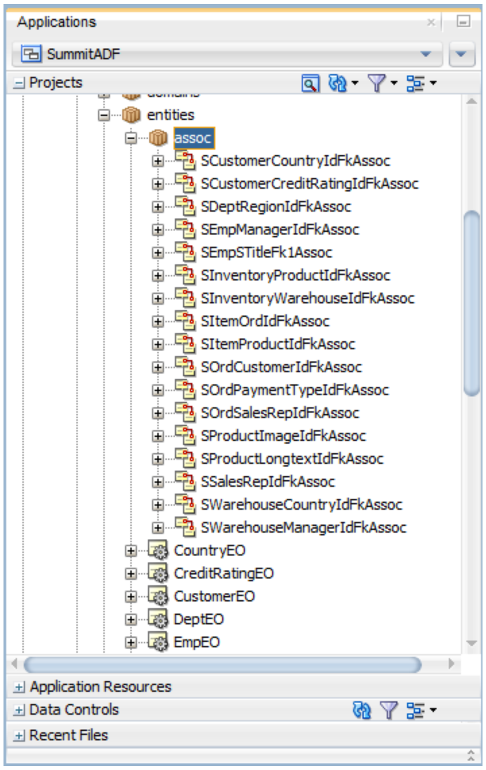
To move a set of business components to a different package:

1. In the Applications window, select the components you want to move.
2. Right-click one of the selected components, and choose **Refactor > Move**.
3. In the Move Business Components dialog, enter the name of the package to move the component(s) to, or click **Browse** to navigate to and select the package.
4. Click **OK**.

If you just want to rename a component, right-click the component you want to rename, and choose **Refactor > Rename**. When you refactor ADF Business Components, JDeveloper moves the XML and Java files related to the components, and updates any other components that might reference them.

Figure 4-7 shows what the Applications window would look like after renaming all of the associations and moving them to the `oracle.summit.model.entities.assoc` subpackage. While you can refactor the associations into any package name you choose, picking a subpackage keeps them logically related to the entities, and allows you to collapse the package of associations to better manage which files display in the Applications window.

Figure 4-7 Applications Window After Association Refactoring



What You May Need to Know About Using a Custom View Object in an Association

You can associate a custom **view object** with the source end or destination end (or both) of an entity association.

When you traverse entity associations in your code, if the entities are not already in the cache, then the ADF Business Components framework performs a query to bring the entity (or entities) into the cache. By default, the query performed to bring an entity into the cache is the find-by-primary-key query that selects values for all persistent entity attributes from the underlying table. If the application performs a lot of programmatic entity association traversal, you could find that retrieving all of the attributes might be heavy-handed for your use cases.

Entity associations support the ability to associate a custom, entity-based view object with the source entity or destination entity in the association, or both. The primary entity usage of the entity-based view object you supply must match the entity type of the association end for which you use it.

Using a custom view object can be useful because the custom view object's query can include fewer columns and it can include an `ORDER BY` clause. This allows you to control how much data is retrieved when an entity is brought into the cache due to association traversal, as well as the order in which any collections of related entities will appear.

For more information about creating a custom view object, see How to Create an Entity-Based Programmatic View Object.

What You May Need to Know About Composition Associations

An association represents a relationship between entities, such as a `Customer` referenced by an `Order` or an `Item` contained in an `Order`. When you create associations, it is useful to know about the kinds of relationships you can represent, and the various options.

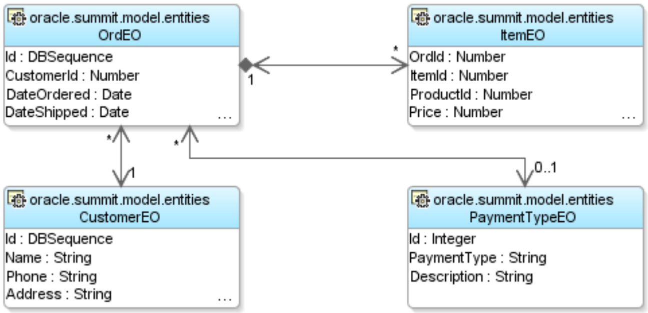
Associations between entity objects can represent two styles of relationships depending on whether the source entity:

- *References* the destination entity
- *Contains* the destination entity as a logical, nested part

Figure 4-8 depicts an application business layer that represents both styles of relationships. For example, an `OrdEO` entry references a `CustomerEO`. This relationship represents the first kind of association, reflecting that a `CustomerEO` or an `OrdEO` entity object can exist independent from each other. In addition, the removal of an `Order` does not imply the cascade removal of the `Customer` to which it was referring.

In contrast, the relationship between an `OrdEO` and its collection of related `ItemEO` details is stronger than a simple reference. The `ItemEO` entries comprise a logical part of the overall `OrdEO`. In other words, an `OrdEO` is composed of `ItemEO` entries. It does not make sense for an `ItemEO` entity row to exist independently from an `OrdEO`, and when an `OrdEO` is removed — assuming the removal is allowed — all of its composed parts should be removed as well. This kind of logical containership represents the second kind of association, called a **composition**. The UML diagram in Figure 4-8 illustrates the stronger composition relationship using the solid diamond shape on the side of the association which composes the other side of the association.

Figure 4-8 OrdEO Composed of ItemEO Entries and References Both CustomerEO and PaymentTypeEO



The Business Components from Tables Wizard creates composition associations by default for any foreign keys that have the `ON DELETE CASCADE` option. You can use the Create Association wizard or the overview editor for the association to indicate that an association is a composition association. Select the **Composition Association** checkbox on either the Association Properties page of the Create Association wizard or the Relationships page of the overview editor.

> **Note:**
>
> A composition association cannot be based on a transient attribute.

An entity object offers additional runtime behavior in the presence of a composition. For the settings that control the behavior, see How to Configure Composition Behavior.

Creating a Diagram of Entity Objects for Your Business Layer

You can use JDeveloper to create a UML diagram of business services based on ADF Business Components. The UML diagram serves as a visualization, visual navigation, and editing tool.

Since your layer of business domain objects represents a key reusable asset for your team, it is often convenient to visualize the business domain layer using a UML model. JDeveloper supports easily creating a diagram for your business domain layer that you and your colleagues can use for reference.

The UML diagram of business components is not just a static picture that reflects the point in time when you dropped the entity objects onto the diagram. Rather, it is a UML-based rendering of the current component definitions, that will always reflect the current state of affairs. What's more, the UML diagram is both a visualization aid and a visual navigation and editing tool. To open the overview editor for any entity object in a diagram, right-click the desired object and choose **Open**. You can also perform some entity object editing tasks directly on the diagram, like renaming entity objects and attributes, and adding or removing attributes. However, changes to the entity objects in a business components diagram have no impact on the underlying database objects.

How to Show Entity Objects in a Business Components Diagram

To create a diagram of your entity objects, you can use the Create Business Components Diagram dialog, which is available in the New Gallery.

Before you begin:

It may be helpful to have an understanding of how entity diagrams are used in the application. For more information, see Creating a Diagram of Entity Objects for Your Business Layer.

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Additional Functionality for Entity Objects.

To create a business components diagram that models existing entity objects:

1. In the Applications window, right-click the project in which you want to create the entity diagram and choose **New > From Gallery**.

2. In the New Gallery, expand **Business Tier**, select **ADF Business Components** and then **Business Components Diagram**, and click **OK**.
3. In the dialog, do the following to create the diagram:
   - Enter a name for the diagram, for example `Business Domain Objects`.
   - Enter the package name in which the diagram will be created. For example, you might create it in a subpackage like `myproject.model.design`.
4. Click **OK**.
5. To add existing entity objects to the diagram, select them in the Applications window and drop them onto the diagram surface.

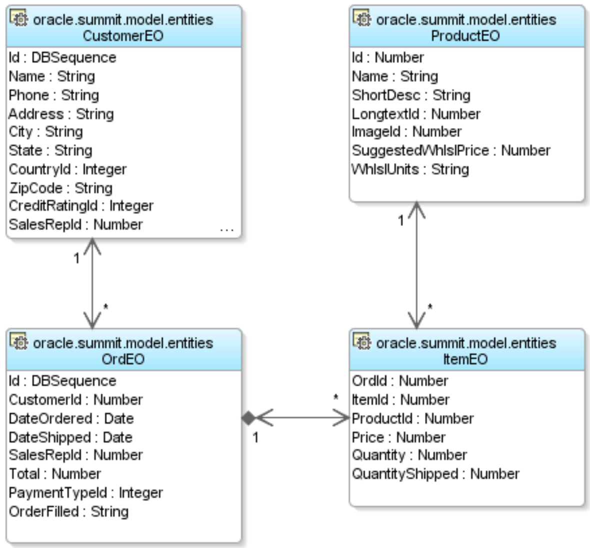
After you have created the diagram you can use the Properties window to adjust visual properties of the diagram. For example you can:

- Hide or show the package name
- Change the font
- Toggle the grid and page breaks on or off
- Display association names that may otherwise be ambiguous

You can also create an image of the diagram in `PNG`, `JPG`, `SVG`, or compressed `SVG` format, by choosing **Publish Diagram** from the context menu on the diagram surface.

Figure 4-9 shows a sample diagram that models various entity objects from the business domain layer.

Figure 4-9 UML Diagram of Business Domain Layer

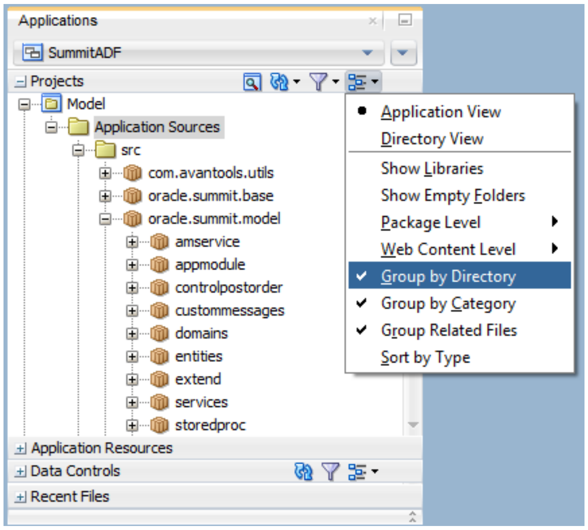


What Happens When You Create an Entity Diagram

When you create a business components diagram, JDeveloper creates an XML file `*.adfbc_diagram` representing the diagram in a subdirectory of the project's model path that matches the package name in which the diagram resides.

By default, the Applications window unifies the display of the project contents paths so that ADF components and Java files in the source path appear in the same package tree as the UML model artifacts in the project model path. However, as shown in Figure 4-10, using the **Applications Window Options** button in the Applications window, you can see the distinct project content path root directories when you prefer.

Figure 4-10 Toggling the Display of Separate Content Path Directories



What You May Need to Know About the XML Component Descriptors

When you include a business component like an entity object in a UML diagram, JDeveloper adds extra metadata to a `<Data>` section of the component's XML component descriptor as shown in the following example. This additional information is used at design time only.

```
<Entity Name="OrderEO" ... >
   <Data>
      <Property Name ="COMPLETE_LIBRARY" Value ="FALSE" />
      <Property Name ="ID"
                Value ="ff16fca0-0109-1000-80f2-8d9081ce706f::::EntityObject" />
      <Property Name ="IS_ABSTRACT" Value ="FALSE" />
      <Property Name ="IS_ACTIVE" Value ="FALSE" />
      <Property Name ="IS_LEAF" Value ="FALSE" />
      <Property Name ="IS_ROOT" Value ="FALSE" />
      <Property Name ="VISIBILITY" Value ="PUBLIC" />
   </Data>
   :
</Entity>
```

What You May Need to Know About Changing the Names of Components

On an entity diagram, the names of entity objects, attributes, and associations can be changed for clarity. Changing names on a diagram does not affect the underlying data names. The name change persists for the diagram only. The new name may contain spaces and mixed case for readability. To change the actual entity object names, attribute names, or association names, open the entity object or association in the overview editor.

Defining Property Sets

Define ADF property sets to use them as control hints and error messages.

A property set is a named collection of properties, where a property is defined as a name/value pair. Property sets are a convenience mechanism to group properties and then reference them from other ADF Business Components objects. Property sets can be used with entity objects and their attributes, view objects and their attributes, and application modules.

Property sets can be used for a variety of functions, such as **control hints** and error messages. A property set may contain control hints and other custom properties, and you can associate them with multiple attributes of different objects.

Properties defined in a property set can be configured to be translatable, in which case the translations are stored in a message bundle file owned by the property set. Because property sets are schema driven, in addition to adding names, you can add descriptions as well.

When defining property sets that contain translatable content, be sure not to overload common terms in different contexts. Even though a term in several contexts might be the same in the source language, a separate distinguishable term should be used for each context. For example, the term "Name" used as a value for a field's label might be applied to both an object and a person in one language, but then translated into two different terms in a target language. In this case you would provide two name/value pairs in which the value in the source language is "Name" for both. Then, when translated into a target language with one term for an object name and a different term for a person name, each term can be translated appropriately. You can use the property's description to provide the distinct context for each term.

How to Define a Property Set

To define a property set, you create a new property set using a dialog and then specify properties using the Properties window.

Before you begin:

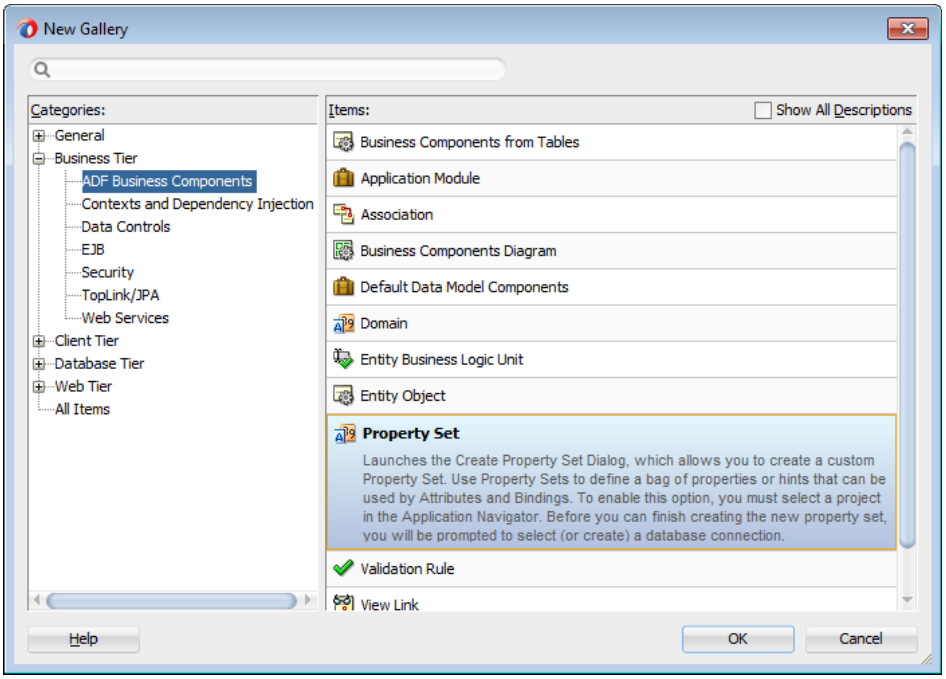
It may be helpful to have an understanding of how property sets can be used. For more information, see Defining Property Sets.

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Additional Functionality for Entity Objects.

To define a property set:

1. In the Applications window, right-click the project in which you want to create the property set, and choose **New > From Gallery**.
2. In the New Gallery, expand **Business Tier**, select **ADF Business Components** and then **Property Set**, and click **OK**.

Figure 4-11 Property Set in New Gallery



Launches the Create Property Set Dialog, which allows you to create a custom Property Set. Use Property Sets to define a bag of properties or hints that can be used by Attributes and Bindings. To enable this option, you must select a project in the Application Navigator. Before you can finish creating the new property set, you will be prompted to select (or create) a database connection.

3. In the Create Property Set dialog, enter the name and location of the property set and click **OK**.
4. From the main menu, choose **Window > Properties**.
5. In the Properties window, define the properties for the property set.

How to Apply a Property Set

After you have created the property set, you can apply the property set to an entity object or attribute, and use the defined properties (or override them, if necessary).

Before you begin:

It may be helpful to have an understanding of how property sets can be used. For more information, see Defining Property Sets.

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Additional Functionality for Entity Objects.

To apply a property set to an entity object or view object:

1. In the Applications window, double-click the object (entity object or view object) to which you want to apply a property set.
2. In the overview editor, click the **General** navigation tab and then click the **Edit** icon for **Property Set**.
3. In the Select Property Set dialog, select the appropriate property set and click **OK**.

To apply a property set to an attribute:

1. In the Applications window, double-click the object (entity object or view object) that contains the attribute to which you want to apply a property set.

2. In the overview editor, click the **Attributes** navigation tab, select the attribute you want to edit, and then click the **Details** tab.
3. On the Details page, select the appropriate property set from the **Property Set** dropdown list.

Defining Attribute Control Hints for Entity Objects

Control hints are inherited by ADF entity-based view objects. Use them to set label text, tooltip and format mask hints for attributes of entity objects.

Control hints allow you to define label text, tooltip, and format mask hints for entity object attributes. If desired, you can use a Groovy expression to calculate the value of the UI hint. The UI hints you define on your business domain layer are inherited by entity-based view objects as well. You can also set additional control hints on view objects and application modules in a similar manner.

How to Add Attribute Control Hints

To add attribute control hints to an entity object, use the overview editor.

Before you begin:

It may be helpful to have an understanding of how control hints are used in an entity object. For more information, see Defining Attribute Control Hints for Entity Objects.

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Additional Functionality for Entity Objects.

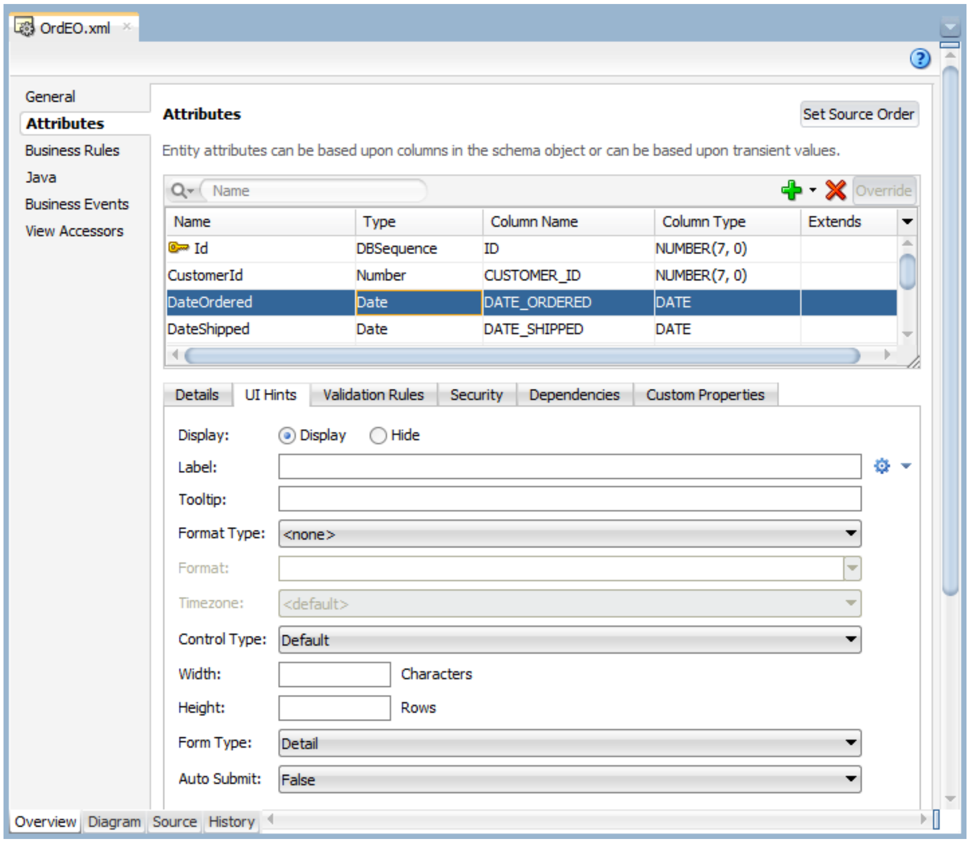
To add attribute control hints to an entity object:

1. In the Applications window, double-click the entity object that contains the attribute to which you want to add control hints.
2. In the overview editor, click the **Attributes** navigation tab and select the attribute you want to edit.
3. In the Attributes page, click the **UI Hints** tab and specify control hints as necessary.

   For example, Figure 4-12 shows control hints defined for the attribute `ExpireDate` of the `PaymentOptionEO` entity object. The defined hints include the following:

   - **Format Type** of `Simple Date`
   - **Format** mask of `mm/yy`

Figure 4-12   Overview Editor, Attributes Page, UI Hints Tab



Attributes

Entity attributes can be based upon columns in the schema object or can be based upon transient values.

Name	Type	Column Name	Column Type	Extends
Id	DBSequence	ID	NUMBER(7, 0)	
CustomerId	Number	CUSTOMER_ID	NUMBER(7, 0)	
DateOrdered	Date	DATE_ORDERED	DATE	
DateShipped	Date	DATE_SHIPPED	DATE	

Details | UI Hints | Validation Rules | Security | Dependencies | Custom Properties

Display:	● Display ○ Hide
Label:	
Tooltip:	
Format Type:	<none>
Format:	
Timezone:	<default>
Control Type:	Default
Width:	Characters
Height:	Rows
Form Type:	Detail
Auto Submit:	False

4. If you want to define the UI hint using a Groovy expression, click the arrow beside the text box and choose **Show Expression**. The application displays the **New Expression Method** hyperlink. Click this hyperlink to display the Edit Expression Editor. Enter the Groovy expression in this editor and click **OK**.

> **Note:**
>
> Java defines a standard set of format masks for numbers and dates that are different from those used by the Oracle database's SQL and PL/SQL languages. For reference, see the Javadoc for the `java.text.DecimalFormat` and `java.text.SimpleDateFormat` classes.

What Happens When You Add Attribute Control Hints

When you define attribute control hints for an entity object, JDeveloper creates a resource bundle file in which to store them. The hints that you define can be used by generated forms and tables in associated view clients. The type of file and its granularity are determined by Resource Bundle options in the Project Properties dialog. For more information, see Working with Resource Bundles.

About Formatters and Masks

When you set the **Format Type** control hint (on the **UI Hints** tab) for an attribute (for example, to **Simple Date**), you can also specify a format mask for the attribute to customize how the UI displays the value. If the mask you want to use is not listed in the **Format** dropdown list, you can simply type it into the field.

Not all formatters require format masks. Specifying a format mask is only needed if that formatter type requires it. For example, the date formatter requires a format mask, but the currency formatter does not. In fact, the currency formatter does not support format mask at all.

The mask elements that you can use are defined by the associated Java format class. For information about the mask elements for the Simple Date format type, see the Javadoc for `java.text.SimpleDateFormat`. For information about the mask elements for the Number format type, see the Javadoc for `java.text.DecimalFormat`.

To map a formatter to a domain for use with control hints, you can either amend one of the default formatters provided in the `oracle.jbo.format` package, or create a new formatter class by extending the `oracle.jbo.format.Formatter` class. The default formatters provided with JDeveloper aggregate the formatters provided in the `java.text` package.

For information on how to define a format mask to use from the **UI Hints** tab in JDeveloper, see How to Define Format Masks.

How to Define Format Masks

If you have a format mask that you will continue to use on multiple occasions, you can add it to the `formatinfo.xml` file, so that it is available from the **Format** dropdown list on the **UI Hints** tab. The entries in this file define the format masks and formatter classes for a domain class. The following example shows the format definitions for the `java.util.Date` domain.

```
<?xml version="1.0"?><FORMATTERS>
. . .
  <DOMAIN CLASS="java.util.Date">
     <FORMATTER name="Simple Date"
class="oracle.jbo.format.DefaultDateFormatter">
        <FORMAT text="yyyy-MM-dd" />
        <FORMAT text="EEE, MMM d, ''yy"  />
        <FORMAT text="dd-MM-yy" />
        <FORMAT text="dd-MMM-yyyy" />
        <FORMAT text="dd/MMM/yyyy" />
     </FORMATTER>
  </DOMAIN>
. . .
</FORMATTERS>
```

You can find the `formatinfo.xml`file in the BC4J subdirectory of the JDeveloper system directory (for example, `C:\Documents and Settings\`*`username`*`\Application Data\JDeveloper\`*`system##`*`\o.BC4J\formatinfo.xml`).

The definition of the format mask belongs to a formatter and a domain class, and includes the text specification of the mask as it appears on the **UI Hints** tab. When you specify the **Format Type** (FORMATTER name) for an attribute of a given type (DOMAIN CLASS), the masks (FORMAT text) appear in the **Format** dropdown list.

It is not necessary to create new domain to map a formatter. You can use an existing domain when the business components project contains a domain of the same data type as the formatter.

Before you begin:

It may be helpful to have an understanding of how control hints are used in an entity object. For more information, see Defining Attribute Control Hints for Entity Objects.

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Additional Functionality for Entity Objects.

To make a new format mask available in JDeveloper:

1. Open the formatinfo.xml file in a text editor.
2. Find the domain class and formatter name for which you want to add a format mask.
3. Insert a new FORMAT entry within the FORMATTER element.

If you create a new domain for the format mask, the XML definition of the formatter must include the new DOMAIN CLASS in addition to the FORMATTER (which includes the name and class) and the list of FORMAT definitions the formatter class specifies.

After defining a format mask, you can select the new format mask from the **Format** dropdown list on the **UI Hints** tab.

Working with Resource Bundles

JDeveloper stores translatable strings that you define on ADF Business Components in project-level resource bundle files.

When you define translatable strings (such as validator error messages, or attribute control hints for an entity object or view object), by default JDeveloper creates a project-level resource bundle file in which to store them. For example, when you define control hints for an entity object in the Model project, JDeveloper creates the message bundle file named ModelBundle.*xxx* for the package. The hints that you define can be used by generated forms and tables in associated view clients.

The resource bundle option that JDeveloper uses is determined by an option on the Resource Bundle page of the Project Properties dialog. By default JDeveloper sets the option to **Properties Bundle**, which produces a .properties file. For more information on this and other resource bundle options, see How to Set Message Bundle Options.

You can inspect the message bundle file for the entity object by selecting the object in the Applications window and looking in the corresponding **Sources** node in the Structure window. The Structure window shows the implementation files for the component you select in the Applications window.

The following example shows a sample message bundle file where the control hint information appears. The first entry in each String array is a message key; the second entry is the locale-specific String value corresponding to that key.

```
oracle.summit.model.views.ItemVO.QuantityShipped_LABEL=Shipped
oracle.summit.model.views.ItemVO.ItemTotal_LABEL=Item Total
oracle.summit.model.views.ItemVO.ItemTotal_FMT_FORMATTER=oracle.jbo.format.Defaul
tCurrencyFormatter
oracle.summit.model.views.ItemVO.Price_FMT_FORMATTER=oracle.jbo.format.DefaultCur
rencyFormatter
oracle.summit.model.views.OrdVO.DateOrdered_FMT_FORMATTER=oracle.jbo.format.Defau
ltDateFormatter
oracle.summit.model.views.OrdVO.DateOrdered_FMT_FORMAT=dd-MM-yyyy
oracle.summit.model.views.OrdVO.DateShipped_FMT_FORMATTER=oracle.jbo.format.Defau
ltDateFormatter
oracle.summit.model.views.OrdVO.DateShipped_FMT_FORMAT=dd-MM-yyyy
oracle.summit.model.entities.OrdEO_Rule_0=You cannot have a shipping date that
is before the order date
oracle.summit.model.entities.OrdEO.CustomerId_Rule_0=This is an invalid
customer id
errorId=This customer must pay cash
oracle.summit.model.entities.ItemEO.ProductId_Rule_0=Invalid Product Id
oracle.summit.model.views.OrdVO.Total_LABEL=Order Total
oracle.summit.model.views.OrdVO.Total_FMT_FORMATTER=oracle.jbo.format.DefaultCurr
encyFormatter
oracle.summit.model.views.OrdVO.OrderFilled_LABEL=Order Filled
. . .
```

How to Set Message Bundle Options

The resource bundle option JDeveloper uses to save control hints and other translatable strings is determined by an option on the Resource Bundle page of the Project Properties dialog. By default, JDeveloper sets the option to **Properties Bundle** which produces a `.properties` file.

Before you begin:

It may be helpful to have an understanding of how resource bundles are used. For more information, see Working with Resource Bundles.

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Additional Functionality for Entity Objects.

To set resource bundle options for your project:

1. In the Applications window, right-click the project for which you want to specify resource bundle options, and choose **Project Properties**.
2. In the Project Properties dialog, click **Resource Bundle**.
3. On the Resource Bundle page, specify whether to use project or custom settings.

   If you select **Use Custom Settings**, the settings apply only to your work with the current project. They are preserved between sessions, but are not recorded with the project and cannot be shared with other users. If you select **Use Project Settings**, your choices are recorded with the project and can be shared with others who use the project.
4. Specify your preference with the following options by selecting or deselecting the option:
   - **Automatically Synchronize Bundle**
   - **Warn About Hard-coded Translatable Strings**

- **Always Prompt for Description**

  For more information on these options, click **Help** to see the online help.

5. Select your choice of resource bundle granularity.
   - **One Bundle Per Project** (default)
   - **One Bundle Per File**
   - **Multiple Shared Bundles** (not available for ADF Business Components)
6. Select the type of file to use.
   - **List Resource Bundle**

     The `ListResourceBundle` class manages resources in a name/value array. Each ListResourceBundle class is contained within a Java class file. You can store any locale-specific object in a ListResourceBundle class.
   - **Properties Bundle** (default)

     A text file containing translatable text in name/value pairs. Property files (like the one described in How to Set Message Bundle Options) can contain values only for String objects. If you need to store other types of objects, you must use a ListResourceBundle instead.
   - **Xliff Resource Bundle**

     The XML Localization Interchange File Format (XLIFF) is an XML-based format for exchanging localization data.
7. Click **OK**.

How to Use Multiple Resource Bundles

When you define translatable strings (for example, for attribute control hints), the Select Text Resource dialog allows you to enter a new string or select one that is already defined in the default resource bundle for the object. You can also use a different resource bundle if necessary. This is helpful when you use a common resource bundle that is shared between projects.

Before you begin:

It may be helpful to have an understanding of how resource bundles are used. For more information, see Working with Resource Bundles.

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Additional Functionality for Entity Objects.

To use strings in a nondefault resource bundle:

1. In the Select Text Resource dialog, select the bundle you want to use from the **Resource Bundle** dropdown list.

   If the desired resource bundle is not included in the **Resource Bundle** dropdown list, click the **Browse** icon to locate and select the resource bundle you want to use.

   The dialog displays the strings that are currently defined in the selected resource bundle.

2. Select an existing string and click **Select**, or enter a new string and click **Save and Select**.

   If you entered a new string it is written to the selected resource bundle.

How to Internationalize the Date Format

Internationalizing the model layer of an application built using ADF Business Components entails producing translated versions of each component message bundle file. For example, the Italian version of the `OrdersImplMsgBundle` message bundle would be a class named `OrdersImplMsgBundle_it` and a more specific *Swiss* Italian version would have the name `OrdersImplMsgBundle_it_ch`. These classes typically extend the base message bundle class, and contain entries for the message keys that need to be localized, together with their localized translation.

The following example shows the Italian version of an entity object message bundle. Notice that in the Italian translation, the format masks for `RequestDate` and `AssignedDate` have been changed to `dd/MM/yyyy HH:mm`. This ensures that an Italian user will see a date value like May 3rd, 2006, as `03/05/2006 15:55`, instead of `05/03/2006 15:55`, which the format mask in the default message bundle would produce. Notice the overridden `getContents()` method. It returns an array of messages with the more *specific* translated strings merged together with those that are not overridden from the superclass bundle. At runtime, the appropriate message bundles are used automatically, based on the current user's locale settings.

```
package oracle.summit.model;
import oracle.jbo.common.JboResourceBundle;
public class ModelImplMsgBundle_it
       extends ModelImplMsgBundle {
  static final Object[][] sMessageStrings = {
    { "AssignedDate_FMT_FORMAT", "dd/MM/yyyy HH:mm" },
    { "AssignedDate_LABEL", "Assegnato il" },
    { "AssignedTo_LABEL", "Assegnato a" },
    { "CreatedBy_LABEL", "Aperto da" },
    { "ProblemDescription_LABEL", "Problema" },
    { "RequestDate_FMT_FORMAT", "dd/MM/yyyy HH:mm" },
    { "RequestDate_LABEL", "Aperto il" },
    { "RequestDate_TOOLTIP", "La data in cui il ticket è stato aperto" },
    { "Status_LABEL", "Stato" },
    { "SvrId_LABEL", "Ticket" }
  };
  public Object[][] getContents() {    return
super.getMergedArray(sMessageStrings, super.getContents());  }
}
```

Defining Business Logic Groups

Use Business Logic Groups to store related control hints, default values and validation logic of ADF entity objects in a separate file for optimized performance and easy maintenance.

Business logic groups allow you to encapsulate a set of related control hints, default values, and validation logic. A business logic group is maintained separate from the base entity in its own file, and can be enabled dynamically based on context values of the current row.

This is useful, for example, for an HR application that defines many locale-specific validations (like national identifier or tax law checks) that are maintained by a dedicated team for each locale. The business logic group eases maintenance by storing these validations in separate files, and optimizes performance by loading them only when they are needed.

Each business logic group contains a set of business logic units. Each unit identifies the set of business logic that is loaded for the entity, based on the value of the attribute associated with the business logic group.

For example, you can define a business logic group for an `Employee` entity object, specifying the `EmpRegion` attribute as the discriminator. Then define a business logic unit for each **region**, one that specifies a range validator for the employee's salary. When the application loads a row from the `Employee` entity, the appropriate validator for the `EmpSalary` attribute is loaded (based on the value of the `EmpRegion` attribute).

In another example, you might have a `Orders` entity object has a business logic group called `FilledOrderGroup` that uses `OrderFilled` as the discriminator attribute. Because this attribute has two valid values (`N` and `Y`), there are two corresponding business logic units.

In this scenario, each business logic unit contains new or modified business logic that pertains only to that person type:

- The `Orders_FilledOrderGroup_N` business logic unit contains logic that pertains to orders that have not yet been filled. For example, the `DateShipped` attribute is configured to be hidden.
- The `Orders_FilledOrderGroup_Y` business logic unit contains logic that pertains to orders that have been filled. For example, the `DateShipped` attribute is configured to be displayed.

How to Create a Business Logic Group

You create the business logic group for an entity object from the overview editor.

Before you begin:

It may be helpful to have an understanding of how business logic groups are used. For more information, see Defining Business Logic Groups.

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Additional Functionality for Entity Objects.

To create a business logic group:

1. In the Applications window, double-click the entity for which you want to create a business logic group.
2. In the overview editor, click the **General** navigation tab.
3. On the General page, expand the **Business Logic Groups** section and click the **Add** icon.
4. In the creation dialog, select the appropriate group discriminator attribute and specify a name for the group.

> **Tip:**
>
> To enhance the readability of your code, you can name the group to reflect the discriminator. For example, if the group discriminator attribute is `OrderFilled`, you can name the business logic group `FilledOrderGroup`.

5. Click **OK**.

The new business logic group is added to the table in the overview editor. After you have created the group, you can add business logic units to it.

How to Create a Business Logic Unit

You can create a business logic unit from the New Gallery, or directly from the context menu of the entity that contains the business logic group.

Before you begin:

It may be helpful to have an understanding of how business logic groups are used. For more information, see Defining Business Logic Groups.

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Additional Functionality for Entity Objects.

To create a business logic unit:

1. In the Applications window, right-click the entity object that contains the business logic group and choose **New Entity Business Logic Unit**.
2. In the Create Business Logic Unit dialog, specify the name of the base entity and select the appropriate business logic group.
3. Enter a name for the business logic unit.

   The name of each business logic unit must reflect a valid value of the group discriminator attribute with which this business logic unit will be associated. For example, if the group discriminator attribute is `OrderFilled`, the name of the business logic unit associated with the `OrderFilled` value of `Y` must be `Y`.
4. Specify the package for the business logic unit.

> **Note:**
>
> The package for the business logic unit does not need to be the same as the package for the base entity or the business logic group. This allows you to develop and deliver business logic units separately from the core application.

5. Click **OK**.

JDeveloper creates the business logic unit and opens it in the overview editor. The name displayed for the business logic unit in the Applications window contains the name of the entity object and business logic group in the format *`EntityName_BusLogicGroupName_BusLogicUnitName`*. For example, when you create a business logic unit with the name Y in the FilledOrderGroup business logic group of the Orders entity object, the displayed name of the business logic unit is Orders_FilledOrderGroup_Y.

After you have created the unit, you can redefine the business logic for it.

How to Add Logic to a Business Logic Unit

After you have created a business logic unit, you can open it in the overview editor and add business logic (such as adding an entity-level validator) just as you would in the base entity.

Before you begin:

It may be helpful to have an understanding of how business logic groups are used. For more information, see Defining Business Logic Groups.

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Additional Functionality for Entity Objects.

To add an entity validator to a business logic unit:

1. In the Applications window, double-click the business logic unit to which you want to add an entity validator.
2. In the overview editor, click the **Business Rules** navigation tab.
3. On the Business Rules page, select the **Entity Validators** node and click the **Add** icon.
4. In the Add Validation Rule dialog, define your validation rule, and click **OK**.

How to Override Attributes in a Business Logic Unit

When you view the Attributes page for the business logic unit (in the overview editor), you can see that the **Extends** column in the attributes table shows that the attributes are "extended" in the business logic unit. Extended attributes are editable only in the base entity, not in the business logic unit. To implement changes in the business logic unit rather than the base entity, you must define attributes as overridden in the business logic unit before you edit them.

Before you begin:

It may be helpful to have an understanding of how business logic groups are used. For more information, see Defining Business Logic Groups.

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Additional Functionality for Entity Objects.

To override attributes in a business logic unit:

1. In the Applications window, double-click the business logic unit that contains the attributes you want to override.
2. In the overview editor, click the **Attributes** navigation tab.
3. On the Attributes page, select the desired attribute and click the **Override** button.

After you make an attribute overridden, you can edit the attribute as you normally would in the tabs below the table. You will notice that in an overridden attribute, you are limited to making modifications to only control hints, validators, and default values.

What Happens When You Create a Business Logic Group

When you create a business logic group, JDeveloper adds a reference to the group in the base entity's XML file, as shown below.

```
<BusLogicGroup
    Name="FilledOrderGroup"
    DiscrAttrName="OrderFilled"/>
```

When you create a business logic unit, JDeveloper generates an XML file similar to that of an entity object. The following example shows XML code for a business logic unit.

> **Note:**
>
> The package for the business logic unit does not need to be the same as the package for the base entity or the business logic group. This allows you to develop and deliver business logic units separately from the core application.

```
<Entity
  xmlns="http://xmlns.oracle.com/bc4j"
  Name="Orders_FilledOrderGroup_N"
  Version="12.1.2.66.11"
  Extends="oracle.summit.model.buslogic.Orders"
  InheritPersonalization="merge"
  DBObjectType="table"
  DBObjectName="S_ORD"
  BindingStyle="OracleName"
  BusLogicGroupName="FilledOrderGroup"
  BusLogicUnitName="N">
  <Attribute
    Name="Total"
    Precision="11"
    Scale="2"
    ColumnName="TOTAL"
    SQLType="NUMERIC"
    Type="java.math.BigDecimal"
    ColumnType="NUMBER"
    TableName="S_ORD">
    <DesignTime>
      <Attr Name="_OverrideAttr" Value="true"/>
    </DesignTime>
    <Properties>
      <SchemaBasedProperties>
        <LABEL

ResId="oracle.summit.model.buslogic.Orders_FilledOrderGroup_N.Total_LABEL"/>
      </SchemaBasedProperties>
    </Properties>
  </Attribute>
  <Attribute
    Name="DateShipped"
    ColumnName="DATE_SHIPPED"
    SQLType="DATE"
    Type="java.sql.Date"
    ColumnType="DATE"
    TableName="S_ORD">
    <DesignTime>
      <Attr Name="_OverrideAttr" Value="true"/>
    </DesignTime>
    <Properties>
      <SchemaBasedProperties>
        <DISPLAYHINT
          Value="Hide"/>
      </SchemaBasedProperties>
    </Properties>
  </Attribute>
  <ResourceBundle>
    <PropertiesBundle
      PropertiesFile="oracle.summit.model.ModelBundle"/>
  </ResourceBundle>
</Entity>
```

What Happens at Runtime: How Business Logic Groups Are Invoked

When a row is loaded in the application at runtime, the entity object decides which business logic units to apply to it.

The base entity maintains a list of business logic groups. Each group references the value of an attribute on the entity, and this value determines which business logic unit to load for that group. This evaluation is performed for each row that is loaded.

If the logic for determining which business logic unit to load is more complex than just a simple attribute value, you can create a transient attribute on the entity object, and use a Groovy expression to determine the value of the transient attribute.

Configuring Runtime Behavior Declaratively

Use ADF entity objects to implement business logic and validation rules for the business layer using its declarative runtime features.

Entity objects offer numerous declarative features to simplify implementing typical enterprise business applications. Depending on the task, sometimes the declarative facilities *alone* may satisfy your needs. The declarative runtime features that describe the basic persistence features of an entity object are covered in this section, while declarative validation and business rules are covered in Defining Validation and Business Rules Declaratively.

> **Note:**
>
> It is possible to go beyond the declarative behavior to implement more complex business logic or validation rules for your business domain layer when needed. In Implementing Validation and Business Rules Programmatically, you'll see some of the most typical ways that you extend entity objects with custom code.

Also, it is important to note as you develop your application that the business logic you implement, either programmatically or declaratively, should not assume that the attributes of an entity object or view row will be set in a particular order. This will cause problems if the end user enters values for the attributes in an order other than the assumed one.

How to Configure Declarative Runtime Behavior

To configure the declarative runtime behavior of an entity object, use the overview editor.

Before you begin:

It may be helpful to have an understanding of declarative configuration of runtime behavior. For more information, see Configuring Runtime Behavior Declaratively.

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Additional Functionality for Entity Objects.

To configure the declarative runtime behavior of an entity object:

1. In the Applications window, double-click an entity object.
2. In the overview editor, click the **General** navigation tab to view the name and package of the entity object, and configure aspects of the object at the entity level, such as its associated schema, alternative keys, custom properties, and security.
   - The **Alternate Keys** section allows you to select entity object attributes mapped to the database that can serve as an alternative primary key. For information on alternative keys, see How to Define Alternate Key Values.
   - The **Tuning** section allows you to set options to make database operations more efficient when you create, modify, or delete multiple entities of the same type in a single transaction. For more information, see How to Use Update Batching.
   - The **Custom Properties** section allows you to define custom metadata that you can access at runtime on the entity.
   - The **Security** section allows you to define role-based updatability permissions for the entity. For more information, see Enabling ADF Security in a Fusion Web Application.
   - The **Business Logic Groups** section allows you to add and edit business logic groups. For more information, see Defining Business Logic Groups.
3. Click the **Attributes** navigation tab to create or delete attributes that represent the data relevant to an entity object, and configure aspects of the attribute, such as validation rules, custom properties, and security.

   Select an attribute and click the **Edit** icon to access the properties of the attribute. For information on how to set these properties, see Setting Attribute Properties.

> **Note:**
>
> If your entity has a long list of attribute names, there's a quick way to find the one you're looking for. In the Structure window with the **Attributes** node expanded, you can begin to type the letters of the attribute name and JDeveloper performs an incremental search to take you to its name in the tree.

4. Click the **Business Rules** navigation tab to define declarative validators for the entity object and its attributes. The script box allows you to view and edit Groovy script for your business rules, and set breakpoints for debugging. For more information, see Defining Validation and Business Rules Declaratively.
5. Click the **Java** navigation tab to select the classes you generate for custom Java implementation. You can use the Java classes for such things as defining programmatic business rules, as in Implementing Validation and Business Rules Programmatically.
6. Click the **Business Events** navigation tab to define events that your entity object can use to notify others of interesting changes in its state, optionally including some or all of the entity object's attributes in the delivered event. For more information about business events, see Creating Business Events.
7. Click the **View Accessors** navigation tab to create and manage view accessors. For more information, see How to Create a View Accessor for an Entity Object or View Object.

What Happens When You Configure Declarative Runtime Behavior

The declarative settings that describe and control an entity object's runtime behavior are stored in its XML document file. When you use the overview editor to modify settings of your entity, JDeveloper updates the component's XML definition file and optional custom Java files.

How to Use Update Batching

You can use update batching to reduce the number of DML statements issued with multiple entity modifications.

By default, the ADF Business Components framework performs a single DML statement (`DELETE`, `INSERT`, `UPDATE`) for each modified entity of a given entity definition type. For example, say you have an Employee entity object type for which multiple instances are modified during typical use of the application. If two instances were created, three existing instances modified, and four existing instances deleted, then at transaction commit time the framework issues nine DML statements (4 `DELETE`s, 2 `INSERT`s, and 3 `UPDATE`s) to save these changes.

If you will frequently be updating more than one entity of a given type in a transaction, consider using the update batching feature for that entity definition type. In the example, update batching (with a threshold of 1) causes the framework to issue just three DML statements: one bulk `DELETE` statement processing four deletes, one bulk `INSERT` statement processing two inserts, and one bulk `UPDATE` statement processing three updates.

The threshold value indicates the number of modified entity rows for a given operation that will be processed individually. When the number of modified entity rows exceeds the threshold value, batch processing will be used for that operation. In this example, if the threshold is 3, `DELETE` will perform batch update (because there are 4 `DELETE`s), but `INSERT`s and `UPDATE`s will not use batch (because there are only 2 `INSERT`s and 3 `UPDATE`s).

> **Note:**
>
> When update batching is used in entity objects with a composition relationship, all composed child objects must define a threshold value greater than 1 if the parent object defines a threshold value greater than 1.

The batch update feature is disabled if any of following conditions are present:

- The entity object has attributes of BLOB or CLOB type. Batch DML with streaming data types is not supported.
- The entity object has attributes that are set to **Refresh After Insert** or **Refresh After Update**. There is no method to bulk-return all of the trigger-assigned values in a single round trip, so retrieving and updating attributes doesn't work with batch DML.
- The entity object was created from a table that did not have a primary key. If an entity object is reverse-engineered from a table that does not have a primary key, a ROWID-valued attribute is created and assigned as the primary key instead. The ROWID value is managed as a Retrieve-on-Insert value, so it will not work with batch DML.

Before you begin:

It may be helpful to have an understanding of the declarative configuration of runtime behavior. For more information, see Configuring Runtime Behavior Declaratively.

You may also find it useful to understand additional functionality that can be added using other entity object features. For more information, see Additional Functionality for Entity Objects.

To enable update batching for an entity

1. In the Applications window, double-click the entity for which you want to enable update batching.
2. In the overview editor, click the **General** navigation tab.
3. On the General page, expand the **Tuning** section, select the **Use Update Batching** checkbox, and specify the appropriate threshold.

This establishes a batch processing threshold beyond which Oracle ADF will process the modifications in a bulk DML operation.

Setting Attribute Properties

Use the declarative framework to set attribute properties on the Attributes page of the overview editor of ADF Business Components.

The declarative framework helps you set attribute properties easily. In all cases, you set these properties on the Attributes page of the overview editor.

How to Set Database and Java Data Types for an Entity Object Attribute

The `Persistent` property controls whether the attribute value corresponds to a column in the underlying table, or whether it is just a transient value. If the attribute is persistent, the **Database Column** area lets you change the name of the underlying column that corresponds to the attribute and indicate its column type with precision and scale information (for example, `VARCHAR2(40)` or `NUMBER(4,2)`). Based on this information, at runtime the entity object enforces the maximum length or precision and scale of the attribute value, and throws an exception if a value does not meet the requirements.

Both the Business Components from Tables wizard and the Create Entity Object wizard infer the Java type of each entity object attribute from the SQL type of the database column type of the column to which it is related.

> **Note:**
>
> The project's Type Map setting also plays a role in determining the Java data type. You specify the Type Map setting when you initialize your business components project, before any business components are created. For more information, see How to Initialize the Data Model Project With a Database Connection.

The **Type** field (on the **Details** tab) allows you to change the Java type of the entity attribute to any type you might need. The **Column Type** field reflects the SQL type of the underlying database column to which the attribute is mapped. The value of the **Column Name** field controls the column to which the attribute is mapped.

Your entity object can handle tables with various column types, as listed in Table 4-1. The default Java attribute types are in the `oracle.jbo.domain` and various `java` packages, and support efficiently working with Oracle database data of the corresponding type. The dropdown list for the **Type** field includes a number of other common Java types that are also supported.

Table 4-1 Default Entity Object Attribute Type Mappings

Oracle Column Type	Entity Column Type	Entity Java Type
`NVARCHAR2(n)`, `VARCHAR2(n)`, `NCHAR VARYING(n)`, `VARCHAR(n)`	`VARCHAR2`	`java.lang.String`

`NUMBER`	`NUMBER`	`oracle.jbo.domain.Number`
		`java.math.BigDecimal`
		`java.math.BigInteger`
		`java.lang.Integer`
		`java.lang.Long`

`DATE`	`DATE`	`java.sql.Timestamp`

Table 4-1 (Cont.) Default Entity Object Attribute Type Mappings

Oracle Column Type	Entity Column Type	Entity Java Type
TIMESTAMP(n), TIMESTAMP(n) WITH TIME ZONE, TIMESTAMP(n) WITH LOCAL TIME ZONE	TIMESTAMP	java.sql.Timestamp
LONG	LONG	java.lang.String
RAW(n)	RAW	oracle.jbo.domain.Raw
LONG RAW	LONG RAW	oracle.jbo.domain.Raw
ROWID	ROWID	oracle.jbo.domain.RowID
NCHAR, CHAR	CHAR	oracle.jbo.domain.Char
CLOB	CLOB	oracle.jbo.domain.ClobDomain
NCLOB	NCLOB	oracle.jbo.domain.NClobDomain
BLOB	BLOB	oracle.jbo.domain.BlobDomain
BFILE	BFILE	oracle.jbo.domain.BFileDomain

Note:

In addition to the types mentioned here, you can use any Java object type as an entity object attribute's type, provided it implements the oracle.jbo.domain.DomainInterface interface.

How to Indicate Data Type Length, Precision, and Scale

When working with types that support defining a maximum length like VARCHAR2(n), the **Column Type** field (on the **Details** tab) includes the maximum attribute length as part of the value. For example, an attribute based on a VARCHAR2(10) column in the database will initially reflect the maximum length of 10 characters by showing VARCHAR2(10) as the database column type. If for some reason you want to restrict the maximum length of the String-valued attribute to fewer characters than the underlying column will allow, just change the maximum length of the **Column Type** value.

For example, if the EMAIL column in the PERSONS table is VARCHAR2(50), then by default the Email attribute in the Persons entity object defaults to the same. But if you know that the actual email addresses are always 8 characters or fewer, you can update the database column type for the Email attribute to be VARCHAR2(8) to enforce a maximum length of 8 characters at the entity object level.

The same holds for attributes related to database column types that support defining a precision and scale like NUMBER(p[,s]). For example, to restrict an attribute based on a NUMBER(7,2) column in the database to instead have a precision of 5 and a scale of 1, just update the value of the **Column Type** field to be NUMBER(5,1).

> **Note:**
>
> Use this feature judiciously. If other applications can access this database column, then you cannot be sure that the other applications will respect the more restrictive constraints.

How to Control the Updatability of an Attribute

The `Updatable` property controls when the value of a given attribute can be updated. You can select the following values:

- **Always,** the attribute is always updatable
- **Never**, the attribute is read-only
- **While New**, the attribute can be set during the transaction that creates the entity row for the first time, but after being successfully committed to the database the attribute is read-only

> **Note:**
>
> In addition to the static declaration of updatability, you can also add custom code in the `isAttributeUpdateable()` method of the entity to determine the updatability of an attribute at runtime.

How to Make an Attribute Mandatory

Select the **Mandatory** checkbox if the field is required. The mandatory property is enforced during entity-level validation at runtime (and not when the attribute validators are run).

How to Define the Primary Key for the Entity

The `Primary Key` property indicates whether the attribute is part of the key that uniquely identifies the entity. Typically, you use a single attribute for the primary key, but multiattribute primary keys are fully supported.

At runtime, when you access the related `Key` object for any entity row using the `getKey()` method, this `Key` object contains the value of the primary key attribute for the entity object. If your entity object has multiple primary key attributes, the `Key` object contains each of their values. It is important to understand that these values appear in the same *relative* sequential order as the corresponding primary key attributes in the entity object definition.

For example, if the `ItemEO` entity object has multiple primary key attributes `OrdId` and `ItemId`. On the Attribute page of the overview editor, `OrdId` is first, and `ItemId` is second. An array of values encapsulated by the `Key` object for an entity row of type `ItemEO` will have these two attribute values in exactly this order.

It is crucial to be aware of the order in which multiple primary key attributes appear on the Entity Attributes page. If you try to use `findByPrimaryKey()` to find an entity with a multiattribute primary key, and the `Key` object you construct has these multiple primary key attributes in the wrong order, the entity row will not be found as expected.

In addition, to populate the primary key in new rows, you might want to use a trigger to assign the value from the database. For more information, see How to Get Trigger-Assigned Primary Key Values from a Database Sequence

How to Define a Static Default Value

The value field on the **Details** tab allows you to specify a static default value for the attribute when the value type is set to **Literal**. For example, you can set the default value of the `ServiceRequest` entity object's `Status` attribute to `Open`, or set the default value of the `User` entity object's `UserRole` attribute to `user`.

> **Note:**
>
> When more than one attribute is defaulted for an entity object, the attributes are defaulted in the order in which they appear in the entity object's XML file.

How to Define a Default Value Using an Expression

You can use a Groovy expression or SQL statement to define a default value for an attribute. This approach is useful if you want to be able to dynamically define default values at runtime, but if the default value is always the same, the value is easier to see and maintain using a value field with the **Literal** type (on the **Details** tab). For general information about using Groovy, see Using Groovy Scripting Language with Business Components.

Before you begin:

It may be helpful to have an understanding of how you set attribute properties. For more information, see Setting Attribute Properties.

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Additional Functionality for Entity Objects.

To define a default value using an expression:

1. In the Applications window, double-click the entity object that contains the attribute for which you want to define a default value.
2. In the overview editor, click the **Attributes** navigation tab.
3. On the Attributes page, select the attribute you want to edit, and then click the **Details** tab.
4. In the Details page, select the value type, and click the **Edit** icon for the **Value** field.
   - To use a Groovy expression, select **Expression**.
   - To use a SELECT statement, select **SQL**.

5. To enter:
   - SQL statement, enter the statement in the field provided and click **OK**. Note that you can enter an SQL statement only for transient attributes.
   - Groovy expression, click the icon beside to open the Edit Expression Editor dialog. In the Edit Expression Editor dialog, enter an expression in the field provided. Attributes that you reference can include any attribute that the business component defines. Specify the attributes on which this attribute is dependent by moving these attributes from **Available to Selected.** Click **OK** to save the expression

What Happens When You Create a Default Value Using a Groovy expression

When you define a default value using a Groovy expression, a `<TransientExpression>` tag is added to this business object's XML file within the appropriate attribute. The following example shows XML code for a Groovy expression that has been added for the `ID` attribute of the `CountryVO` view object. The XML definition informs you that the `.bcs` file where this Groovy expression script resides in is pointed to by the `operations.xml` file, which in this example is `CodeSourceName=”CountryVOOperations.xml”`. The `operations.xml` file has an URI which points the user to the `.bcs` file.

```
<View Attribute
    Name="Id"
    IsNotNull="true"
    PrecisionRule="true"
    EntityAttrName="Id"
    EntityUsage="CountryEO"
    AliasName="ID">
    <TransientExpression
        Name="ExpressionScript"
        trustMode="untrusted"
        CodeSourceName="CountryVORow"/>

</ViewAttribute>
```

> **Note:**
>
> If Groovy file generation is enabled for this Model project or application-wide, then the Groovy expression displays as a hyperlink in Overview Editor page of this business component under the Default Value section. Click this hyperlink to open the `.bcs` file associated with this business component. Refer the topic What You May Need to Know About Groovy Project Settings to read more about project settings. Refer this topic What Happens When You Enter Expressions to read about how the application manages Groovy expressions when you enter them.

How to Synchronize with Trigger-Assigned Values

If you know that the underlying column value will be updated by a database trigger during insert or update operations, you can enable the respective **Refresh on Insert** or **Refresh on Update** checkboxes on the **Details** tab to ensure that the framework automatically retrieves the modified value and keeps the entity object and database row in sync. The entity object will use the Oracle SQL RETURNING INTO feature, while performing the INSERT or UPDATE to return the modified column back to your application in a single database roundtrip.

> **Note:**
>
> If you create an entity object for a synonym that resolves to a remote table over a DBLINK, use of this feature will give an error at runtime like:
>
> ```
> JBO-26041: Failed to post data to database during "Update"
> ## Detail 0 ##
> ORA-22816: unsupported feature with RETURNING clause
> ```
>
> Basing an Entity Object on a Join View or Remote DBLink describes a technique to circumvent this database limitation.

How to Get Trigger-Assigned Primary Key Values from a Database Sequence

One common case for refreshing an attribute after insert occurs when a primary key attribute value is assigned by a BEFORE INSERT FOR EACH ROW trigger. Often the trigger assigns the primary key from a database sequence using PL/SQL logic, as shown in the following example.

```
CREATE OR REPLACE TRIGGER ASSIGN_SVR_ID
BEFORE INSERT ON SERVICE_REQUESTS FOR EACH ROW
BEGIN
 IF :NEW.SVR_ID IS NULL OR :NEW.SVR_ID < 0 THEN
   SELECT SERVICE_REQUESTS_SEQ.NEXTVAL
     INTO :NEW.SVR_ID
     FROM DUAL;
   END IF;
END;
```

> **Note:**
>
> You can create a trigger in a live database or, as an alternative, create it in an offline database and subsequently import the trigger to the live database. For more information about working with offline database objects, see the "Creating, Editing, and Dropping Database Objects" section in *Developing Applications with Oracle JDeveloper*.

On the **Details** tab (on the Attributes page of the overview editor), you can set the value of the **Type** field to the built-in data type named DBSequence, and the primary key will be assigned automatically by the database sequence. Setting this data type automatically selects the **Refresh on Insert** checkbox.

> **Note:**
>
> The sequence name shown on the **Sequence** tab is used only at design time when you use the **Create Database Tables** feature described in How to Create Database Tables from Entity Objects. The sequence indicated here will be created along with the table on which the entity object is based.

When you create a new entity row whose primary key is a `DBSequence`, a unique negative number is assigned as its temporary value. This value acts as the primary key for the duration of the transaction in which it is created. If you are creating a set of interrelated entities in the same transaction, you can assign this temporary value as a foreign key value on other new, related entity rows. At transaction commit time, the entity object issues its `INSERT` operation using the `RETURNING INTO` clause to retrieve the actual database trigger-assigned primary key value. In a composition relationship, any related new entities that previously used the temporary negative value as a foreign key will get that value updated to reflect the actual new primary key of the master.

You will typically also set the **Updatable** property of a DBSequence-valued primary key to **Never**. The entity object assigns the temporary ID, and then refreshes it with the actual ID value after the `INSERT` operation. The end user never needs to update this value.

For information on how to implement this functionality for an association that is not a composition, see Associations Based on DBSequence-Valued Primary Keys.

> **Note:**
>
> For a metadata-driven alternative to the DBSequence approach, see Assigning the Primary Key Value Using an Oracle Sequence.

What You May Need to Know About Changing the Value of Primary Keys

The updating of primary key values through the user interface is not supported by ADF Business Components. You can, however, update a primary key value using DBSequence, as described in How to Get Trigger-Assigned Primary Key Values from a Database Sequence or programmatically, as described in Assigning the Primary Key Value Using an Oracle Sequence. In some cases, this kind of user interface behavior can be implemented using a surrogate key rather than allowing the user to change the primary key.

How to Protect Against Losing Simultaneously Updated Data

At runtime, the framework provides "lost update" detection for entity objects to ensure that a user cannot unknowingly modify data that another user has updated and committed in the meantime. Typically, this check is performed by comparing the original values of each persistent entity attribute against the corresponding current column values in the database at the time the underlying row is locked. Before updating a row, the entity object verifies that the row to be updated is still consistent with the current state of the database. If the row and database state are inconsistent, then the entity object raises the RowInconsistentException.

You can make the lost update detection more efficient by identifying any attributes of your entity whose values you know will be updated whenever the entity is modified. Typical candidates include a version number column or an updated date column in the row. The change-indicator attribute's value might be assigned by a database trigger you've written and refreshed in the entity object, because you selected the **Refresh on Insert** or **Refresh on Update** option (on the **Details** tab). Alternatively, you can indicate that the entity object should manage updating the change-indicator attribute's value using the history attribute feature described in How to Track Created and Modified Dates Using the History Column.

To detect, in the most efficient way, whether an entity row has been modified since the user queried it, select the **Change Indicator** option to compare only the change-indicator attribute values. Entity objects without a change-indicator attribute may perform less efficiently because values in all attributes will be used in data consistency checks. Entity objects without a change-indicator attribute are also subject to a loss of data consistency when the a user session loses affinity with its application module (when **application module pooling** is used). For more information, see What You May Need to Know About State Management and Data Consistency. For more information about application module pooling, see Tuning Application Module Pools.

How to Protect Against Truncated Attribute Values at Runtime

In rare situations, at runtime during application module passivation, string values may not be preserved correctly. For example, whitespace may get truncated. You may define a custom property on the entity object attribute to encapsulate your string value in CDATA section.

To define the XML_CDATA custom property and preserve whitespace formatting of an entity object attribute's default value:

1. In the overview editor for the entity object, select the **Attribute** page and then select the attribute and click the Custom Properties tab.
2. In the Custom Properties tab, add the property with the name XML_CDATA and set the value to true.

JDeveloper adds the custom property to the entity object XML definition, as the following example for an attribute CategoryCode shows.

```
<Attribute
   Name="CategoryCode"
   ...
   <Properties>
     <CustomProperties>
       <Property
         Name="XML_CDATA"
         ResId="model.Departments.LocationId.XML_CDATA_VALUE"/>
     </CustomProperties>
   </Properties>
</Attribute>
```

How to Track Created and Modified Dates Using the History Column

If you need to keep track of historical information in your entity object, such as when an entity was created or modified and by whom, or the number of times the entity has been modified, you specify an attribute with the **Track Change History** option selected (on the **Details** tab).

If an attribute's data type is `Number`, `String`, or `Date`, and if it is not part of the primary key, then you can enable this property to have your entity automatically maintain the attribute's value for historical auditing. How the framework handles the attribute depends which type of history attribute you indicate:

- **Created On**: This attribute is populated with the time stamp of when the row was created. The time stamp is obtained from the database.
- **Created By**: The attribute is populated with the name of the user who created the row. The user name is obtained using the `getUserPrincipalName()` method on the `Session` object.
- **Modified On**: This attribute is populated with the time stamp whenever the row is updated/created.
- **Modified By**: This attribute is populated with the name of the user who creates or updates the row.
- **Version Number**: This attribute is populated with a long value that is incremented whenever a row is created or updated.

How to Configure Composition Behavior

An entity object exhibits composition behavior when it creates (or composes) other entities, such as an `OrdEO` entity creating an `ItemEO` entity. This additional runtime behavior determines its role as a logical container of other nested entity object parts. Because of this relationship, a composition association cannot be based on a transient attribute.

> **Note:**
>
> Composition also affects the order in which entities are validated. For more information, see Understanding the Impact of Composition on Validation Order.

The features that are always enabled for composing entity objects are described in the following sections:

- Orphan-Row Protection for New Composed Entity Objects
- Ordering of Changes Saved to the Database
- Cascade Update of Composed Details from Refresh-On-Insert Primary Keys

The additional features, and the properties that affect their behavior, are described in the following sections:

- Cascade Delete Support
- Cascade Update of Foreign Key Attributes When Primary Key Changes
- Locking of Composite Parent Entity Objects
- Using Batch Update with Composition Entity Objects
- Updating of Composing Parent History Attributes

Orphan-Row Protection for New Composed Entity Objects

When a composed entity object is created, it performs an existence check on the value of its foreign key attribute to ensure that it identifies an existing entity as its owning parent entity. At create time, if no foreign key is found or else a value that does not identify an existing entity object is found, the entity object throws an `InvalidOwnerException` instead of allowing an orphaned child row to be created without a well-identified parent entity.

> **Note:**
>
> The existence check finds new pending entities in the current transaction, as well as existing ones in the database if necessary.

Ordering of Changes Saved to the Database

Composition behavior ensures that the sequence of data manipulation language (DML) operations performed in a transaction involving both composing and composed entity objects is performed in the correct order. For example, an `INSERT` statement for a new composing parent entity object will be performed before the DML operations related to any composed children.

Cascade Update of Composed Details from Refresh-On-Insert Primary Keys

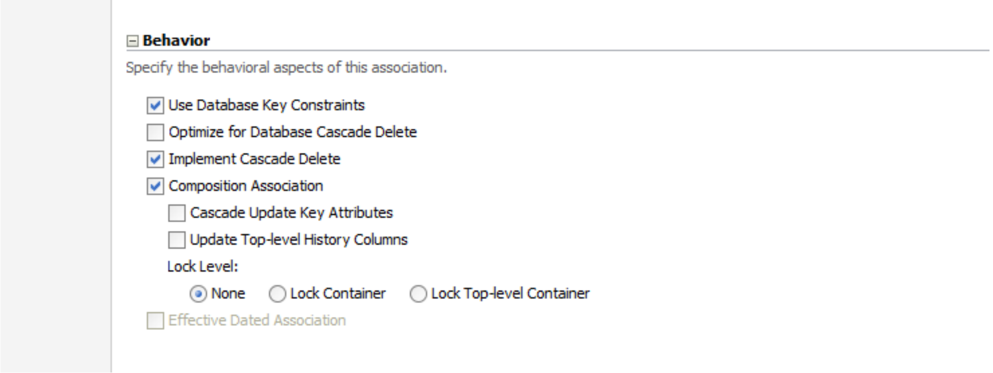
When a new entity row having a primary key configured to refresh on insert is saved, then after its trigger-assigned primary value is retrieved, any composed entities will have their foreign key attribute values updated to reflect the new primary key value.

There are a number of additional composition related features that you can control through settings on the Association Properties page of the Create Association wizard or the overview editor. Figure 4-13 shows the Relationships page for the `SItemOrdIdFkAssoc` association between two entity objects: `ItemEO` and `OrdEO`.

Cascade Delete Support

You can either enable or prevent the deletion of a composing parent while composed children entities exist. When the **Implement Cascade Delete** option (see Figure 4-13) is deselected, the removal of the composing entity object is prevented if it contains any composed children.

Figure 4-13 Behavior Settings on Relationship Page of Overview Editor for Associations



⊟ **Behavior**

Specify the behavioral aspects of this association.

☑ Use Database Key Constraints
□ Optimize for Database Cascade Delete
☑ Implement Cascade Delete
☑ Composition Association
□ Cascade Update Key Attributes
□ Update Top-level History Columns

Lock Level:

● None ○ Lock Container ○ Lock Top-level Container

□ Effective Dated Association

When selected, this option allows the composing entity object to be removed unconditionally together with any composed children entities. If the related **Optimize for Database Cascade Delete** option is deselected, then the composed entity objects perform their normal `DELETE` statement at transaction commit time to make the changes permanent. If the option is selected, then the composed entities do not perform the `DELETE` statement on the assumption that the database `ON DELETE CASCADE` constraint will handle the deletion of the corresponding rows.

Cascade Update of Foreign Key Attributes When Primary Key Changes

Select the **Cascade Update Key Attributes** option (see Figure 4-13) to enable the automatic update of the foreign key attribute values in composed entities when the primary key value of the composing entity is changed.

Locking of Composite Parent Entity Objects

Select the **Lock Top-Level Container** option (see Figure 4-13) to control whether adding, removing, or modifying a composed detail entity row should attempt to lock the composing entity before allowing the changes to be saved.

Using Batch Update with Composition Entity Objects

When update batching is used in entity objects with a composition relationship, all composed child objects must define a threshold value greater than 1 if the parent object defines a threshold value greater than 1.

Updating of Composing Parent History Attributes

Select the **Update Top-Level History Columns** option (see Figure 4-13) to control whether adding, removing, or modifying a composed detail entity object should update the **Modified By** and **Modified On** history attributes of the composing parent entity.

How to Set the Discriminator Attribute for Entity Object Inheritance Hierarchies

Sometimes a single database table stores information about several different kinds of logically related objects. For example, a payroll application might work with hourly, salaried, and contract employees all stored in a single `EMPLOYEES` table with an `EMPLOYEE_TYPE` column. In this case, the value of the `EMPLOYEE_TYPE` column contains values like `H`, `S`, or `C` to indicate respectively whether a given row represents an hourly, salaried, or contract employee. And while it is possible that many attributes and behavior are the same for all employees, certain properties and business logic may also depend on the type of employee.

In situations where common information exists across related objects, it may be convenient to represent these different types of entity objects using an inheritance hierarchy. For example, attributes and methods common to all employees can be part of a base `Employee` entity object, while subtype entity objects like `HourlyEmployee`, `SalariedEmployee`, and `ContractEmployee` extend the base `Employee` object and add additional properties and behavior. The **Discriminator** attribute setting is used to indicate which attribute's value distinguishes the type of row. Using Inheritance in Your Business Domain Layer, explains how to set up and use inheritance.

How to Define Alternate Key Values

Database primary keys are often generated from a sequence and may not be data you want to expose to the user for a variety of reasons. For this reason, it's often helpful to have alternate key values that are unique. For example, you might want to enforce that every customer have a unique email address. Because a customer may change their email address, you won't want to use that value as a primary key, but you still want the user to have a unique field they can use for login or other purposes.

Alternate keys are useful for direct row lookups via the `findByKey` class of methods. Alternate keys are frequently used for efficient uniqueness checks in the middle tier. For information on how to find out if a value is unique, see How to Ensure That Key Values Are Unique.

To define an alternate key, you use the Create Entity Constraint wizard.

Before you begin:

It may be helpful to have an understanding of how you set attribute properties. For more information, see Setting Attribute Properties.

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Additional Functionality for Entity Objects.

To define alternate key values:

1. In the Applications window, right-click the entity object for which you want to define an alternate key and choose **New Entity Constraint**.
2. Follow the steps in the Create Entity Constraint wizard to name your constraint and select the attribute or attributes that participate in the key.
3. On the Properties page, select **Alternate Key** and choose the appropriate **Key Properties** options.

For more information about the **Key Properties** options, press the F1 key or click **Help**.

What Happens When You Define Alternate Key Values

When you define alternate key values, a hash map is created for fast access to entities that are already in memory.

What You May Need to Know About Alternate Key Values

The Unique key constraint is used only for forward generation of `UNIQUE` constraints in the database, not for alternate key values.

Adding Transient and Calculated Attributes to an Entity Object

Add transient attributes in ADF entity objects based on one-to-many columns. Use an entity object’s Java class or use a Groovy expression to define a calculated attribute of a transient attribute.

In addition to having attributes that map to columns in an underlying table, your entity objects can include transient attributes that display values calculated (for example, using Java or Groovy) or that are value holders. For example, a transient attribute you create, such as `FullName`, could be calculated based on the concatenated values of `FirstName` and `LastName` attributes.

Once you create the transient attribute, you can perform a calculation in the entity object Java class, or use a Groovy expression in the attribute definition to specify a default value.

If you want to be able to change the way the value is calculated at runtime, you can use a Groovy expression. If the way the value is calculated is not likely to change (for example, if it's a sum of the line items), you can perform the calculation directly in the entity object Java class.

How to Add a Transient Attribute

Use the Attributes page of the overview editor to create a transient attribute.

Before you begin:

It may be helpful to have an understanding of the use of transient and calculated attributes. For more information, see Adding Transient and Calculated Attributes to an Entity Object.

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Additional Functionality for Entity Objects.

To add a transient attribute to an entity object:

1. In the Applications window, double-click the entity object to which you want to add a transient attribute.

2. In the overview editor, click the **Attributes** navigation tab, and then click the **New** icon and choose **New Attribute**.
3. In the New Entity Attribute dialog, enter a name for the attribute, select the Java attribute type from the **Type** dropdown list, and click **OK**.
4. On the Attributes page of the overview editor, click the **Details** tab and select the **Transient** radio button.
5. If the value will be calculated, select **Never** from the **Updatable** dropdown list.

What Happens When You Add a Transient Attribute

When you add a transient attribute, JDeveloper updates the XML document for the entity object to reflect the new attribute. The `<Attribute>` tag of a transient attribute has no `TableName` or `ColumnName`, as shown in the following example.

```
<Attribute
   Name="FullName"
   IsUpdateable="false"
   IsQueriable="false"
   IsPersistent="false"
   Type="java.lang.String"
   SQLType="VARCHAR" >
</Attribute>
```

In contrast, a persistent entity attribute has both a `TableName` and a `ColumnName`, as shown in the following example.

```
<Attribute
   Name="FirstName"
   IsNotNull="true"
   Precision="30"
   ColumnName="FIRST_NAME"
   Type="java.lang.String"
   ColumnType="VARCHAR2"
   SQLType="VARCHAR"
   TableName="USERS" >
</Attribute>
```

How to Base a Transient Attribute on a Groovy Expression

When creating a transient attribute, you can use a Groovy expression to provide the default value.

Before you begin:

It may be helpful to have an understanding of transient and calculated attributes. For more information, see Adding Transient and Calculated Attributes to an Entity Object.

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Additional Functionality for Entity Objects.

To create a transient attribute based on a Groovy expression:

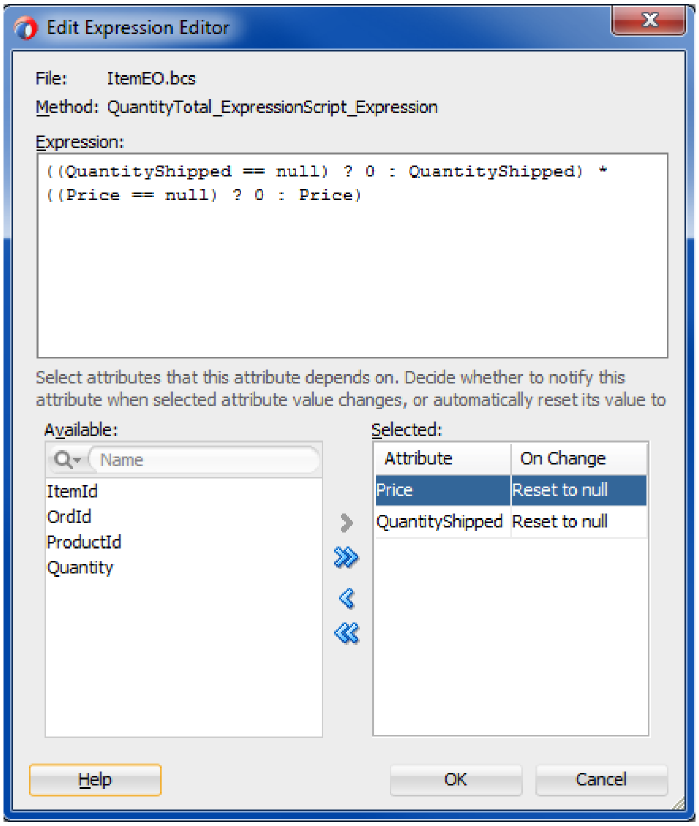
1. In the Applications window, double-click the entity object to which you want to add a transient attribute.

2. In the overview editor, click the **Attributes** navigation tab, and then click the **New** icon and choose **New Attribute**.
3. In the New Entity Attribute dialog, enter a name for the attribute, select the Java attribute type from the **Type** dropdown list, and click **OK**.
4. On the Attributes page of the overview editor, click the **Details** tab and select the **Transient** option.
5. If the value will be calculated, select **Never** from the **Updatable** dropdown list.
6. Select **Expression** checkbox under **Default Value** and click the icon beside to open the **Edit Expression Editor** dialog.

   Expressions that you define are evaluated using the Groovy scripting language, as described in Using Groovy Scripting Language with Business Components. Use the Groovy programming language to insert expressions and variables into strings. The expression is saved as part of the entity object definition.
7. In the Edit Expression Editor dialog, enter an expression in the field provided.

   Attributes that you reference can include any attribute that the entity object defines. For attributes not defined by the entity object, use the appropriate context object to reference these attributes in the expression.

Figure 4-14 Edit Expression Editor



File: ItemEO.bcs

Method: QuantityTotal_ExpressionScript_Expression

Expression:

```
((QuantityShipped == null) ? 0 : QuantityShipped) *
((Price == null) ? 0 : Price)
```

Select attributes that this attribute depends on. Decide whether to notify this attribute when selected attribute value changes, or automatically reset its value to

Available:

ItemId
OrdId
ProductId
Quantity

Selected:

Attribute	On Change
Price	Reset to null
QuantityShipped	Reset to null

8. Specify the attributes on which this attribute is dependent. Move these attributes from **Available** to **Selected**.
9. Click **OK** to save the expression.

10. If required, apply a recalculation expression that if evaluates to be true, recalculates the default expression. In the **Refresh Expression Value** field, click the icon beside to open the Edit Expression Editor dialog. Use the same procedure that you used to define a default expression to define the recalculation expression as well.

    For example, the following expression defined using the Edit Expression Editor dialog for the **Refresh Expression Value** field causes the attribute to be recalculated when either the `QuantityShipped` attribute or the `Price` attributes are changed:

    ```
    return (adf.object.isAttributeChanged("QuantityShipped") ||
    adf.object.isAttributeChanged("Price"));
    ```

> **Note:**
>
> If either the value expression or the optional recalculate expression that you define references an attribute from the base entity object, you must define this as a dependency. If you didn't provide dependencies in the Edit Expression Editor dialog when you defined the expression, you can do so on the **Dependencies** tab (on the Attributes page). On the **Dependencies** tab, locate the attributes in the **Available** list and shuttle each to the **Selected** list.

What Happens When You Base a Transient Attribute on a Groovy Expression

When you base a transient attribute on a Groovy expression, a `<TransientExpression>` tag is added to the entity object's XML file within the appropriate attribute, as shown in the following example.

```
<Attribute
  Name="QuantityTotal"
  ColumnName="QUANTITY_TOTAL"
  ...
  <TransientExpression
    Name="ExpressionScript"
    trustMode="untrusted"
    CodeSourceName="ItemEORow"/>
  <Dependencies>
    <Item
      Value="Price"/>
    <Item
      Value="QuantityShipped"/>
  </Dependencies>
</Attribute>
```

How to Add Java Code in the Entity Class to Perform Calculation

A transient attribute is a placeholder for a data value. If you change the **Updatable** property of the transient attribute to **While New** or **Always**, then the end user can enter a value for the attribute. If you want the transient attribute to display a calculated value, then you'll typically leave the **Updatable** property set to **Never** and write custom Java code that calculates the value.

After adding a transient attribute to the entity object, to make it a *calculated* attribute you need to:

- Enable a custom entity object class on the Java page of the overview editor, choosing to generate accessor methods
- Write Java code inside the accessor method for the transient attribute to return the calculated value
- Specify each dependent attribute for the transient attribute on the **Dependencies** tab of the Attributes page

For example, after generating the view row class, the Java code to return the transient attribute's calculated value would reside in the getter method for the attribute (such as `FullName`), as shown in the following example.

```
// Getter method for FullName calculated attribute in UserImpl.java
public String getFullName() {
  // Commented out original line since we'll always calculate the value
  // return (String)getAttributeInternal(FULLNAME);
      return getFirstName()+" "+getLastName();
}
```

To ensure that the transient attribute is reevaluated whenever the attributes to be concatenated (such as `LastName` and `FirstName`) might be changed by the end user, specify the dependent attributes for the transient attribute. Even though you perform the computation in the code, the dependencies are needed to fire the appropriate events that cause the UI to refresh when any of the dependent attribute values change. To specify dependent attributes, on the Dependencies tab of the Attributes page, locate the attributes in the **Available** list and shuttle each to the **Selected** list.

Creating Business Events

You can define and publish business events from the ADF Model layer and synchronize them with external systems by way of Oracle Mediator.

Business events raised from the model layer are useful for launching business processes and triggering external systems synchronization.

You declaratively define business events at the entity level. You may also specify conditions under which those events should be raised. Business events that meet the specified criteria are raised upon successful commit of the changed data. A business event is raised to the Mediator on a successful create, update, or delete of an entity object.

To implement a business event, you perform the following tasks:

1. Create an event definition, as described in How to Create a Business Event.
2. Map the event definition to an event point and publish the event definition, as described in How to Define a Publication Point for a Business Event.

Introducing Event Definitions

An event definition describes an event that will be published and raised with an event system Mediator. An event definition is stored in an entity object's XML file with the elements shown in Table 4-2.

Table 4-2 Event Definition Elements for Entity Objects

Element	Description
Event Name	Name of the event, for example, `OrderUpdated`
Payload	A list of attributes sent to the subscriber. Attributes marked as optional appear on payload only if changed.

Introducing Event Points

An event point is a place from which an event can be raised. On a successful commit, one of the event points shown in Table 4-3 can be raised to the Mediator for each entity in a transaction.

Table 4-3 Example Event Points Raised to the Mediator

DML Type	Event Name	Event Description
CREATE	*Entity*Created	A new *Entity* has been created.
UPDATE	*Entity*Updated	An existing *Entity* has been updated.
DELETE	*Entity*Deleted	An existing *Entity* has been deleted.

Note that no events are raised by default; all events are custom. When you create the event, you can specify the name and DML operation appropriately.

For each event point, you must specify which event definitions should be raised on a particular event point. In other words, you must declaratively map each event definition to an event point.

What You May Need to Know About Event Points

Event delivery occurs outside the current database transaction, and is asynchronous. Therefore, the following are not supported:

- **Transactional event delivery,** where event delivery is part of the transaction, is not supported by the framework.
- **Synchronous events**, where the publisher waits for further processing until the subscriber has confirmed event reception, is not supported by the framework.

How to Create a Business Event

To create a business event, use the Business Events page of the overview editor.

Before you begin:

It may be helpful to have an understanding of how business events work. For more information, see Creating Business Events.

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Additional Functionality for Entity Objects.

To create a business event:

1. In the Applications window, double-click the entity object for which you want to define a business event.
2. In the overview editor, click the **Business Events** navigation tab.
3. On the Business Events page, expand the **Event Definitions** section and click the **New** icon.
4. In the Create Business Event Definition dialog, provide a name that describes this event, such as `EmployeeContactInfoChanged`.
5. In the payload table, click **New** and **Delete** to select the appropriate attributes for this event.

   Alternatively, you can double-click the cell and pick the attributes you want.

   > **Note:**
   >
   > Only attributes of supported types are displayed in the Entity Attribute column. While `ClobDomain` attributes are supported, very large clob data can impact performance.

6. In the **Value Sent** field, choose whether the value should **Always** be sent, or **Only if changed**.

   The **Only if changed** option provides the best performance because the attribute will be considered optional for the payload. If you leave the default **Always**, the payload will require the attribute whether or not the value has changed. For more details about payload efficiency, see What You May Need to Know About Payload.
7. Use the arrow buttons to rearrange the order of attributes.

   The order that the attributes appear in defines their order in the generated XSD. Since you'll be using the XSD to build your Fabric mediator and BPEL process, you might want the most frequently accessed attributes at the top.
8. Click **OK**.

Repeat the procedure for each business event that you want to define. To publish an event, see How to Define a Publication Point for a Business Event.

What Happens When You Create a Business Event

When you create a business event, the entity object's XML file is updated with the event definition, as shown in the following example.

```
<EventDef
    Name="CustBusEvent1">
    <Payload>
      <PayloadItem
        AttrName="Id"/>
      <PayloadItem
        AttrName="PaymentOptionId"/>
      <PayloadItem
        AttrName="PaymentTypeEO.Id"
        SendOnlyIfChanged="true"/>
    </Payload>
  </EventDef>
```

JDeveloper also generates an associated XSD file for the event schema and an event definition (EDL) file for the entity object. The EDL file provides an event definition, as shown in the following example.

```
<definitions
    targetNamespace="http://oracle/summit/model/entities/events/edl/OrdEO"
    xmlns:ns0="http://oracle/summit/model/entities/events/schema/OrdEO"
    xmlns="http://schemas.oracle.com/events/edl">
  <schema-import
     namespace="http://oracle/summit/model/entities/events/schema/OrdEO"
     location="OrdEO.xsd"/>
  <event-definition name="CustBusEvent1">
     <content element="ns0:CustBusEvent1Info"/>
  </event-definition>
</definitions>
```

The XSD file allows specification of required attributes and optional attributes. Required attributes correspond to **Value Sent - Always** in the Create Business Event Definition dialog, whereas optional attributes are those for which you changed **Value Sent** to **Only if changed**.

The following example shows an XSD event schema for a business event.

```
<?xml version = '1.0' encoding = 'UTF-8'?>
<xs:schema targetNamespace="http://oracle/summit/model/entities/events/schema/
OrdEO" xmlns="http://oracle/summit/model/entities/events/schema/OrdEO"
                                           elementFormDefault="qualified"
attributeFormDefault="unqualified" xmlns:xs="http://www.w3.org/2001/XMLSchema">
   <xs:element name="CustBusEvent1Info">
      <xs:complexType>
         <xs:sequence>
            <xs:element name="Id" type="DecimalValuePair" minOccurs="1"/>
            <xs:element name="PaymentOptionId" type="IntValuePair"
minOccurs="1"/>
         </xs:sequence>
      </xs:complexType>
   </xs:element>
   <xs:complexType name="ValuePair" abstract="true"/>
   <xs:complexType name="IntValuePair">
      <xs:complexContent>
         <xs:extension base="ValuePair">
            <xs:sequence>
               <xs:element name="newValue" minOccurs="0">
                  <xs:complexType>
                     <xs:complexContent>
                        <xs:extension base="xs:anyType">
                           <xs:attribute name="value" type="xs:int"/>
                        </xs:extension>
                     </xs:complexContent>
                  </xs:complexType>
               </xs:element>
               <xs:element name="oldValue" minOccurs="0">
                  <xs:complexType>
                     <xs:complexContent>
                        <xs:extension base="xs:anyType">
                           <xs:attribute name="value" type="xs:int"/>
                        </xs:extension>
                     </xs:complexContent>
                  </xs:complexType>
               </xs:element>
            </xs:sequence>
         </xs:extension>
      </xs:complexContent>
   </xs:complexType>
   <xs:complexType name="DecimalValuePair">
      <xs:complexContent>
         <xs:extension base="ValuePair">
            <xs:sequence>
               <xs:element name="newValue" minOccurs="0">
                  <xs:complexType>
                     <xs:complexContent>
                        <xs:extension base="xs:anyType">
                           <xs:attribute name="value" type="xs:decimal"/>
                        </xs:extension>
                     </xs:complexContent>
                  </xs:complexType>
               </xs:element>
               <xs:element name="oldValue" minOccurs="0">
                  <xs:complexType>
                     <xs:complexContent>
                        <xs:extension base="xs:anyType">
                           <xs:attribute name="value" type="xs:decimal"/>
                        </xs:extension>
                     </xs:complexContent>
                  </xs:complexType>
               </xs:element>
            </xs:sequence>
         </xs:extension>
      </xs:complexContent>
   </xs:complexType>
</xs:schema>
```

What You May Need to Know About Payload

The attributes of the associated entity object constitute the payload of a business event. The payload attributes for a business event are defined by the creator of the event. It isn't automatically optimized. When the event is defined, an attribute can be marked as sent **Always** or **Only if changed**. For events fired during creation, only new values are sent. For events fired during an update or delete, the new and old values are sent for only the attributes that should be sent, based on the **Value Sent** setting. For best performance, you should include only the primary key attribute for delete events.

To support composition scenarios (such as a purchase order with line items), a child entity can raise events defined on the parent entity, and events defined on the child entity can include attributes from the parent entity. When a child entity raises an event on a parent entity, only a single event is raised for a particular top-level entity per transaction, regardless of how many times the child entity raises it.

In the case of entity subtypes (for example, a `Staff` entity object is a subtype of the `Persons` entity), ADF Business Components does not support the overriding of business events. Because the subscriber to a business event listens to the event using the event name, overriding of events could cause the event subscriber to receive payload data unintended for that subscriber. Therefore, this capability is not supported.

When defining business events, remember that while `ClobDomain` attributes are supported, very large clob data can have performance implications.

How to Define a Publication Point for a Business Event

To define a publication point for a business event, use the Business Events page of the entity object overview editor.

Before you begin:

It may be helpful to have an understanding of how business events are used in the application. For more information, see Creating Business Events.

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Additional Functionality for Entity Objects.

You need to have already created the event definition, as described in How to Create a Business Event, before you can publish it.

To define a publication point for a business event:

1. In the Applications window, double-click the entity object that contains the business event you want to publish.
2. In the overview editor, click the **Business Events** navigation tab.
3. On the Business Events page, expand the **Event Publication** section and click the **Edit event publications** icon.
4. In the Edit Event Publications dialog, click **New** to create a new event.
5. Click the new cell in **Event** column, and select the appropriate event.
6. Click the corresponding cell in **Event Point** column, and select the appropriate event point action.
7. You can optionally define conditions for raising the event using the **Raise Conditions** table.
8. Click **OK**.

How to Subscribe to Business Events

After you have created a business event, you can subscribe and respond to the event.

> **Note:**
>
> To subscribe to business events or create and deploy SOA composite applications and projects in JDeveloper, you must install the Oracle SOA Suite extension. For instructions on installing this extension for JDeveloper, see Enabling Oracle JDeveloper Extensions in *Installing Oracle JDeveloper*. For more information about the capabilities of Oracle SOA Suite, see Introduction to Building Applications with Oracle SOA Suite in *Developing SOA Applications with Oracle SOA Suite*.

Before you begin:

It may be helpful to have an understanding of business events. For more information, see Creating Business Events.

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Additional Functionality for Entity Objects.

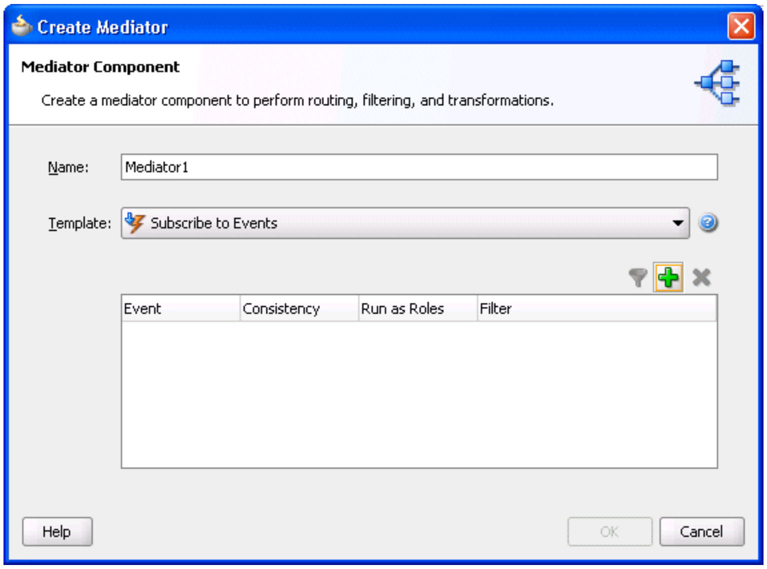
You will also need to complete the following task:

- Publish the business event, as described in How to Define a Publication Point for a Business Event.

To subscribe to a business event:

1. Using the file system, copy the XSD and event definition files for the business event into your SCA project's source path.
2. In JDeveloper, open the SCA project that will subscribe to the business event.
3. In the Applications window, right-click the SCA project that will subscribe to the business event, and choose **New > From Gallery**.
4. In the New Gallery, expand **SOA Tier**, select **Service Components** and then **Mediator**, and click **OK**.
5. In the Create Mediator dialog, select the **Subscribe to Events** template, as shown in Figure 4-15.

Figure 4-15 Create Mediator Dialog, Subscribe to Events



Create Mediator

Mediator Component

Create a mediator component to perform routing, filtering, and transformations.

Name: Mediator1

Template: Subscribe to Events

Event	Consistency	Run as Roles	Filter

6. Click the **Add** icon to add an event.
7. In the Event Chooser dialog, click the **Browse** icon to navigate to and select the event's definition file, and then click **OK**.
8. In the Create Mediator dialog, you can optionally change the **Consistency** option and specify a **Filter** for the event.
9. Click **OK** to generate the mediator.

   The resulting mediator (`.mplan` file) is displayed in the overview editor.

10. In the overview editor, click the **Add** icon in the **Routing Rules** section to add a rule for how to respond to the event.

Generating Custom Java Classes for an Entity Object

Enable custom Java generation for an ADF entity object to implement custom business logic that cannot be achieved using its declarative runtime functionality.

As described in this chapter, all of the database interaction and a large amount of declarative runtime functionality of an entity object can be achieved without using custom Java code. When you need to go beyond the declarative features to implement custom business logic for your entities, you'll need to enable custom Java generation for the entities that require custom code. Most Commonly Used ADF Business Components Methods, provides a quick reference to the most common code that you will typically write, use, and override in your custom entity object and entity definition classes.

> **Best Practice:**
>
> Oracle recommends that developers getting started with ADF Business Components consider overriding the base framework classes to enable all generated custom Java classes to pick up customized behavior. This practice is preferred over creating custom classes for individual entity objects and view objects. You use the **ADF Business Components > Base Classes** page of the Project Properties dialog to specify the framework classes to override when you generate custom component. See Extending Business Components Functionality.

How to Generate Custom Classes

To enable the generation of custom Java classes for an entity object, use the Java page of the overview editor.

Before you begin:

It may be helpful to have an understanding of custom Java classes. For more information, see Generating Custom Java Classes for an Entity Object.

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Additional Functionality for Entity Objects.

To generate a custom Java class for an entity object:

1. In the Applications window, double-click the entity object for which you want to generate a custom Java class.
2. In the overview editor, click the **Java** navigation tab, and then click the **Edit Java options** icon.
3. In the Select Java Options dialog, select the types of Java classes you want to generate.
   - Entity Object Class — the most frequently customized, it represents each row in the underlying database table.

- Entity Collection Class — rarely customized.
- Entity Definition Class — less frequently customized, it represents the related class that manages entity rows and defines their structure.

4. Click **OK**.

What Happens When You Generate Custom Classes

When you select one or more custom Java classes to generate, JDeveloper creates the Java file(s) you've indicated. For example, assuming an entity object named `summit.model.entities.OrdEO`, the default names for its custom Java files will be `OrdEOImpl.java` for the entity object class and `OrdEODefImpl.java` for the entity definition class. Both files are created in the same `./summit/model/entities` directory as the component's XML document file.

The Java generation options for the entity object continue to be reflected on subsequent visits to the Java page of the overview editor. Just as with the XML definition file, JDeveloper keeps the generated code in your custom Java classes up to date with any changes you make in the editor. If later you decide you didn't require a custom Java file for any reason, disabling the relevant options on the Java page causes the custom Java files to be removed.

What Happens When You Generate Entity Attribute Accessors

When you enable the generation of a custom entity object class, if you also enable the **Accessors** option, then JDeveloper generates getter and setter methods for each attribute in the entity object.

> **Caution:**
>
> Because the resulting hard-coded method names impose limitations on optimizations or customizations for entity and view rows, you should generate an accessor method only if there is a business need for it (such as, having to expose the view row as a Java interface to a client).

For example, an `OrdEO` entity object that has the corresponding custom `OrdEOImpl.java` class might have methods like those shown in the following example.

```
public DBSequence getId() { ... }
public void setId(DBSequence value) { ... }

public Date getDateOrdered() { ... }
public void setDateOrdered(Date value) { ... }

public Integer getPaymentTypeId() { ... }
public void setPaymentTypeId(Integer value) { ... }

public Number getCustomerId() { ... }
public void setCustomerId(Number value) { ... }

public String getOrderFilled() { ... }
public void setOrderFilled(String value) { ... }
```

These methods allow you to work with the row data with compile-time checking of the correct data type usage. That is, instead of writing a line like this to get the value of the `CustomerId` attribute:

```
Number customerId = (Number)order.getAttribute("CustomerId");
```

you can write the code like:

```
Number customerId = order.getCustomerId();
```

You can see that with the latter approach, the Java compiler would catch a typographical error had you accidentally typed `CustomerCode` instead of `CustomerId`:

```
// spelling name wrong gives compile error
Number customerId = order.getCustomerCode();
```

Without the generated entity object accessor methods, an incorrect line of code like the following cannot be caught by the compiler:

```
// Both attribute name and type cast are wrong, but compiler cannot catch it
String customerId = (String)order.getAttribute("CustomerCode");
```

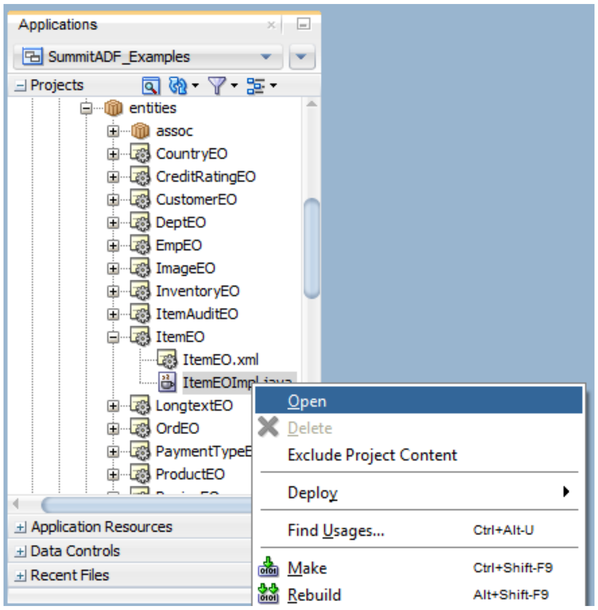
It contains both an incorrectly spelled attribute name, as well as an incorrectly typed cast of the `getAttribute()` return value. When you use the generic APIs on the `Row` interface, which the base `EntityImpl` class implements, errors of this kind raise exceptions at runtime instead of being caught at compile time.

How to Navigate to Custom Java Files

As shown in Figure 4-16, when you've enabled generation of custom Java classes, they also appear as child nodes under the **Application Sources** node for the entity object. As with all ADF components, when you select an entity object in the Applications window, the Structure window provides a structural view of the entity. When you need to see or work with the source code for a custom Java file, there are multiple ways to open the file in the source editor:

- You can right-click the Java file, and choose **Open**, as shown in Figure 4-16.
- You can right-click an item in a node in the Structure window, and choose **Go To Source**.
- You can click the link on the Java page of the overview editor.

Figure 4-16 Seeing and Navigating to Custom Java Classes for an Entity Object



What You May Need to Know About Custom Java Classes

The custom Java classes generated by JDeveloper extend the base classes for your entity object, and allow you the flexibility to implement custom code while maintaining the integrity of the generated code. The following sections provide additional information about custom Java classes.

Framework Base Classes for an Entity Object

When you use an XML-only entity object, at runtime its functionality is provided by the default ADF Business Components implementation classes. Each custom Java class that is generated extends the appropriate ADF Business Components base class so that your code inherits the default behavior and you can easily add to or customize it. An entity object class will extend `EntityImpl`, while the entity definition class will extend `EntityDefImpl` and the entity collection class will extend `EntityCache`. These base classes are in the `oracle.jbo.server` package.

Safely Adding Code to the Custom Component File

Some developers are hesitant to add their own code to generated Java source files. Each custom Java source code file that JDeveloper creates and maintains for you includes the following comment at the top of the file to clarify that it is safe for you to add your own custom code to this file.

```
// ---------------------------------------------------------------------
// ---    File generated by ADF Business Components Design Time.
// ---    Custom code may be added to this class.
// ---    Warning: Do not modify method signatures of generated methods.
// ---------------------------------------------------------------------
```

JDeveloper does not blindly regenerate the file when you click **OK** or **Apply** in an edit dialog. Instead, it performs a smart update to the methods that it needs to maintain, leaving your own custom code intact.

Configuring Default Java Generation Preferences

You can generate custom Java classes for your view objects when you need to customize their runtime behavior or when you simply prefer to have strongly typed access to bind variables or view row attributes.

To configure the default settings for ADF Business Components custom Java generation, you can choose **Tools >Preferences** from the main menu and open the Business Components page to set your preferences to be used for business components created in the future. Developers getting started with ADF Business Components should set their preference to generate no custom Java classes (the default setting). As you run into a specific need for custom Java code, you can enable just the bit of custom Java you need for that one component. Over time, you'll discover which set of defaults works best for you.

Attribute Indexes and InvokeAccessor Generated Code

The entity object is designed to function based on XML only or as an XML document combined with a custom Java class. To support this design choice, attribute values are not stored in private member fields of an entity's class (a file that is not present in the XML-only situation). By default, reflection is used at runtime to assign attribute enumerations and invoke attribute accessors.

When reflection is turned off for the project, attributes are assigned a name and a numerical index in the entity's XML document based on the zero-based, sequential order of the `<Attribute>` and association-related `<AccessorAttribute>` tags in that file. At runtime, attribute values in an entity row are stored in a sparse array structure managed by the base `EntityImpl` class, indexed by the attribute's numerical position in the entity's attribute list.

> **Note:**
>
> To disable reflection, deselect **Use reflection to invoke attribute accessors** on the **Options** page (under **ADF Business Components**) in the Project Properties dialog.

For the most part, this private implementation detail is unimportant, since as a developer using entity objects, you are shielded from having to understand this. However, when you enable a custom Java class for your entity object, this implementation detail relates to some of the generated code that JDeveloper maintains in your entity object class. It is sensible to understand what that code is used for. For example, in the custom Java class for an `Orders` entity object, each attribute or accessor attribute has a corresponding generated integer enum. JDeveloper ensures that the values of these enums correctly reflect the ordering of the attributes in the XML document.

You'll also notice that the automatically maintained, strongly typed getter and setter methods in the entity object class use these attribute enums, as shown in the following example.

```
// In oracle.summit.model.appmodule.OrdersImpl class
    /**
     * Gets the attribute value for DateOrdered, using the alias name DateOrdered.
     * @return the value of DateOrdered
     */
  public Date getDateOrdered() {
    return (Date)getAttributeInternal(DATEORDERED);
  }
    /**
     * Sets <code>value</code> as the attribute value for DateOrdered.
     * @param value to set the DateOrdered
     */
  public void setDateOrdered(Date value) {
    setAttributeInternal(DATEORDERED, value);
  }
```

Another aspect of the maintained code related to entity attribute enums is the `getAttrInvokeAccessor()` and `setAttrInvokeAccessor()` methods. These methods optimize the performance of attribute access by numerical index, which is how generic code in the `EntityImpl` base class typically accesses attribute values when performing generic processing. An example of the `getAttrInvokeAccessor()` method is shown in the following example. The companion `setAttrInvokeAccessor()` method looks similar.

```
// In oracle.summit.model.appmodule.OrdersImpl class
  /**
   * getAttrInvokeAccessor: generated method. Do not modify.
   * @param index the index identifying the attribute
   * @param attrDef the attribute
   * @return the attribute value
   * @throws Exception
   */

protected Object getAttrInvokeAccessor(int index, AttributeDefImpl attrDef)
      throws Exception {
  if ((index >= AttributesEnum.firstIndex()) && (index < AttributesEnum.count())) {
    return AttributesEnum.staticValues()[index - AttributesEnum.firstIndex()].get(this);
  }
  return super.getAttrInvokeAccessor(index, attrDef);
}
```

The rules of thumb to remember about this generated attribute index-related code are the following.

The Do's

- Add custom code if needed inside the strongly typed attribute getter and setter methods.
- Use the overview editor to change the order or type of entity object attributes.

  JDeveloper changes the Java signature of getter and setter methods, as well as the related XML document for you.

The Don'ts

- Don't modify the `getAttrInvokeAccessor()` and `setAttrInvokeAccessor()` methods.
- Don't change the values of the attribute index numbers manually.

Note:

If you need to manually edit the generated attribute enums because of source control merge conflicts or other reasons, you must ensure that the zero-based ordering reflects the sequential ordering of the `<Attribute>` and `<AccessorAttribute>` tags in the corresponding entity object XML document.

Programmatic Example for Comparison Using Custom Entity Class References

To better evaluate the difference between using strongly typed entity references to custom entity classes and working with the generic entity references, consider the following example. The following example shows a version of strongly typed entity references in a custom application module class (`AppModuleImpl.java`). Some important differences to notice are:

- Attribute access is performed using strongly typed attribute accessors.
- Association accessor attributes return the strongly typed entity class on the other side of the association.
- Using the `getDefinitionObject()` method in your custom entity class allows you to avoid working with fully qualified entity definition names as strings.
- The `createPrimaryKey()` method in your custom entity class simplifies creating the `Key` object for an entity.

```
package oracle.summit.model.appmodule.service;

import oracle.jbo.ApplicationModule;
import oracle.jbo.JboException;
import oracle.jbo.Key;
import oracle.jbo.Row;
import oracle.jbo.RowSetIterator;
import oracle.jbo.ViewObject;
import oracle.jbo.client.Configuration;
import oracle.jbo.domain.DBSequence;
import oracle.jbo.domain.Number;
import oracle.jbo.server.EntityDefImpl;
import oracle.jbo.server.EntityImpl;
import oracle.jbo.server.ViewLinkImpl;

import oracle.summit.base.SummitApplicationModuleImpl;
import oracle.summit.base.SummitViewObjectImpl;
import oracle.summit.model.appmodule.CustomerImpl;
import oracle.summit.model.appmodule.OrdersImpl;
import oracle.summit.model.appmodule.service.common.AppModule;

// ---------------------------------------------------------------------
// ---    File generated by Oracle ADF Business Components Design Time.
// ---    Custom code may be added to this class.
// ---    Warning: Do not modify method signatures of generated methods.
// ---------------------------------------------------------------------
public class AppModuleImpl extends SummitApplicationModuleImpl
       implements AppModule {
```

```
  /* This is the default constructor (do not remove). */
  public AppModuleImpl() {
  }

  private OrdersImpl retrieveOrderById(long orderId) {
    EntityDefImpl orderDef = OrdersImpl.getDefinitionObject();
    Key orderKey = OrdersImpl.createPrimaryKey(new DBSequence(orderId));
    return (OrdersImpl)orderDef.findByPrimaryKey(getDBTransaction(),orderKey);
  }

  public String findOrderAndCustomer(long orderId) {
    OrdersImpl order = retrieveOrderById(orderId);
    if (order != null) {
      CustomerImpl cust = order.getCustomer();
      if (cust != null) {
        return "Customer: " + cust.getName() + ", Location: " + cust.getCity();
        }
      else {
        return "Unassigned";
      }
    }
    else {
      return null;
    }
  }

    /* Find an Order by Id */
  public String findOrderTotal(long orderId) {
    OrdersImpl order = retrieveOrderById(orderId);
    if (order != null) {
      return order.getTotal().toString();
    }
    return null;
  }

    /* Update the status of an existing order */
  public void updateOrderStatus(long orderId, String newStatus) {
    OrdersImpl order = retrieveOrderById(orderId);
    if (order != null) {
      order.setOrderFilled(newStatus);
      try {
        getDBTransaction().commit();
      }
      catch (JboException ex) {
        getDBTransaction().rollback();
        throw ex;
      }
    }
  }

    /* Create a new Customer and Return the new id */
  public long createCustomer(String name, String city, Integer countryId) {
    EntityDefImpl customerDef = CustomerImpl.getDefinitionObject();
    CustomerImpl newCustomer =
(CustomerImpl)customerDef.createInstance2(getDBTransaction(),null);
    newCustomer.setName(name);
    newCustomer.setCity(city);
    newCustomer.setCountryId(countryId);
    try {
      getDBTransaction().commit();
    }
  catch (JboException ex) {
    getDBTransaction().rollback();
    throw ex;
  }
  DBSequence newIdAssigned = newCustomer.getId();
  return newIdAssigned.getSequenceNumber().longValue();
}

  /* Custom method in an application module implementation class */
public void doSomeCustomProcessing() {
  ViewObject vo = getCustomerView1();
  // create secondary row set iterator with system-assigned name
  RowSetIterator iter = vo.createRowSetIterator(null);
  while (iter.hasNext()) {
    Row r = iter.next();
    // Do something with the current row.
    Integer custId = (Integer)r.getAttribute("Id");
    String name  = (String)r.getAttribute("Name");
    System.out.println(custId + " " + name);
  }
  // close secondary row set iterator
  iter.closeRowSetIterator();
}

  /* Container's getter for CustomerView1.
   * @return CustomerView1 */
public SummitViewObjectImpl getCustomerView1() {
  return (SummitViewObjectImpl) findViewObject("CustomerView1");
}

  /* Container's getter for ItemView1.
   * @return ItemView1 */
public SummitViewObjectImpl getItemView1() {
  return (SummitViewObjectImpl) findViewObject("ItemView1");
}

  /* Container's getter for OrderView1.
   * @return OrderView1 */
public SummitViewObjectImpl getOrderView1() {
  return (SummitViewObjectImpl) findViewObject("OrderView1");
}

  /* Container's getter for ItemView2.
   * @return ItemView2 */
public SummitViewObjectImpl getItemView2() {
  return (SummitViewObjectImpl) findViewObject("ItemView2");
}

  /* Container's getter for OrderView2.
   * @return OrderView2 */
public SummitViewObjectImpl getOrderView2() {
  return (SummitViewObjectImpl) findViewObject("OrderView2");
}

  /*Container's getter for SItemOrdIdFkLink1.
   * @return SItemOrdIdFkLink1 */
public ViewLinkImpl getSItemOrdIdFkLink1() {
  return (ViewLinkImpl) findViewLink("SItemOrdIdFkLink1");
}

  /* Container's getter for SOrdCustomerIdFkLink1.
```

```
     * @return SOrdCustomerIdFkLink1 */
  public ViewLinkImpl getSOrdCustomerIdFkLink1() {
    return (ViewLinkImpl) findViewLink("SOrdCustomerIdFkLink1");
  }
}
```

Working Programmatically with Entity Objects and Associations

JDeveloper allows you to access an ADF application module and work directly in the ADF data model using an external client program.

You may not always need or want UI-based or programmatic clients to work directly with entity objects. Sometimes, you may just want to use an external client program to access an application module and work directly with the view objects in its data model. Defining SQL Queries Using View Objects describes how to easily combine the flexible SQL-querying of view objects with the business logic enforcement and automatic database interaction of entity objects to build powerful applications. The combination enables a fully updatable **application module data model**, designed to meet the needs of the current end-user tasks at hand, that shares the centralized business logic in your reusable domain business object layer.

However, it is important first to understand how view objects and entity objects can be used on their own before learning to harness their combined power. By learning about these objects in greater detail, you will have a better understanding of when you should use them alone and when to combine them in your own applications.

Since clients don't work *directly* with entity objects, any code you write that works programmatically with entity objects will typically be custom code in a custom application module class or in the custom class of another entity object.

How to Find an Entity Object by Primary Key

To access an entity row, you use a related object called the **entity definition**. At runtime, each entity object has a corresponding entity definition object that describes the structure of the entity and manages the instances of the entity object it describes. After creating an application module and enabling a custom Java class for it, imagine you wanted to write a method to return a specific order. It might look like the `retrieveOrderById()` method shown in the sample code below.

Before you begin:

It may be helpful to have an understanding of when to use a programmatic approach for working with entity objects and associations. For more information, see Working Programmatically with Entity Objects and Associations.

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Additional Functionality for Entity Objects.

To find an entity object by primary key:

1. Find the entity definition.

   You obtain the entity definition for an entity object (for example, `Orders`) by passing its fully qualified name to the static `getDefinitionObject()` method imported from the EntityDefImpl class. The EntityDefImpl class in the oracle.jbo.server package implements the entity definition for each entity object.

2. Construct a key.

   You build a Key object containing the primary key attribute that you want to look up. For example, for the Orders entity object you create a key containing the single orderId value passed into the method as an argument.

3. Find the entity object using the key.

   You use the entity definition's findByPrimaryKey() method to find the entity object by key, passing in the current transaction object, which you can obtain from the application module using its getDBTransaction() method. The concrete class that represents an entity object row is the oracle.jbo.server.EntityImpl class.

4. Return the object or some of its data to the caller.

The following example code shows a retrieveOrderById() method developed using this basic procedure.

```
/* Helper method to return an Order by Id  */
private OrdersImpl retrieveOrderById(long orderId) {
  EntityDefImpl orderDef = OrdersImpl.getDefinitionObject();
  Key orderKey = OrdersImpl.createPrimaryKey(new DBSequence(orderId));
  return (OrdersImpl)orderDef.findByPrimaryKey(getDBTransaction(),orderKey);
}
```

> **Note:**
>
> The oracle.jbo.Key object constructor can also take an Object array to support creating multi-attribute keys, in addition to the more typical single-attribute value keys.

How to Access an Associated Entity Using the Accessor Attribute

You can create a method to access an associated entity based on an accessor attribute that requires no SQL code. For example, the method findOrderAndCustomer() might find an order, then access the associated Customer entity object representing the customer assigned to the order. For an explanation of how associations enable easy access from one entity object to another, see Creating and Configuring Associations.

To prevent a conflict with an existing method in the application module that finds the same associated entity using the same accessor attribute, you can refactor this functionality into a helper method that you can then reuse anywhere in the application module it is required. For example, you might create a retrieveOrderById() method to refactor the functionality that finds an order, as the shown in the sample code below.

Before you begin:

It may be helpful to have an understanding of when to use a programmatic approach for working with entity objects and associations. For more information, see Working Programmatically with Entity Objects and Associations.

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Additional Functionality for Entity Objects.

To access an associated entity object using the accessor attribute:

1. Find the associated entity by the accessor attribute.

   For example, the `findOrderAndCustomer()` method uses the `retrieveOrderById()` helper method to retrieve the `Orders` entity object by ID.

2. Access the associated entity using the accessor attribute.

   Using the attribute getter method, you can pass in the name of an association accessor and get back the entity object on the other side of the relationship. (Note that How to Change Entity Association Accessor Names, explains that renaming the association accessor allows it to have a more intuitive name.)

3. Return some of its data to the caller.

   For example, the `findOrderAndCustomer()` method uses the getter methods on the returned `Customer` entity to return the assigned customer's name and location.

Notice that you did not need to write any SQL to access the related `Customer` entity. The relationship information captured in the ADF association between the `Orders` and `Customer` entity objects is enough to allow the common task of data navigation to be automated.

The following example shows the code for `findOrderCustomer()` that uses the helper method.

```
/*  Access an associated Customer entity from the Order entity   */
public String findOrderAndCustomer(long orderId) {
  OrdersImpl order = retrieveOrderById(orderId);
  if (order != null) {
    CustomerImpl cust = order.getCustomer();
    if (cust != null) {
      return "Customer: " + cust.getName() + ", Location: " + cust.getCity();
    }
    else {
      return "Unassigned";
    }
  }
  else {
    return null;
  }
}
```

How to Update or Remove an Existing Entity Row

Once you've got an entity row in hand, it's simple to update it or remove it. For example, you could add a method like `updateOrderStatus()` shown in the sample code below to perform the task.

Before you begin:

It may be helpful to have an understanding of when to use a programmatic approach for working with entity objects and associations. For more information, see Working Programmatically with Entity Objects and Associations.

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Additional Functionality for Entity Objects.

To update an entity row:

1. Find the order by ID.

   Using the `retrieveOrderById()` helper method, the `updateOrderStatus()` method retrieves the `Orders` entity object by Id.

2. Set one or more attributes to new values.

   Using the `EntityImpl` class' `setAttribute()` method, the `updateOrderStatus()` method updates the value of the `OrderFilled` attribute to the new value passed in.

3. Commit the transaction.

   You will then need to use the application module's `getDBTransaction()` method to access the current transaction object and calls its `commit()` method to commit the transaction.

   ```
   /* Update the status of an existing order */
   public void updateOrderStatus(long orderId, String newStatus) {
     OrdersImpl order = retrieveOrderById(orderId);
     if (order != null) {
       order.setOrderFilled(newStatus);
     }
   }
   ```

The code for *removing* an entity row would be the same, except that after finding the existing entity, you would use the following line to remove the entity before committing the transaction:

```
// Remove the entity instead
order.remove();
```

How to Create a New Entity Row

In addition to using the entity definition to find existing entity rows, you can also use it to create new ones. In the case of customer entities, you could write a `createCustomer()` method like the one shown in the following example to accept the name and description of a new customer, and return the new customer ID assigned to it. This example assumes that the `Id` attribute of the `Customer` entity object has been updated to have the `DBSequence` type (see How to Get Trigger-Assigned Primary Key Values from a Database Sequence). This setting ensures that the attribute value is refreshed to reflect the value of the trigger from the corresponding database table, assigned to it from the table's sequence in the application schema.

```
/* Create a new Customer and Return the new id */
public long createCustomer(String name, String city, Integer countryId) {
  EntityDefImpl customerDef = CustomerImpl.getDefinitionObject();
  CustomerImpl newCustomer =
(CustomerImpl)customerDef.createInstance2(getDBTransaction(),null);
    newCustomer.setName(name);
    newCustomer.setCity(city);
    newCustomer.setCountryId(countryId);
  try {
    getDBTransaction().commit();
  }
  catch (JboException ex) {
```

```
    getDBTransaction().rollback();
    throw ex;
  }
  DBSequence newIdAssigned = newCustomer.getId();
  return newIdAssigned.getSequenceNumber().longValue();
}
```

Before you begin:

It may be helpful to have an understanding of when to use a programmatic approach for working with entity objects and associations. For more information, see Working Programmatically with Entity Objects and Associations.

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Additional Functionality for Entity Objects.

To create an entity row:

1. Find the entity definition.

   Using the `getDefinitionObject()` method, the `createCustomer()` method finds the entity definition for the `Customer` entity.

2. Create a new instance.

   Using the `createInstance2()` method on the entity definition, the `createCustomer()` method creates a new instance of the entity object.

   > **Note:**
   >
   > The method name has a `2` at the end. The regular `createInstance()` method has `protected` access and is designed to be customized as described EntityImpl Class of Most Commonly Used ADF Business Components Methods. The second argument of type `AttributeList` is used to supply attribute values that *must* be supplied at create time; it is *not* used to initialize the values of all attributes found in the list. For example, when creating a new instance of a composed child entity row using this API, you must supply the value of a composing parent entity's foreign key attribute in the `AttributeList` object passed as the second argument. Failure to do so results in an `InvalidOwnerException`.

3. Set attribute values.

   Using the attribute setter methods on the entity object, the `createCustomer()` method assigns values for the `Name`, `City`, and other attributes in the new entity row.

4. Commit the transaction.

   Calling `commit()` on the current transaction object, the `createCustomer()` method commits the transaction.

5. Return the trigger-assigned customer ID to the caller.

   Using the attribute getter method to retrieve the value of the `Id` attribute as a `DBSequence`, and then calling `getSequenceNumber().longValue()`, the `createCustomer()` method returns the sequence number as a `long` value to the caller.

Assigning the Primary Key Value Using an Oracle Sequence

As an alternative to using a trigger-assigned value (as described in How to Get Trigger-Assigned Primary Key Values from a Database Sequence), you can assign the value to a primary key when creating a new row using an Oracle sequence. This metadata-driven approach allows you to centralize the code to retrieve the primary key into a single Java file that can be reused by multiple entity objects.

The following example shows a simple `CustomEntityImpl` framework extension class on which the entity objects are based. Its overridden `create()` method tests for the presence of a custom attribute-level metadata property named `SequenceName` and if detected, populates the attribute's default value from the next number in that sequence.

Before you begin:

It may be helpful to have an understanding of when to use a programmatic approach for working with entity objects and associations. For more information, see Working Programmatically with Entity Objects and Associations.

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Additional Functionality for Entity Objects.

To assign the primary key value using an Oracle sequence:

1. Create the `CustomEntityImpl.java` file in your project, and insert the code shown in the sample below.
2. In the Applications window, double-click the entity you want to edit.
3. In the overview editor, click the **Attributes** navigation tab, and select the attribute you want to edit.
4. Click the **Details** tab and select **Number** from the **Type** dropdown list.
5. Click the **Custom Properties** tab, and click the **Add** icon.
6. Create a custom property with `SequenceName` for the name, and the name of the database sequence for the value.

   For example, for a `Dept` entity, you could define the custom property `SequenceName` on its `Deptno` attribute with the value `DEPT_TABLE_SEQ`.

```
package sample;

import oracle.jbo.AttributeDef;
import oracle.jbo.AttributeList;
import oracle.jbo.server.EntityImpl;
import oracle.jbo.server.SequenceImpl;

public class CustomEntityImpl extends EntityImpl {
    protected void create(AttributeList attributeList) {
        super.create(attributeList);
        for (AttributeDef def : getEntityDef().getAttributeDefs()) {
            String sequenceName = (String)def.getProperty("SequenceName");
            if (sequenceName != null) {
               SequenceImpl s = new 
SequenceImpl(sequenceName,getDBTransaction());
               setAttribute(def.getIndex(),s.getSequenceNumber());
```

```
                }
            }
        }
    }
```

How to Update a Deleted Flag Instead of Deleting Rows

For auditing purposes, once a row is added to a table, sometimes your requirements may demand that rows never be physically deleted from the table. Instead, when the end user deletes the row in the user interface, the value of a DELETED column should be updated from "N" to "Y" to mark it as deleted. You can use two method overrides to alter an entity object's default behavior to achieve this effect.

To accomplish this, you need to perform the following tasks:

- Update a deleted flag when a row is removed, as described in Updating a Deleted Flag When a Row Is Removed.
- Force the entity object to be updated instead of deleted, as described in Forcing an Update DML Operation Instead of a Delete.

Updating a Deleted Flag When a Row Is Removed

To update a deleted flag when a row is removed, enable a custom Java class for your entity object and override the remove() method to set the deleted flag before calling the super.remove() method. The following example shows what this would look like in the custom Java class of an entity object. It is important to set the attribute *before* calling super.remove() since an attempt to set the attribute of a deleted row will encounter the DeadEntityAccessException.

This example presumes that you've altered the table to have an additional DELETED column, and synchronized the entity object with the database to add the corresponding Deleted attribute.

The row will still be removed from the row set, but it will have the value of its Deleted flag modified to "Y" in the entity cache. The second part of implementing this behavior involves forcing the entity to perform an UPDATE instead of an DELETE when it is asked to perform its DML operation. You need to implement both parts for a complete solution.

```
// In your custom Java entity class
public void remove() {
  setDeleted("Y");
  super.remove();
}
```

Forcing an Update DML Operation Instead of a Delete

To force an entity object to be updated instead of deleted, override the doDML() method and write code that conditionally changes the operation flag. When the operation flag equals DML_DELETE, your code will change it to DML_UPDATE instead. The following example shows what this would look like in the custom Java class of an entity object.

This example presumes that you've altered the table to have an additional DELETED column, and synchronized the entity object with the database to add the corresponding Deleted attribute.

With this overridden doDML() method in place to complement the overridden remove() method described in Updating a Deleted Flag When a Row Is Removed, any attempt to remove an entity through any view object with a corresponding entity usage will update the DELETED column instead of physically deleting the row. Of course, in order to prevent "deleted" products from appearing in your view object query results, you will need to appropriately modify their WHERE clauses to include only products WHERE DELETED = 'N'.

```
// In your custom Java entity class
protected void doDML(int operation, TransactionEvent e) {
   if (operation == DML_DELETE) {
     operation = DML_UPDATE;
   }
   super.doDML(operation, e);
  }
```

How to Control Entity Posting Order to Prevent Constraint Violations

Due to database constraints, when you perform DML operations to save changes to a number of related entity objects in the same transaction, the order in which the operations are performed can be significant. If you try to insert a new row containing foreign key references *before* inserting the row being referenced, the database can complain with a constraint violation. You must understand what the default order for the processing of entity objects is during commit time and how to programmatically influence that order when necessary.

> **Note:**
>
> The example in this section refers to the oracle.summit.model.controlpostorder package in the SummitADF_Examples application workspace.

Default Post Processing Order

By default, when you commit the transaction the entity objects in the pending changes list are processed in chronological order, in other words, the order in which the entities were added to the list. This means that, for example, if you create a new employee (EmpEO entity object) and then a new department (DeptEO entity object) related to that product, the new EmpEO will be inserted first and the new DeptEO second.

Compositions and Default Post Processing Order

When two entity objects are related by a *composition*, the strict chronological ordering is modified automatically to ensure that composed parent and child entity rows are saved in an order that prevents violating any constraints. This means, for example, that a new parent entity row is inserted before any new composed children entity rows.

Overriding postChanges() to Control Post Order

If your related entities are associated but *not* composed, then you need to write a bit of code to ensure that the related entities get saved in the appropriate order.

Observing the Post Ordering Problem First Hand

Consider the `newEmployeeForNewDepartment()` custom method from the `AppModule` application module in the following example. It accepts a set of parameters and:

1. Creates a new employee.
2. Creates a new department.
3. Sets the department ID to which the employee pertains.
4. Commits the transaction.
5. Constructs a `Result` Java bean to hold new employee ID and department ID.
6. Returns the result.

> **Note:**
>
> The code makes the assumption that both `EmpEO.Id` and `DeptEO.Id` have been set to have `DBSequence` data type to populate their primary keys based on a sequence.

```
// In AppModuleImpl.java
public Result newEmployeeForNewDepartment(String deptName,
                                           Number regionId,
                                           String lastName,
                                           String firstName,
                                           Number salary)  {

   oracle.jbo.domain.Date today = new Date(Date.getCurrentDate());
   Number objectId = new Number(0);

   // 1. Create a new employee
   EmpEOImpl newEmp = createNewEmp();
   // 2. Create a new department
   DeptEOImpl newDept = createNewDept();
   newDept.setName(deptName);
   newDept.setRegionId(regionId);
   // 3. Set the department id to which the employee pertains
   newEmp.setDeptId(newDept.getId().getSequenceNumber());
   newEmp.setLastName(lastName);
   newEmp.setFirstName(firstName);
   newEmp.setUserid((firstName + "." + lastName).substring(8));
   newEmp.setSalary(salary);
   // 4. Commit the transaction
   getDBTransaction().commit();
   // 5. Construct a bean to hold new department id and employee id
   Result result = new Result();
   result.setEmpId(newEmp.getId().getSequenceNumber());
   result.setDeptId(newDept.getId().getSequenceNumber());
   // 6. Return the result
   return result;
}
private DeptEOImpl createNewDept(){
   EntityDefImpl deptDef = DeptEOImpl.getDefinitionObject();
   return (DeptEOImpl) deptDef.createInstance2(getDBTransaction(), null);
}
```

```
private EmpEOImpl createNewEmp(){
   EntityDefImpl empDef = EmpEOImpl.getDefinitionObject();
   return (EmpEOImpl) empDef.createInstance2(getDBTransaction(), null);
}
```

If you add this method to the application module's client interface and test it from a test client program, you get an error:

```
oracle.jbo.DMLConstraintException: JBO-26048:
Constraint "S_EMP_DEPT_ID_FK" is violated during post operation
"Insert" using SQL statement
"BEGIN INSERT INTO S_EMP(ID,LAST_NAME,FIRST_NAME,USERID,DEPT_ID,SALARY)
 VALUES (:1,:2,:3,:4,:5,:6) RETURNING ID INTO :7; END;".
: ORA-02291: integrity constraint (SUMMIT_ADF.S_EMP_DEPT_ID_FK)
violated - parent key not found
```

When the S_EMP row is inserted, the database complains that the value of its DEPT_ID foreign key doesn't correspond to any row in the S_DEPT table. This occurred because:

- The code created the employee before the department.
- The EmpEO and DeptEO entity objects are associated but not composed.
- The DML operations to save the new entity rows is done in chronological order, so the new EmpEO gets inserted before the new DeptEO.

Forcing the Department to Post Before the Employee

To remedy the problem of attempting to add an employee with a not-yet-valid department ID, you could reorder the lines of code in the example to create the DeptEO first, then the EmpEO. While this would address the immediate problem, it still leaves the chance that another application developer could create things in an incorrect order.

The better solution is to make the entity objects *themselves* handle the posting order so the posting will work correctly regardless of the order of creation. To do this, you need to override the postChanges() method in the entity that contains the foreign key attribute referencing the associated entity object and write code as shown in the following example. In this example, since it is the EmpEO that contains the foreign key to the DeptEO entity, you need to update the EmpEO to conditionally force a related, new DeptEO to post before the employee posts itself.

The code tests whether the entity being posted is in the STATUS_NEW or STATUS_MODIFIED state. If it is, it retrieves the related product using the getDeptEO() association accessor. If the related DeptEO also has a post-state of STATUS_NEW, then *first* it calls postChanges() on the related parent row before calling super.postChanges() to perform its own DML.

If you were to run this example, you would see that without changing the creation order in the newEmployeeForNewDepartment() method's code, entities now post in the correct order — first new DeptEO, then new EmpEO. Yet, there is still a problem. The constraint violation still appears, but now for a different reason.

If the primary key for the DeptEO entity object were user-assigned, then the code in the following example would be all that is required to address the constraint violation by correcting the post ordering. In this example, however, the DeptEO.Id is assigned from a database sequence, and not user-assigned. So when a new DeptEO entity row gets posted, its Id attribute is refreshed to reflect the database-assigned sequence value. The foreign key value in the EmpEO.DeptId attribute referencing the new supplier is "orphaned" by this refreshing of the supplier's ID value. When the product's row is saved, its S_DEPT_ID value still doesn't match a row in the S_DEPT table, and the constraint violation occurs again. The next two sections discuss the solution to address this "orphaning" problem.

```
// In EmpEOImpl.java
public void postChanges(TransactionEvent transactionEvent) {
   /* If current entity is new or modified */
   if (getPostState() == STATUS_NEW ||
      getPostState() == STATUS_MODIFIED) {
      /* Get the associated dept for the employee */
      DeptEOImpl dept = getDeptEO();
      /* If there is an associated dept */
      if (dept != null) {
         /* And if it's post-status is NEW */
         if (dept.getPostState() == STATUS_NEW) {
            /*
   * Post the department first, before posting this
   * entity by calling super below
   */
            dept.postChanges(transactionEvent);
         }
      }
   }
   super.postChanges(transactionEvent);
}
```

Note:

An alternative to the programmatic technique discussed here, which solves the problem at the Java EE application layer, is the use of deferrable constraints at the database layer. If you have control over your database schema, consider defining (or altering) your foreign key constraints to be DEFERRABLE INITIALLY DEFERRED. This database setting causes the database to defer checking the constraint until transaction commit time. When this is done, the application can perform DML operations in any order, provided that by COMMIT time all appropriate related rows have been saved and would alleviate the parent/child ordering. However, you would still need to write the code to cascade-update the foreign key values if the parent's primary key is assigned from a sequence, as described in Associations Based on DBSequence-Valued Primary Keys, and Refreshing References to DBSequence-Assigned Foreign Keys.

Associations Based on DBSequence-Valued Primary Keys

Recall from How to Get Trigger-Assigned Primary Key Values from a Database Sequence, that when an entity object's primary key attribute is of DBSequence type, during the transaction in which it is created, its numerical value is a unique, temporary negative number. If you create a number of associated entities in the same transaction, the relationships between them are based on this temporary negative key value. When the entity objects with DBSequence-value primary keys are posted, their primary key is refreshed to reflect the correct database-assigned sequence number, leaving the associated entities that are still holding onto the temporary negative foreign key value "orphaned."

For entity objects based on a *composition*, when the parent entity object's DBSequence-valued primary key is refreshed, the composed children entity rows automatically have their temporary negative foreign key value updated to reflect the owning parent's refreshed, database-assigned primary key. This means that for composed entities, the "orphaning" problem does not occur.

However, when entity objects are related by an association that is not a composition, you need to write a little code to insure that related entity rows referencing the temporary negative number get updated to have the refreshed, database-assigned primary key value. The next section outlines the code required.

Refreshing References to DBSequence-Assigned Foreign Keys

When an entity like DeptEO in this example has a DBSequence-valued primary key, and it is referenced as a foreign key by other entities that are associated with (but not composed by) it, you need to override the postChanges() method as shown in the sample code below to save a reference to the row set of entity rows that might be referencing this new DeptEO row.

If the status of the current DeptEO row is New, then the code assigns the RowSet-valued return of the getEmp() association accessor to the newEmployeesBeforePost member field before calling super.postChanges().

This saved RowSet object is then used by the overridden refreshFKInNewContainees() method shown in the following example. It gets called to allow a new entity row to cascade-update its refreshed primary key value to any other entity rows that were referencing it before the call to postChanges(). It iterates over the EmpEOImpl rows in the newEmployeesBeforePost row set (if non-null) and sets the new department ID value of each one to the new sequence-assigned supplier value of the newly posted DeptEO entity.

```
// In DeptEOImpl.java
protected void refreshFKInNewContainees() {
   if (newEmployeesBeforePost != null) {
      Number newDeptId = getId().getSequenceNumber();
      /*
       * Process the rowset of employees that referenced
       * the new department prior to posting, and update their
       * Id attribute to reflect the refreshed Id value
       * that was assigned by a database sequence during posting.
       */
      while (newEmployeesBeforePost.hasNext()){
         EmpEOImpl emp =
            (EmpEOImpl)newEmployeesBeforePost.next();
         emp.setDeptId(newDeptId);
      }
      closeNewProductRowSet();
   }
}
```

After implementing this change, the sample in Observing the Post Ordering Problem First Hand runs without encountering any database constraint violations.

```
// In DeptEOImpl.java
RowSet newEmployeesBeforePost = null;
public void postChanges(TransactionEvent transactionEvent) {
   /* Only bother to update references if Department is a NEW one */
   if (getPostState() == STATUS_NEW) {
   /*
```

```
       * Get a rowset of employees related
       * to this new department before calling super
       */
      newEmployeesBeforePost = (RowSet)getEmpEO();
      }
      super.postChanges(transactionEvent);
   }
```

Advanced Entity Association Techniques

This section describes several advanced techniques for working with associations between entity objects.

Modifying Association SQL Clause to Implement Complex Associations

When you need to represent a more complex relationship between entities than one based only on the equality of matching attributes, you can modify the association's SQL clause to include more complex criteria. For example, sometimes the relationship between two entities depends on effective dates. A `Product` may be related to a `Supplier`, but if the name of the supplier changes over time, each row in the `SUPPLIERS` table might include additional `EFFECTIVE_FROM` and `EFFECTIVE_UNTIL` columns that track the range of dates in which that product row is (or was) in use. The relationship between a `Product` and the `Supplier` with which it is associated might then be described by a combination of the matching `SupplierId` attributes and a condition that the product's `RequestDate` lie between the supplier's `EffectiveFrom` and `EffectiveUntil` dates.

You can set up this more complex relationship in the overview editor for the association. First, add any additional necessary attribute pairs on the Relationship page, which in this example would include one (`EffectiveFrom`, `RequestDate`) pair and one (`EffectiveUntil`, `RequestDate`) pair. Then, on the Query page you can edit the **Where** field to change the WHERE clause to be:

```
(:Bind_SupplierId = Product.SUPPLIER_ID) AND
(Product.REQUEST_DATE BETWEEN :Bind_EffectiveFrom
                                 AND :Bind_EffectiveUntil)
```

For more information about creating associations, see Creating and Configuring Associations.

Exposing View Link Accessor Attributes at the Entity Level

When you create a **view link** between two entity-based view objects, on the View Link Properties page, you have the option to expose **view link accessor** attributes both at the view object level as well as at the entity object level. By default, a view link accessor is exposed only at the view object level of the destination view object. By selecting the appropriate **In Entity Object: *SourceEntityName*** or **In Entity Object:*DestinationEntityName*** checkbox, you can opt to have JDeveloper include a view link attribute in either or both of the source or destination entity objects. This can provide a handy way for an entity object to access a set of related view rows, especially when the query to produce the rows depends only on attributes of the current row.

What You May Need to Know About Custom Entity Object Methods

Custom methods that you implement in the entity object class must not be dependent on the return type of an application module. At runtime, in specific cases, methods that execute with such a dependency may throw a `ClassCastException` because the returned application module does not match the expected type. It is therefore recommended that custom methods that you implement should not have code to get a specific application module implementation or view object implementation as shown below.

```
((MyAM)getTransaction().getRootApplicationModule()).getMyVO
```

Specifically, the above code fails with a `ClassCastException` in the following scenarios:

- When your code uses the entity object in the context of a different view object, other that the view object that you reference in the method. Because your application may map more than one view object definition to the same same entity object, the ADF Business Components runtime does not guarantee that methods with dependencies on the view object will execute consistently.
- When you manually nest an application module under a **root application module**. In this case, the nested application modules share the same `Transaction` object and there is no guarantee that the expected application module type is returned with the above code.
- When the ADF Business Components framework implementation changes with releases. For example, in previous releases, the framework created an internal root application module in order to control declarative transactions that the application defined using **ADF task flows**.

Working Programmatically with Custom Data Sources and the Framework Base Class ProgrammaticEntityImpl

ADF Business Components supports programmatic entity objects that you can create to interact with a custom data source.

The framework base class `oracle.jbo.server.ProgrammaticEntityImpl` lets you take control of performing create/delete/update/lock/rollback operations for each row in a custom data source when you generate an `EntityObjectImpl` class for the entity object in your ADF Model project. At different phases of the entity populate, commit, or rollback lifecycle, the framework will call these hook points and the framework base implementation class gives the application developer control to interact with the custom data source and perform these operations.

> **Note:**
>
> Prior to the `ProgrammaticEntityImpl` base class, applications extended from the base class `oracle.jbo.server.EntityImpl` and overrode a number of lifecycle methods for custom behavior. Using the programmatic entity base class means integration with a custom data source is accomplished by the application developer without needing to understand the entire entity lifecycle.

To work with programmatic entity objects in the ADF Model project, you select the **Programmatic** option in the Create Entity Object wizard. Then entity object Java classes that you generate in the wizard will by default extend from the framework base class `oracle.jbo.server.ProgrammaticEntityImpl` instead of extending from the classic style framework class `oracle.jbo.server.EntityImpl`.

> **Note:**
>
> In order to ensure this behavior when creating an entity object, you must de-select the feature in the ADF Business Components Preferences dialog that enables extending from the base class `oracle.jbo.server.EntityImpl`. The option in the ADF Business Components-View Objects page of the Tools-Preferences dialog is **Classic Programmatic View** (de-select to enable extending from `ProgrammaticEntityImpl`).

How to Create an Entity Object Class Extending ProgrammaticEntityImpl

To create a programmatic entity object that extends `oracle.jbo.server.ProgrammaticEntityImpl`, you use the Create Entity Object wizard and select the **Programmatic** data source option.

Before you begin:

You must disable classic style programmatic entity object generation. In the Tools-Preferences dialog, de-select the option **Classic Programmatic View** on the ADF Business Components-View Objects page. If you leave this option selected (default), JDeveloper will generate entity object Java classes that extend `oracle.jbo.server.EntityImpl`. For details about classic mode, see Using Classic Style Programmatic View Objects for Alternative Data Sources.

To create programmatic entity objects extending ProgrammaticEntityImpl:

1. In the Applications window, right-click the project in which you want to create the entity object and choose **New**.
2. In the New Gallery, expand **Business Tier**, select **ADF Business Components** and then **Entity Object**, and click **OK**.
3. In the Create Entity Object wizard, in the Name page, provide a name and package for the entity object. For the data source, select **Programmatic**.

4. In the Attributes page, click **New** one or more times to define the entity object attributes your programmatic entity object requires.
5. In the Attribute Settings page, adjust any setting you may need to for the attributes you defined.
6. In the Java page, select **Generate Entity Object Class** to enable a custom entity object class (`ProgrammaticEntityImpl`) to contain your code.
7. Click **Finish** to create the entity object.

   In your entity object's custom Java class, override the methods described in Key Framework Methods to Override for ProgrammaticEntityImpl Based Entity Objects to implement your custom data retrieval strategy.

Key Framework Methods to Override for ProgrammaticEntityImpl Based Entity Objects

The generated `EntityObjectImpl.java` that you generate for a programmatic entity object has the following methods that you must override.

- `postDataProvider(int operationType , ArrayList retAttrNames)`

  This method will be called during doDML execution. By default, the framework throws an Exception from the base implementation. Possible values of `operationType` argument are `EntityImpl.DML_INSERT` ,`EntityImpl.DML_UPDATE` or `EntityImpl.DML_DELETE`. Based on the operation type, applications should insert/update/delete the row from custom data source (or secondary cache, if maintaining). The `retAttrNames` argument contains the list of attribute names whose value the application should return back after row has been updated/ inserted.

- `getRowFromDataSource(HashMap[String , Object] origPrimaryKeyMap)`

  Applications should override this API to return the latest row data from custom data source. During entity fault-in, commit or re-populate, the framework needs to know the current row data to handle a multiple user access use case. The origPrimaryKeyMap argument will contain the original primary key information, even if the application has modified any of the key attributes.

- `lockDataProvider(boolean lock)`

  Applications should override this API to lock/unlock the corresponding row in custom data source. Similar to the database behavior, the framework will try to ensure the locking of custom row `DataProvider` to avoid multiple users committing simultaneously. In case the row is already locked, the application should throw `oracle.jbo.AlreadyLockedException` exception.

- `rollbackDataProvider()`

  If custom `DataSource` is also maintaining intermediate caching, applications can override this API to perform a rollback. The framework will invoke this API if any Exception occurs during the commit/postChanges phase.

- `commitDataProvider(ProgrammaticEntityImpl.CommitActionType commitActionType)`

  If custom `DataSource` is also maintaining intermediate caching, applications should merge the cache with original data by overriding this API. Possible values for `commitActionType` argument are:

  - `CommitActionType.DELETE` - Delete the row from cache and original custom DS
  - `CommitActionType.MERGE` - Merge the row from cache to original custom DS
  - `CommitActionType.IGNORE` - Ignore this cached row and perform NO action for original row

Other Useful Protected APIs

- `getPrimaryKeyMap(boolean orig)`

  Based on `orig` argument, this API returns the values map of the Primary key. If `orig` is TRUE, it will return the primary key values as read from the data source, else the current primary key values.

- `getChangedAttributeMap()`

  This API return the HashMap of all the changed attributes.

What You May Need to Know About Programmatic Entity Object Triggers

You can use triggers defined in the Groovy scripting language in place of overridden methods of `ProgrammaticEntityObjectImpl.java` to perform CUD operations on the custom datasource. Triggers can be called from various programmatic entity object lifecycle points.

The application allows you to call triggers that are defined in Groovy script in place of Java hook points. These triggers interact with a custom data source and perform operations such as creation, updation and deletion of data. They can be called from the various programmatic entity object lifecycle points. Although the option of subclassing the framework class, namely `ProgrammaticEntityImpl` and overriding certain methods to define implementations specific to a custom datasource remains, in order for the trigger to execute, the Java hook points should not be defined. Alternatively, if Java hook points are defined, then the triggers shouldn't exist for the Java hook points to execute. Note that if Java hook points as well as triggers are defined, then the triggers will not execute.

Specifically, the programmatic entity object triggers are:

`InsertData`

This trigger is used to create a new record in the custom datasource. It is invoked in place of the `postDataProvider()` method of `ProgrammaticEntityObjectImpl.java` during an insert operation. Use this trigger to optionally retrieve a map of attribute names and their associated values after performing the insert operation. Use this trigger to perform the insert operation in the custom data source and subsequently populate attribute values in the entity object.

The following code sample illustrates the usage of this trigger. The code sample utilizes these context specific keywords:

- `dmlFilter.key` — Returns an `oracle.jbo.Key` containing the key values of the object to be inserted.
- `dmlFilter.changeMap` – Returns a map containing the name and value pairs of attributes that were changed during the insert operation.

- returnMap - The map that is to be populated with attribute values from the data source after performing the insert operation. The map is pre-populated with keys corresponding to attribute names marked with RetrievedOnInsert=true.

```
@TriggerExpression(triggerType="InsertData", name="InsertData_Rule_0")
def InsertData_Rule_0_ValidationRuleScript_Trigger()
{
   def keyValue = dmlFilter.key.keyValues[0]
   def newEmp = new model.data.EmpObject()
   if (dmlFilter.changeMap.size() > 0)
   {
      def mapIter = dmlFilter.changeMap.keySet().iterator()
      while(mapIter.hasNext())
      {
         def attrName = mapIter.next()
         newEmp.setAttribute(attrName, dmlFilter.changeMap.get(attrName))
      }
   }
   model.data.DataCenter.getInstance().getEmpData().add(newEmp)

   if(returnMap.size() > 0)
   {
      def retMapIter = returnMap.keySet().iterator()
      while (retMapIter.hasNext())
      {
         String attrName = retMapIter.next()
         returnMap.put(attrName, newEmp.getAttribute(attrName))
      }
   }
}
```

UpdateData

This trigger is used to update an existing record in the custom datasource. It is invoked in place of the postDataProvider() method of ProgrammaticEntityObjectImpl.java during an update operation. You can optionally use this trigger to retrieve a map of name and value pairs of attributes after performing the update operation. This trigger populates the entity object with the returned name and value pairs of these attributes.

The following code sample illustrates the usage of this trigger. The code sample utilizes these context specific keywords:

- dmlFilter.key - Returns an oracle.jbo.Key containing the key values of the object to be updated.
- dmlFilter.changeMap - Returns a map containing the names and values of the attributes that were changed during the update operation.
- returnMap - The map that is to be populated with attribute values from the datasource after performing the update operation. The map is pre-populated with keys corresponding to attribute names marked with RetrievedOnUpdate=true

```
@TriggerExpression(triggerType="UpdateData", name="UpdateData_Rule_0")
def UpdateData_Rule_0_ValidationRuleScript_Trigger()
{
   def keyValue = dmlFilter.key.keyValues[0]
   def empData = model.data.DataCenter.getInstance().getEmpData()
   model.data.EmpObject empToUpdate = null
   for (emp in empData)
   {
      if (emp.empno.equals(keyValue))
```

```
            {
               //Found the EmpObject to update.
               empToUpdate = emp
               break
            }
         }
         if (dmlFilter.changeMap.size() > 0)
         {
            def mapIter = dmlFilter.changeMap.keySet().iterator()
            while(mapIter.hasNext())
            {
               def attrName = mapIter.next()
               empToUpdate.setAttribute(attrName, dmlFilter.changeMap.get(attrName))
            }
         }

         if(returnMap.size() > 0)
         {
            def retMapIter = returnMap.keySet().iterator()
            while (retMapIter.hasNext())
            {
               String attrName = retMapIter.next()
               returnMap.put(attrName, empToUpdate.getAttribute(attrName))
            }
         }
      }
```

`DeleteData`

This trigger is used to delete an existing record in the custom datasource. It is invoked in place of the `postDataProvider()` method of the `ProgrammaticEntityObjectImpl.java` class during a delete operation.

The following code sample illustrates the usage of this trigger. This code utilizes this context specific keyword:

- `dmlFilter.key` - Returns an `oracle.jbo.Key` containing the key values of object to be deleted.

```
@TriggerExpression(triggerType="DeleteData", name="DeleteData_Rule_0")
def DeleteData_Rule_0_ValidationRuleScript_Trigger()
{
   def keyValue = dmlFilter.key.keyValues[0]
   def empData = model.data.DataCenter.getInstance().getEmpData()
   def iter = empData.iterator()
   while (iter.hasNext())
   {
      def emp = iter.next();
      if (emp.empno.equals(keyValue))
      {
         //Found the EmpObject to delete
         iter.remove()
         break;
      }
   }
}
```

Defining Triggers for Programmatic Entity Objects

ADF Business Components allows you to define and apply triggers in a programmatic entity object’s lifecycle. This process is similar to the process to define triggers in normal entity objects. You can use Groovy script to implement business rules that are executed in response to entity-level triggers.

To add a trigger for a programmatic entity object:

1. In the Applications Navigator, double-click the entity object you want to add a trigger to.
2. In the overview editor, click the **Business Rules** navigation tab.
3. On the Business Rules page, select the Entity-Level Triggers node, and click the **Create New Trigger** icon.
4. Select the appropriate trigger point from the **Type** dropdown list.
5. Enter a Groovy expression.

When you create a trigger, a `<trigger>` tag is added to the entity object's XML file, and the `.bcs` file associated with the view object is populated .

Creating Custom, Validated Data Types Using Domains

Encapsulate common validation logic in custom data types, called as ADF Domains, which simplifies application maintenance and puts validation in a single place.

When you find yourself repeating the same sanity-checking validations on the values of similar attributes across multiple entity objects, you can save yourself time and effort by creating your own data types that encapsulate this validation. For example, imagine that across your business domain layer there are numerous entity object attributes that store strings that represent email addresses. One technique you could use to ensure that end users always enter a valid email address everywhere one appears in your business domain layer is to:

- Use a basic `String` data type for each of these attributes
- Add an attribute-level method validator with Java code that ensures that the `String` value has the format of a valid email address for each attribute

However, these approaches can become tedious quickly in a large application. Fortunately, ADF Business Components offers an alternative that allows you to create your own `EmailAddress` data type that represents an email address. After centralizing all of the sanity-checking regarding email address values into this new custom data type, you can use the `EmailAddress` as the type of every attribute in your application that represents an email address. By doing this, you make the intention of the attribute values more clear to other developers and simplify application maintenance by putting the validation in a single place. ADF Business Components calls these developer-created data types **domains**.

Domains are Java classes that extend the basic data types like `String`, `Number`, and `Date` to add constructor-time validation to ensure the candidate value passes relevant sanity checks. They offer you a way to define custom data types with cross-cutting behavior such as basic data type validation, formatting, and custom metadata properties in a way that is inherited by any entity objects or view objects that use the domain as the Java type of any of their attributes.

> **Note:**
>
> The example in this section refers to the `oracle.summit.model.domains` package in the `SummitADF_Examples` application workspace.

How to Create a Domain

To create a domain, use the Create Domain wizard. This wizard is available from the New Gallery in the **ADF Business Components** category.

Before you begin:

It may be helpful to have an understanding of using domains. For more information, see Creating Custom, Validated Data Types Using Domains.

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Additional Functionality for Entity Objects.

To create a domain:

1. In the Applications window, right-click the project for which you want to create a domain and choose **New > From Gallery**.
2. In the New Gallery, expand **Business Tier**, select **ADF Business Components** and then **Domain**, and click **OK**.
3. In the Create Domain wizard, on the Name page, specify a name for the domain and a package in which it will reside. To create a domain based on a simple Java type, leave **Domain for an Oracle Object Type** unselected.
4. Click **Next**.
5. On the Settings page, indicate the base type for the domain and the database column type to which it will map.

   For example, if you were creating a domain called `ShortEmailAddress` to hold eight-character short email addresses, you would set the base type to `String` and the **Database Column Type** to `VARCHAR2(8)`. You can set other common attribute settings on this panel as well.
6. Click **Finish**.

What Happens When You Create a Domain

When you create a domain, JDeveloper creates its XML document in the subdirectory of your project's source path that corresponds to the package name you chose. For example, if you created the `ShortEmailAddress` domain in the `summit.model.domains` package, JDeveloper would create the `ShortEmailAddress.xml` file in the `./summit/model/domains` subdirectory. A domain always has a corresponding Java class, which JDeveloper creates in the `common` subpackage of the package where the domain resides. This means it would create the `ShortEmailAddress.java` class in the `summit.model.domains.common` package. The domain's Java class is generated with the appropriate code to behave in a way that is identical to one of the built-in data types.

What You May Need to Know About Domains

Domains can be created as a variety of different types, and have different characteristics than standard attributes. The sections that follow describe some of the things you may need to know about when working with domains.

Domains as Entity and View Object Attributes

After you've created a domain in a project, it appears among the list of available data types in the **Attribute Type** dropdown list in the entity object and view object wizards and dialogs. To use the domain as the type of a given attribute, just pick it from the list.

> **Note:**
>
> The entity-mapped attributes in an entity-based view object inherit their data type from their corresponding underlying entity object attribute. Therefore, if the entity attribute uses a domain type, the matching view object attribute will as well. For transient or SQL-derived view object attributes, you can directly set the type to use a domain since it is not inherited from any underlying entity.

DataCreationException in Custom validate() Method

Typically, the only coding task you need to do for a domain is to write custom code inside the generated `validate()` method. Your implementation of the `validate()` method should perform your sanity checks on the candidate value being constructed, and throw a `DataCreationException` in the `oracle.jbo` package if the validation fails.

In order to throw an exception message that is translatable, you can create a message bundle class similar to the one shown in the following example. Create it in the same package as your domain classes themselves. The message bundle returns an array of {*MessageKeyString*,*TranslatableMessageString*} pairs.

> **Note:**
>
> There are additional formats available for resource bundles, such as XLIFF. For more information, see Working with Resource Bundles.

```
package oracle.summit.model.domains;

import java.util.ListResourceBundle;

public class ErrorMessages extends ListResourceBundle {
  public static final String INVALID_SHORTEMAIL = "30002";
  public static final String INVALID_EVENNUMBER = "30003";
  private static final Object[][] sMessageStrings = new String[][] {
      { INVALID_SHORTEMAIL,
        "A valid short email address has no @-sign or dot."},
      { INVALID_EVENNUMBER,
        "Number must be even."}
    };

  /**
   * Return String Identifiers and corresponding Messages
   * in a two-dimensional array.
   */
  protected Object[][] getContents() {
    return sMessageStrings;
  }
}
```

String Domains and String Value Aggregation

Since `String` is a base JDK type, a domain based on a `String` aggregates a `private mData String` member field to hold the value that the domain represents. Then, the class implements the `DomainInterface` expected by the ADF runtime, as well as the `Serializable` interface, so the domain can be used in method arguments or return types of the custom client interfaces of Oracle ADF components.

The following example shows the `validate()` method for a simple `ShortEmailAddress` domain class. It tests to make sure that the `mData` value does not contains an at-sign or a dot, and if it does, then the method throws `DataCreationException` referencing an appropriate message bundle and message key for the translatable error message.

```
public class ShortEmailAddress implements DomainInterface, Serializable {
  private String mData;
  // . . .
  protected void validate() {
  /**Implements domain validation logic and throws a JboException on error. */
    int atpos = mData.indexOf('@');
    int dotpos = mData.lastIndexOf('.');
    if (atpos > -1 || dotpos > -1) {
      throw new DataCreationException(ErrorMessages.class,
      ErrorMessages.INVALID_SHORTEMAIL,null,null);
    }
  }
  // . . .
}
```

Simple Domains and Built-In Types

Other simple domains based on a built-in type in the `oracle.jbo.domain` package extend the base type, as shown in the following example. It illustrates the `validate()` method for a simple Number-based domain called `EvenNumber` that represents even numbers.

```
public class EvenNumber extends Number {
  // . . .
  /**
   * Validates that value is an even number, otherwise
   * throws a DataCreationException with a custom
   * error message.
   */
  protected void validate() {
    if (getValue() % 2 == 1) {
      throw new DataCreationException(ErrorMessages.class,
      ErrorMessages.INVALID_EVENNUMBER,null,null);
    }
  }
  // . . .
}
```

Simple Domains As Immutable Java Classes

When you create a simple domain based on one of the basic data types, it is an *immutable* class. That means that once you've constructed a new instance of it like this:

```
ShortEmailAddress email = new ShortEmailAddress("emailaddress1");
```

You cannot change its value. If you want to reference a different short email address, you just construct another one:

```
ShortEmailAddress email = new ShortEmailAddress("emailaddress2");
```

This is not a new concept because it's the same way that `String`, `Number`, and `Date` classes behave, among others.

Creating Domains for Oracle Object Types When Useful

Oracle Database supports the ability to create user-defined types in the database. For example, you could create a type called `POINT_TYPE` using the following DDL statement:

```
create type point_type as object (
   x_coord number,
   y_coord number
);
```

If you use user-defined types like `POINT_TYPE`, you can create domains based on them, or you can reverse-engineer tables containing columns of object type to have JDeveloper create the domain for you. You can use the Create Domain wizard to create Oracle object type domains manually.

To manually create an Oracle object type domain:

1. In the Applications window, right-click the project for which you want to create a domain and choose **New > From Gallery**.
2. In the New Gallery, expand **Business Tier**, select **ADF Business Components** and then **Domain**, and click **OK**.
3. In the Create Domain wizard, on the Name page, select the **Domain for an Oracle Object Type** checkbox, then select the object type for which you want to create a domain from the **Available Types** list.
4. Click **Next**.
5. On the Settings page, use the **Attribute** dropdown list to switch between the multiple domain properties to adjust the settings as appropriate.
6. Click **Finish**.

To reverse-engineer an Oracle object type domain:

In addition to manually creating object type domains, when you use the Business Components from Tables wizard and select a table containing columns of an Oracle object type, JDeveloper creates domains for those object types as part of the reverse-engineering process. For example, imagine you created a table like this with a column of type POINT_TYPE:

```
create table interesting_points(
  id number primary key,
  coordinates point_type,
  description varchar2(20)
);
```

If you create an entity object for the INTERESTING_POINTS table in the Business Components from Tables wizard, then you get both an InterestingPoints entity object and a PointType domain. The latter is generated, based on the POINT_TYPE object type, because it was required as the data type of the Coordinates attribute of the InterestingPoints entity object.

Unlike simple domains, object type domains are mutable. JDeveloper generates getter and setter methods into the domain class for each of the elements in the object type's structure. After changing any domain properties, when you set that domain as the value of a view object or entity object attribute, it is treated as a single unit. Oracle ADF does not track which domain properties have changed, only that a domain-valued attribute value has changed.

> **Note:**
>
> Domains based on Oracle object types are useful for working programmatically with data whose underlying type is an oracle object type. They can also simplify passing and receiving structure information to stored procedures. However, support for working with object type domains in the **ADF binding** layer is not complete, so it is not straightforward to use object domain-valued attributes in declaratively databound user interfaces.

Domains Packaged in the Common JAR

When you create an ADF Library JAR for your business components, as described in How to Package a Component into an ADF Library JAR, the domain classes and message bundle files in the *.common subdirectories of your project's source path get packaged into the *CSCommon.jar. They are classes that are common to both the middle-tier application server and to an eventual remote client you might need to support.

Custom Domain Properties and Attributes in Entity and View Objects

You can define custom metadata properties on a domain. Any entity object or view object attribute based on that domain inherits those custom properties as if they had been defined on the attribute itself. If the entity object or view object attribute defines the same custom property, its setting takes precedence over the value inherited from the domain.

Inherited Restrictive Properties of Domains in Entity and View Objects

JDeveloper enforces the declarative settings you impose at the domain definition level: they cannot be made *less* restrictive for the entity object or view object for an attribute based on the domain type. For example, if you define a domain to have its **Updatable** property set to **While New**, then when you use your domain as the Java type of an entity object attribute, you can set **Updatable** to be **Never** (more restrictive) but you cannot set it to be **Always**. Similarly, if you define a domain to be **Persistent**, you cannot make it transient later. When sensible for your application, set declarative properties for a domain to be as lenient as possible, so you can later make them more restrictive as needed.

Creating New History Types

Use history types to track time-specific data. JDeveloper provides out-of-the-box ADF history types and also allows you to create custom history types.

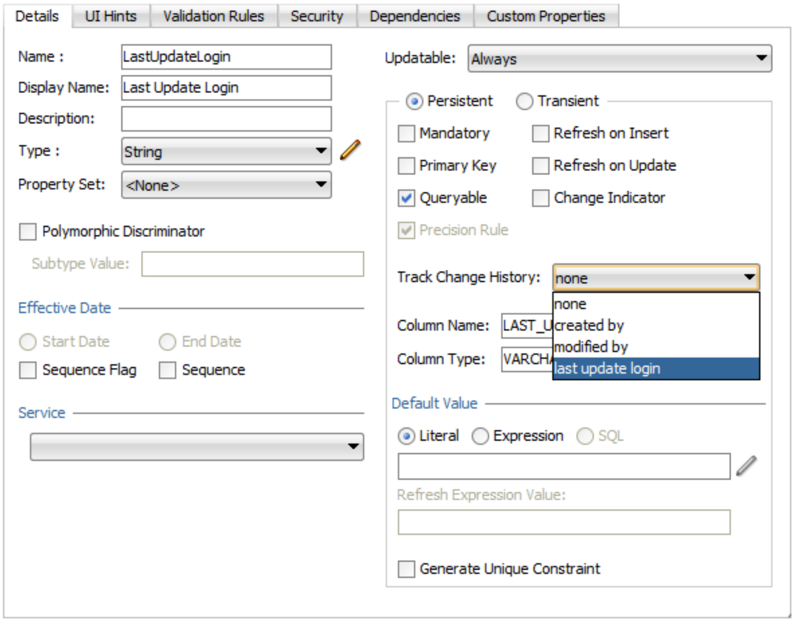
History types are used to track data specific to a point in time. JDeveloper ships with a number of history types, but you can also create your own. For information on the standard history types and how to use them, see How to Track Created and Modified Dates Using the History Column.

How to Create New History Types

You are not limited to the history types provided: you can add or remove custom history types using the History Types page in the Preferences dialog, and then write custom Java code to implement the desired behavior. The code to handle custom history types should be written in your application-wide entity base class for reuse.

Figure 4-17 shows a custom type called `last update login` with type ID of `11`. Assume that `last_update_login` is a foreign key in the `FND_LOGINS` table.

Figure 4-17 New History Types in the Overview Editor



Name : LastUpdateLogin
Display Name: Last Update Login
Description:
Type : String
Property Set: <None>
□ Polymorphic Discriminator
Subtype Value:
Updatable: Always
● Persistent ○ Transient
□ Mandatory □ Refresh on Insert
□ Primary Key □ Refresh on Update
☑ Queryable □ Change Indicator
☑ Precision Rule
Track Change History: none
Column Name: LAST_U
Column Type: VARCHA
○ Start Date ○ End Date
□ Sequence Flag □ Sequence
● Literal ○ Expression ○ SQL
Refresh Expression Value:
□ Generate Unique Constraint

Before you begin:

It may be helpful to have an understanding of history types. For more information, see Creating New History Types.

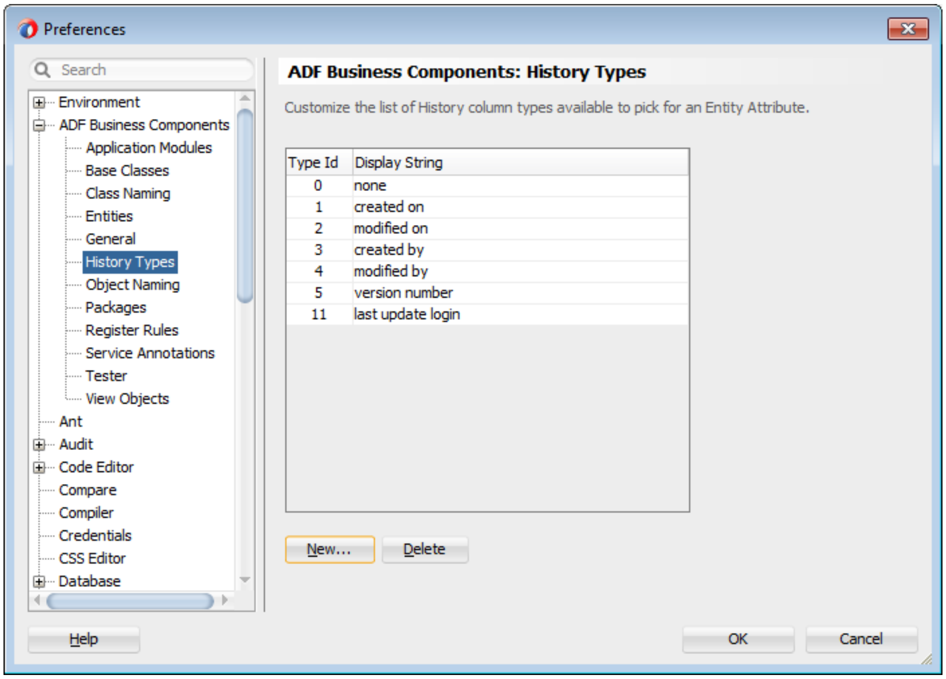
You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Additional Functionality for Entity Objects.

To create a custom history type:

1. From the main menu, choose **Tools > Preferences**.
2. In the Preferences dialog, expand **ADF Business Components** and click **History Types**.
3. In the Preferences dialog, on the History Types page, click **New**.
4. In the New History Type dialog, enter a string value for the name (spaces are allowed) and a numerical ID.

   The **Type Id** must be an integer between 11 and 126. The numerical values 0-10 are reserved for internal use. The display string is displayed in the **Track Change History** dropdown list the next time you use the overview editor. Figure 4-18 shows the new history type in the Preferences dialog.

   Figure 4-18 Custom History Type in Preferences



ADF Business Components: History Types

Customize the list of History column types available to pick for an Entity Attribute.

Type Id	Display String
0	none
1	created on
2	modified on
3	created by
4	modified by
5	version number
11	last update login

5. Open the entity object's Java class file (or the extension class on which it is based) and add a definition similar to the following.

```
private static final byte LASTUPDATELOGIN_HISTORY_TYPE = 11;
```

6. Override the `getHistoryContextForAttribute(AttributeDefImpl attr)` method from the `EntityImpl` base class with code similar the following.

```
@Override
protected Object getHistoryContextForAttribute(AttributeDefImpl attr) {
    if (attr.getHistoryKind() == LASTUPDATELOGIN_HISTORY_TYPE) {
        // Custom History type logic goes here
    }
    else {
        return super.getHistoryContextForAttribute(attr);
        }
    }
```

How to Remove a History Type

Because they are typically used for auditing values over the life of an application, it is rare that you would want to remove a history type. However, in the event that you need to do so, perform the following tasks:

1. Remove the history type from the JDeveloper history types list in the Preferences dialog.
2. Remove any custom code you implemented to support the history type in the `getHistoryContextForAttribute` method in the entity object's Java class file (or the extension class on which it is based).
3. Remove all usages of the history type in the entity attribute metadata. Any attribute that you have defined to use this history type must be edited.

Before you begin:

It may be helpful to have an understanding of history types. For more information, see Creating New History Types.

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Additional Functionality for Entity Objects.

To remove a history type from the JDeveloper history types list:

1. From the main menu, choose **Tools > Preferences**.
2. In the Preferences dialog, expand **ADF Business Components** and click **History Types**.
3. In the Preferences dialog, on the History Types page, select the history type that you want to remove and click **Delete**.

Basing an Entity Object on a PL/SQL Package API

The ADF framework allows you to use the PL/SQL API in combination with a database view. It allows client programs to read using the database view and write to the table using the PL/SQL API.

If you have a PL/SQL package that encapsulates insert, update, and delete access to an underlying table, you can override the default DML processing event for the entity object that represents that table to invoke the procedures in your PL/SQL API instead. Often, such PL/SQL packages are used in combination with a companion database view. Client programs read data from the underlying table using the database view, and "write" data back to the table using the procedures in the PL/SQL package.

For example, say you want to create a `Product` entity object based on such a combination of a view and a package.

Given the `S_PRODUCT` table in the Summit ADF schema, consider a database view named `PRODUCT_V`, created using the following DDL statement:

```
create or replace view product_v
as select id,name,image_id,short_desc from s_product;
```

In addition, consider the simple PRODUCTS_API package shown below that encapsulates insert, update, and delete access to the underlying S_PRODUCT table.

```
create or replace PACKAGE PRODUCTS_API as
  procedure insert_product(p_id number,
                           p_name varchar2,
                           p_short_desc varchar2,
                           p_longtext_id number,
                           p_image_id number,
                           p_suggested_whlsl_price number,
                           p_whlsl_units varchar2);
  procedure update_product(p_id number,
                           p_name varchar2,
                           p_short_desc varchar2,
                           p_longtext_id number,
                           p_image_id number,
                           p_suggested_whlsl_price number,
                           p_whlsl_units varchar2);
  procedure delete_product(p_id number);
end products_api;
```

To create an entity object based on this combination of database view and PL/SQL package, you would perform the following tasks:

1. Create a view-based entity object, as described in How to Create an Entity Object Based on a View.
2. Create a base class for the entity object, as described in How to Centralize Details for PL/SQL-Based Entities into a Base Class.
3. Implement the appropriate stored procedure calls, as described in How to Implement the Stored Procedure Calls for DML Operations.
4. Handle selecting and locking functionality, if necessary, as described in How to Add Select and Lock Handling.

Note:

The example in these sections refers to the oracle.summit.model.wrapplsql package in the SummitADF_Examples application workspace.

Although the example uses an table-based entity object rather than a view-based entity object, the configuration details are congruent.

How to Create an Entity Object Based on a View

To create an entity object based on a view, you use the Create Entity Object wizard.

Before you begin:

It may be helpful to have an understanding of how entity objects can use the PS/SQL API. For more information, see Basing an Entity Object on a PL/SQL Package API.

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Additional Functionality for Entity Objects.

You will need to launch the Create Entity Object wizard as described in How to Create Single Entity Objects Using the Create Entity Wizard, and proceed through the wizard with the exceptions noted in the following procedure.

To create an entity object based on a view:

1. On the Name page, give the entity a name.
2. In the Select Schema Object dialog, in the **Object Type** section, select the **Views** checkbox.

   This enables the display of the available database views in the current schema in when you click **Query**.
3. In the **Available Objects** list, select the desired database view.
4. On the Attribute Settings page, use the **Select Attribute** dropdown list to choose the attribute that will act as the primary key, and then select the **Primary Key** checkbox for that attribute.

   > **Note:**
   >
   > When defining the entity based on a view, JDeveloper cannot automatically determine the primary key attribute since database views do not have related constraints in the database data dictionary.

What Happens When You Create an Entity Object Based on a View

By default, an entity object based on a view performs all of the following directly against the underlying database view:

- `SELECT` statement (for `findByPrimaryKey()`)
- `SELECT FOR UPDATE` statement (for `lock()`), and
- `INSERT`, `UPDATE`, `DELETE` statements (for `doDML()`)

To use stored procedure calls, you will need to override the `doDML()` operations (as described in How to Centralize Details for PL/SQL-Based Entities into a Base Class), and possibly override the `lock()` and `findByPrimaryKey()` handling (as described in How to Implement the Stored Procedure Calls for DML Operations).

How to Centralize Details for PL/SQL-Based Entities into a Base Class

If you plan to have more than one entity object based on a PL/SQL API, it's a smart idea to abstract the generic details into a base framework extension class. In doing this, you'll be using several of the concepts described in Extending Business Components Functionality. Start by creating a `PLSQLEntityImpl` class that extends the base `EntityImpl` class that each one of your PL/SQL-based entities can use as their base class.

> **Note:**
>
> If you are already using an extended entity implementation class for your entity, you can extend it further with the PLSQLEntityImpl class. For example, if you have a framework extension class named zzEntityImpl, you would create a PLSQLEntityImpl class that extends the zzEntityImpl class.

As shown below, you'll override the doDML() method of the base class to invoke a different helper method based on the operation.

```
// In PLSQLEntityImpl.java
protected void doDML(int operation, TransactionEvent e) {
  // super.doDML(operation, e);
  if (operation == DML_INSERT)
    callInsertProcedure(e);
  else if (operation == DML_UPDATE)
    callUpdateProcedure(e);
  else if (operation == DML_DELETE)
    callDeleteProcedure(e);
}
```

In the PLSQLEntityImpl.java base class, you can write the helper methods so that they perform the default processing like this:

```
// In PLSQLEntityImpl.java
/* Override in a subclass to perform nondefault processing */
protected void callInsertProcedure(TransactionEvent e) {
  super.doDML(DML_INSERT, e);
}
/* Override in a subclass to perform nondefault processing */
protected void callUpdateProcedure(TransactionEvent e) {
  super.doDML(DML_UPDATE, e);
}
/* Override in a subclass to perform nondefault processing */
protected void callDeleteProcedure(TransactionEvent e) {
  super.doDML(DML_DELETE, e);
}
```

After putting this infrastructure in place, when you base an entity object on the PLSQLEntityImpl class, you can use the **Source > Override Methods** menu item to override the callInsertProcedure(), callUpdateProcedure(), and callDeleteProcedure() helper methods and perform the appropriate stored procedure calls for that particular entity.

> **Note:**
>
> If you do not override these helper methods in a subclass, they will perform the default processing as defined in the superclass. You only need to override the operations in the doDML() method that you want to provide alternative processing for.

To simplify the task of implementing these calls, you could add the `callStoredProcedure()` helper method (described in Invoking Stored Procedures and Functions) to the `PLSQLEntityImpl` class as well. This way, any PL/SQL-based entity objects that extend this class can leverage the helper method.

How to Implement the Stored Procedure Calls for DML Operations

To implement the stored procedure calls for DML operations, you will need to create a custom Java class for the entity object and override the operations in it.

Before you begin:

It may be helpful to have an understanding of entity objects that are based on a PL/SQL package API. For more information, see Basing an Entity Object on a PL/SQL Package API.

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Additional Functionality for Entity Objects.

To create the custom Java class with the override methods:

1. In the Applications window, double-click the entity object.
2. In the overview editor, click the **Java** navigation tab, and then click the **Edit Java options** icon.
3. In the Select Java Options dialog, click **Classes Extend**.
4. In the Override Base Classes dialog, in the **Row** field, enter the package and class of the `PLSQLEntityImpl` class, or click **Browse** to search and select it.
5. Select **Generate Entity Object Class**, and click **OK**.
6. In the Applications window, double-click the generated entity object class.
7. From the main menu, choose **Source > Override Methods**.
8. In the Override Methods dialog, select the `callInsertProcedure()`, `callUpdateProcedure()`, and `callDeleteProcedure()` methods, and click **OK**.
9. In the source editor, enter the necessary code to override these procedures.

The following example shows some sample code that you would write in these overridden helper methods to invoke insert, update, and delete procedures.

```
// In ProductImpl.java
protected void callInsertProcedure(TransactionEvent e) {
   callStoredProcedure("products_api.insert_product(?,?,?,?,?,?,?)",
      new Object[] { getId(), getName(), getShortDesc(), getLongtextId(),
            getImageId(), getSuggestedWhlslPrice(), getWhlslUnits() });
}
protected void callUpdateProcedure(TransactionEvent e) {
   callStoredProcedure("products_api.update_product(?,?,?,?,?,?)",
      new Object[] { getId(), getName(), getShortDesc(), getLongtextId(),
            getImageId(), getSuggestedWhlslPrice(), getWhlslUnits() });
}
protected void callDeleteProcedure(TransactionEvent e) {
   callStoredProcedure("products_api.delete_product(?)",
      new Object[] { getId() });
}
```

At this point, if you create a default entity-based view object called ProductView for the Product entity object and add an instance of it to a AppModule application module you can quickly test inserting, updating, and deleting rows from the ProductView view object instance in the Oracle ADF Model Tester.

Often, overriding just the insert, update, and delete operations will be enough. The default behavior that performs the SELECT statement for findByPrimaryKey() and the SELECT FOR UPDATE statement for the lock() against the database view works for most basic kinds of views.

However, if the view is complex and does not support SELECT FOR UPDATE or if you need to perform the findByPrimaryKey() and lock() functionality using additional stored procedures APIs, then you can use the technique described in How to Add Select and Lock Handling.

How to Add Select and Lock Handling

You can handle the lock() and findByPrimaryKey() functionality of an entity object by invoking stored procedures if necessary. Imagine that the PRODUCTS_API package were updated to contain the two additional procedures shown here.

```
/* Added to PRODUCTS_API package */
procedure lock_product(p_id number,
                       p_name OUT varchar2,
                       p_short_desc OUT varchar2,
                       p_longtext_id OUT number,
                       p_image_id OUT number,
                       p_suggested_whlsl_price OUT number,
                       p_whlsl_units OUT varchar2);
procedure select_product(p_id number,
                       p_name OUT varchar2,
                       p_short_desc OUT varchar2,
                       p_longtext_id OUT number,
                       p_image_id OUT number,
                       p_suggested_whlsl_price OUT number,
                       p_whlsl_units OUT varchar2);
```

Both the lock_product and select_product procedures accept a primary key attribute as an IN parameter and return values for the remaining attributes using OUT parameters.

To add select and lock handling, you will need to perform the following tasks:

1. Update the base class to handle lock and select, as described in Updating PLSQLEntityImpl Base Class to Handle Lock and Select.
2. Update the entity object implementation class to implement the lock and select behaviors, as described in Implementing Lock and Select for the Product Entity.
3. Override the lock() method in the entity object implementation class to refresh the entity object after a RowInconsistentException has occurred, as described in Refreshing the Entity Object After RowInconsistentException.

Updating PLSQLEntityImpl Base Class to Handle Lock and Select

You can extend the PLSQLEntityImpl base class to handle the lock() and findByPrimaryKey() overrides using helper methods similar to the ones you added for insert, update, delete. At runtime, both the lock() and findByPrimaryKey()

operations invoke the lower-level entity object method called doSelect(boolean lock). The lock() operation calls doSelect() with a true value for the parameter, while the findByPrimaryKey() operation calls it passing false instead.

The following example shows the overridden doSelect() method in PLSQLEntityImpl to delegate as appropriate to two helper methods that subclasses can override as necessary.

```
// In PLSQLEntityImpl.java
protected void doSelect(boolean lock) {
  if (lock) {
    callLockProcedureAndCheckForRowInconsistency();
  } else {
    callSelectProcedure();
  }
}
```

The two helper methods are written to just perform the default functionality in the base PLSQLEntityImpl class:

```
// In PLSQLEntityImpl.java
/* Override in a subclass to perform nondefault processing */
protected void callLockProcedureAndCheckForRowInconsistency() {
  super.doSelect(true);
}
/* Override in a subclass to perform nondefault processing */
protected void callSelectProcedure() {
  super.doSelect(false);
}
```

Notice that the helper method that performs locking has the name callLockProcedureAndCheckForRowInconsistency(). This reminds you that you need to perform a check to detect at the time of locking the row whether the newly selected row values are the same as the ones the entity object in the entity cache believes are the current database values.

To assist subclasses in performing this old-value versus new-value attribute comparison, you can add one final helper method to the PLSQLEntityImpl class like this:

```
// In PLSQLEntityImpl
protected void compareOldAttrTo(int attrIndex, Object newVal) {
  if ((getPostedAttribute(attrIndex) == null && newVal != null) ||
      (getPostedAttribute(attrIndex) != null && newVal == null) ||
      (getPostedAttribute(attrIndex) != null && newVal != null &&
       !getPostedAttribute(attrIndex).equals(newVal))) {
    throw new RowInconsistentException(getKey());
  }
}
```

Implementing Lock and Select for the Product Entity

With the additional infrastructure in place in the base PLSQLEntityImpl class, you can override the callSelectProcedure() and callLockProcedureAndCheckForRowInconsistency() helper methods in the entity object implementation class (for example, ProductImpl). Because the select_product and lock_product procedures have OUT arguments, as described in How to Call Other Types of Stored Procedures, you need to use a JDBC CallableStatement object to perform these invocations.

The example below shows the code you would use to invoke the `select_product` procedure for the `ProductImpl` entity object implementation class to select a row by primary key. It's performing the following basic steps:

1. Creating a `CallableStatement` for the PLSQL block to invoke.
2. Registering the `OUT` parameters and types, by one-based bind variable position.
3. Setting the `IN` parameter value.
4. Executing the statement.
5. Retrieving the possibly updated column values.
6. Populating the possibly updated attribute values in the row.
7. Closing the statement.

```
// In ProductImpl.java
protected void callSelectProcedure() {
   String stmt = "begin products_api.select_product(?,?,?,?,?,?,?);end;";
   // 1. Create a CallableStatement for the PLSQL block to invoke
   CallableStatement st = getDBTransaction().createCallableStatement(stmt, 0);
   try {
      // 2. Register the OUT parameters and types
      st.registerOutParameter(2, VARCHAR2);
      st.registerOutParameter(3, VARCHAR2);
      st.registerOutParameter(4, NUMBER);
      st.registerOutParameter(5, NUMBER);
      st.registerOutParameter(6, NUMBER);
      st.registerOutParameter(7, VARCHAR2);
      // 3. Set the IN parameter value
      st.setObject(1, getId());
      // 4. Execute the statement
      st.executeUpdate();
      // 5. Retrieve the possibly updated column values
      String possiblyUpdatedName = st.getString(2);
      String possiblyUpdatedShortDesc = st.getString(3);
      String possiblyUpdatedLongTextId = st.getString(4);
      String possiblyUpdatedImageId = st.getString(5);
      String possiblyUpdatedSuggestedPrice = st.getString(6);
      String possiblyUpdatedWhlslUnits = st.getString(7);
      // 6. Populate the possibly updated attribute values in the row
      populateAttribute(NAME, possiblyUpdatedName, true, false,
                        false);
      populateAttribute(SHORTDESC, possiblyUpdatedShortDesc, true,
                        false, false);
      populateAttribute(LONGTEXTID, possiblyUpdatedLongTextId, true,
                        false, false);
      populateAttribute(IMAGEID, possiblyUpdatedImageId, true, false,
                        false);
      populateAttribute(SUGGESTEDWHLSLPRICE, possiblyUpdatedSuggestedPrice,
                        true, false, false);
      populateAttribute(WHLSLUNITS, possiblyUpdatedWhlslUnits, true,
                        false, false);
   } catch (SQLException e) {
      throw new JboException(e);
   } finally {
      if (st != null) {
         try {
            // 7. Closing the statement
            st.close();
         } catch (SQLException e) {
         }
      }
   }
}
```

The following example shows the code to invoke the `lock_product` procedure. It's doing basically the same steps as those above, with just the following two interesting differences:

- After retrieving the possibly updated column values from the `OUT` parameters, it uses the `compareOldAttrTo()` helper method inherited from the `PLSQLEntityImpl` to detect whether or not a `RowInconsistentException` should be thrown as a result of the row lock attempt.
- In the `catch (SQLException e)` block, it is testing to see whether the database has thrown the error:

  ```
  ORA-00054: resource busy and acquire with NOWAIT specified
  ```

  and if so, it again throws the ADF Business Components `AlreadyLockedException` just as the default entity object implementation of the `lock()` functionality would do in this situation.

With these methods in place, you have and entity object that wraps the PL/SQL package (in this case, a `Product` entity object with the `PRODUCTS_API` package) for all of its database operations. Due to the clean separation of the data querying functionality of view objects and the data validation and saving functionality of entity objects, you can now leverage this entity object in any way you would use a normal entity object. You can build as many different view objects as necessary that use this entity object as their entity usage.

```
// In ProductsImpl.java
protected void callLockProcedureAndCheckForRowInconsistency() {
   String stmt = "begin products_api.lock_product(?,?,?,?,?,?,?);end;";
   CallableStatement st =
      getDBTransaction().createCallableStatement(stmt, 0);
   try {
      st.registerOutParameter(2, VARCHAR2);
      st.registerOutParameter(3, VARCHAR2);
      st.registerOutParameter(4, NUMBER);
      st.registerOutParameter(5, NUMBER);
      st.registerOutParameter(6, NUMBER);
      st.registerOutParameter(7, VARCHAR2);
      st.setObject(1, getId());
      st.executeUpdate();
      String possiblyUpdatedName = st.getString(2);
      String possiblyUpdatedShortDesc = st.getString(3);
      String possiblyUpdatedLongTextId = st.getString(4);
      String possiblyUpdatedImageId = st.getString(5);
      String possiblyUpdatedSuggestedPrice = st.getString(6);
      String possiblyUpdatedWlhslUnits = st.getString(7);
      compareOldAttrTo(NAME, possiblyUpdatedName);
      compareOldAttrTo(SHORTDESC, possiblyUpdatedShortDesc);
      compareOldAttrTo(LONGTEXTID, possiblyUpdatedLongTextId);
      compareOldAttrTo(IMAGEID, possiblyUpdatedImageId);
      compareOldAttrTo(SUGGESTEDWHLSLPRICE, possiblyUpdatedSuggestedPrice);
      compareOldAttrTo(WHLSLUNITS, possiblyUpdatedWlhslUnits);
   } catch (SQLException e) {
      if (Math.abs(e.getErrorCode()) == 54) {
         throw new AlreadyLockedException(e);
      } else {
```

```
            throw new JboException(e);
         }
      } finally {
         if (st != null) {
            try {
               st.close();
            } catch (SQLException e) {
            }
         }
      }
   }
```

Refreshing the Entity Object After RowInconsistentException

You can override the `lock()` method to refresh the entity object after a `RowInconsistentException` has occurred. The following example shows code that can be added to the entity object implementation class to catch the `RowInconsistentException` and refresh the entity object.

```
// In the entity object implementation class
@Override
public void lock() {
  try {
    super.lock();
  }
  catch (RowInconsistentException ex) {
    this.refresh(REFRESH_UNDO_CHANGES);
    throw ex;
  }
}
```

Basing an Entity Object on a Join View or Remote DBLink

Some types of schema do not support the `RETURNING` clause. So disable this clause in the ADF entity object. The entity object then implements `Refresh on Insert`and `Refresh on Update`.

If you need to create an entity object based on either of the following:

- Synonym that resolves to a remote table over a `DBLINK`
- View with `INSTEAD OF` triggers

Then you will encounter the following error if any of its attributes are marked as **Refresh on Insert** or **Refresh on Update**:

```
JBO-26041: Failed to post data to database during "Update"
## Detail 0 ##
ORA-22816: unsupported feature with RETURNING clause
```

These types of schema objects do not support the `RETURNING` clause, which by default the entity object uses to more efficiently return the refreshed values in the same database roundtrip in which the `INSERT` or `UPDATE` operation was executed.

How to Disable the Use of the RETURNING Clause

Because some types of schema objects do not support the `RETURNING` clause, you might need to disable the `RETURNING` clause in your entity object. The following procedures explains how to do that.

Before you begin:

It may be helpful to have an understanding of the types of schema objects don't support the `RETURNING` clause. For more information, see Basing an Entity Object on a Join View or Remote DBLink.

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Additional Functionality for Entity Objects.

To disable the use of the RETURNING clause for an entity object of this type:

1. Enable a custom entity definition class for the entity object.
2. In the custom entity definition class, override the `createDef()` method to call `setUseReturningClause(false)`.
3. If the **Refresh on Insert** attribute is the primary key of the entity object, you must specify some other attribute in the entity as an alternate unique key by setting the **Unique Key** property on it.

What Happens at Runtime: Refresh Behavior With Disabled RETURNING Clause

At runtime, when you have disabled the use of the `RETURNING` clause as described in How to Disable the Use of the RETURNING Clause,, the entity object implements the **Refresh on Insert** and **Refresh on Update** behavior using a separate `SELECT` statement to retrieve the values to refresh after insert or update as appropriate.

Using Inheritance in Your Business Domain Layer

ADF Business Components supports inheritance to allow you to create new components by extending existing ones.

Inheritance is a powerful feature of object-oriented development that can simplify development and maintenance when used appropriately. As shown in Creating Extended Components Using Inheritance, ADF Business Components supports using inheritance to create new components that extend existing ones in order to add additional properties or behavior or modify the behavior of the parent component. Inheritance can be useful in modeling the different kinds of entities in your reusable business domain layer.

Note:

The example in this section is an extension of the oracle.summit.model.polymorphic package in the SummitADF_Examples workspace.

Understanding When Inheritance Can Be Useful

Your application's database schema might contain tables where different logical kinds of business information are stored in rows of the same table. These tables will typically have one column whose value determines the kind of information stored in each row. For example, an application's S_CUSTOMER table stores information about both domestic and international customers in the same table. It contains a CUSTOMER_TYPE_CODE column whose value determines what kind of S_CUSTOMER the row represents.

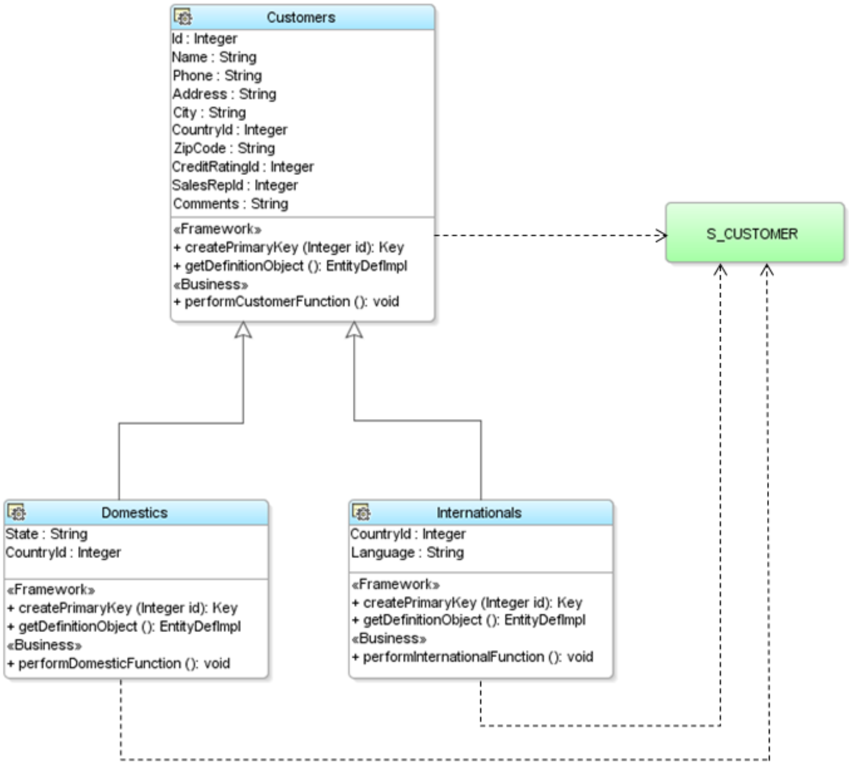
While the Summit ADF sample application implementation doesn't contain this differentiation, it's reasonable to assume that revisions of the application might require:

- Managing additional database-backed attributes that are specific to domestic customers or specific to international customers
- Implementing common behavior for all users that is different for domestic or international customers
- Implementing new functionality that is specific to only domestic or only international customers

Figure 4-19 shows what the business domain layer would look like if you created distinct Customers, Domestics, and Internationals entity objects to allow distinguishing the different kinds of business information in a more formal way inside your application. Since domestic and international are special *kinds* of customers, their corresponding entity objects would extend the base Customers entity object. This base Customers entity object would contain all of the attributes and methods common to all types of users. The performCustomerFunction() method in the figure represents one of these common methods.

Then, for the Domestics and Internationals entity objects you can add specific additional attributes and methods that are unique to that kind of user. For example, Domestics has an additional State attribute of type String to track the state in which the domestic customer is located. There is also a performDomesticFunction() method that is specific to domestic customers. Similarly, the Internationals entity object has an additional Language attribute to track whether the customer speaks English. The performInternationalFunction() is a method that is specific to international customers.

Figure 4-19 Distinguishing Customers, Domestics, and Internationals Using Inheritance



By modeling these different kinds of customers as distinct entity objects in an inheritance hierarchy in your domain business layer, you can simplify having them share common data and behavior and implement the aspects of the application that make them distinct.

How to Create Entity Objects in an Inheritance Hierarchy

To create entity objects in an inheritance hierarchy, you use the Create Entity Object wizard to create each entity.

To create entity objects in an inheritance hierarchy, you will perform the following tasks:

1. Identify the discriminator column and values, as described in Identifying the Discriminator Column and Distinct Values.
2. Identify the subset of attributes for each entity object, as described in Identifying the Subset of Attributes Relevant to Each Kind of Entity.
3. Create the base entity object, as described in Creating the Base Entity Object in an Inheritance Hierarchy.
4. Create the subtype entity objects, as described in Creating a Subtype Entity Object in an Inheritance Hierarchy.

Identifying the Discriminator Column and Distinct Values

Before creating entity objects in an inheritance hierarchy based on a table containing different kinds of information, you should first identify which column in the table is used to distinguish which kind of row it is.

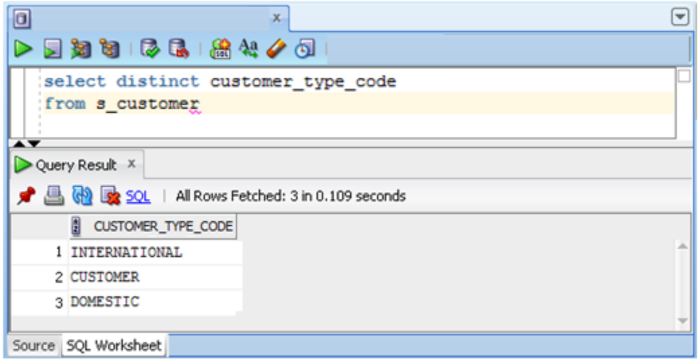
For example, in a `S_CUSTOMER` table, this might be a `CUSTOMER_TYPE_CODE` column. Since it helps partition or "discriminate" the rows in the table into separate groups, this column is known as the **discriminator column**.

Next, determine the valid values that the discriminator column takes on in your table. You might know this, or you could execute a simple SQL statement in the JDeveloper **SQL Worksheet** to determine the answer. To access the worksheet:

1. With the application open in JDeveloper, choose **Window > Database > Databases** from the main menu.
2. Expand the workspace node, and select the connection.
3. Right-click the database connection, and choose **Open SQL Worksheet**.

Figure 4-20 shows the results of performing a `SELECT DISTINCT` query in the SQL Worksheet on the `CUSTOMER_TYPE_CODE` column in the `S_CUSTOMER` table. It confirms that the rows are partitioned into two groups based on the `CUSTOMER_TYPE_CODE` discriminator values: `DOMESTIC` and `INTERNATIONAL`.

Figure 4-20 Using the SQL Worksheet to Find Distinct Discriminator Column Values



```
select distinct customer_type_code
from s_customer
```

Query Result

SQL | All Rows Fetched: 3 in 0.109 seconds

	CUSTOMER_TYPE_CODE
1	INTERNATIONAL
2	CUSTOMER
3	DOMESTIC

Identifying the Subset of Attributes Relevant to Each Kind of Entity

Once you know how many different kinds of business entities are stored in the table, you will also know how many entity objects to create to model these distinct items. You'll typically create one entity object per kind of item. Then, to help determine which entity should act as the base of the hierarchy, you need to determine which subset of attributes is relevant to each kind of item.

For example, assume you determine that all of the attributes except `State` and `Language` are relevant to all users, and that:

- `State` is specific to domestic customers

- `Language` is specific to international customers.

This information leads you to determine that the `Customers` entity object should be the base of the hierarchy, with the `Domestics` and `Internationals` entity objects each extending `Customers` to add their specific attributes.

Creating the Base Entity Object in an Inheritance Hierarchy

To create the base entity object in an inheritance hierarchy, you use the Create Entity Object wizard.

Before you begin:

It may be helpful to have an understanding of entity objects in an inheritance hierarchy. For more information, see Using Inheritance in Your Business Domain Layer.

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Additional Functionality for Entity Objects.

You will also need to determine the discriminator column and values, as described in Identifying the Discriminator Column and Distinct Values, and determine the attributes for each entity object, as described in Identifying the Subset of Attributes Relevant to Each Kind of Entity.

To create the base entity object

1. In the Applications window, right-click the project to which you want to add the entity object and choose **New > From Gallery**.
2. In the New Gallery, expand **Business Tier**, select **ADF Business Components** and then **Entity Object**, and click **OK**.
3. In the Create Entity Object wizard, on the Name Page, provide a name and package for the entity, and select the schema object on which the entity will be based.

   In this example, name the entity object `Customers` and base it on the `S_CUSTOMER` table.
4. On the Attributes page, select the attributes in the **Entity Attributes** list that are not relevant to the base entity object (if any) and click **Remove** to remove them.

   In this example, you would remove the `State` and `Language` attributes from the list.
5. On the Attribute Settings page, use the **Select Attribute** dropdown list to choose the attribute that will act as the discriminator for the family of inherited entity objects and select the **Discriminator** checkbox to identify it as such. Importantly, you must also supply a **Default Value** for this discriminator attribute to identify rows of this base entity type.

   In this example, you would select the `CustomerTypeCode` attribute, mark it as a discriminator attribute, and set its **Default Value** to the value "`CUSTOMER`".

> **Note:**
>
> Leaving the **Default Value** blank for a discriminator attribute is legal. A blank default value means that a row with the discriminator column value `IS NULL` will be treated as this base entity type.

6. Then click **Finish**.

Creating a Subtype Entity Object in an Inheritance Hierarchy

To create a subtype entity object in an inheritance hierarchy, you use the Create Entity Object wizard.

Before you begin:

It may be helpful to have an understanding of entity objects in an inheritance hierarchy. For more information, see Using Inheritance in Your Business Domain Layer.

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Additional Functionality for Entity Objects.

You will also need to perform the following tasks:

1. Determine the discriminator column and values, as described in Identifying the Discriminator Column and Distinct Values.
2. Determine the attributes for each entity object, as described in Identifying the Subset of Attributes Relevant to Each Kind of Entity.
3. Create the parent entity object from which your new entity object will extend, as described in Creating the Base Entity Object in an Inheritance Hierarchy.
4. Make sure that the parent entity has a discriminator attribute already identified.

   If it does not, use the overview editor to set the **Discriminator** property on the appropriate attribute of the parent entity before creating the inherited child.

To create the new subtype entity object in the hierarchy:

1. In the Applications window, right-click the project to which you want to add the entity object and choose **New > From Gallery**.
2. In the New Gallery, expand **Business Tier**, select **ADF Business Components** and then **Entity Object**, and click **OK**.
3. In the Create Entity Object wizard, on the Name Page, provide a name and package for the entity, and click the **Browse** button next to the **Extends** field to select the parent entity from which the entity being created will extend.

   In this example, you would name the new entity `Domestics` and select the `Customers` entity object in the **Extends** field.
4. On the Attributes page, the **Entity Attributes** list displays the attributes from the underlying table that are not included in the base entity object. Select the attributes you do not want to include in this entity object and click **Remove**.

   In this example, because you are creating the `Domestics` entity you would remove the `Language` attribute and leave the `State` attribute.

5. Click **Override** to select the discriminator attribute so that you can customize the attribute metadata to supply a distinct **Default Value** for the `Domestics` entity subtype.

   In this example, you would override the `CustomerTypeCode` attribute.

6. On the Attribute Settings page, use the **Select Attribute** dropdown list to select the discriminator attribute. Change the **Default Value** field to supply a distinct default value for the discriminator attribute that defines the entity subtype being created.

   In this example, you would select the `CustomerTypeCode` attribute and change its **Default Value** to the value "`DOMESTIC`".

7. Click **Finish**.

> **Note:**
>
> You can repeat the same steps to define the `Internationals` entity object that extends `Customers` to add the additional `Language` attribute and overrides the **Default Value** of the `CustomerTypeCode` discriminator attribute to have the value "`INTERNATIONAL`".

How to Add Methods to Entity Objects in an Inheritance Hierarchy

To add methods to entity objects in an inheritance hierarchy, enable the custom Java class for the entity object and use the source editor to add the method. Methods that are common to all entity objects in the hierarchy are added to the base entity, while subtype-specific methods are added to the subtype. You can also override methods from the base entity object in the subtypes as necessary.

Adding Methods Common to All Entity Objects in the Hierarchy

To add a method that is common to all entity objects in the hierarchy, you add the method to the implementation class of the base entity object.

Before you begin:

It may be helpful to have an understanding of entity objects in an inheritance hierarchy. For more information, see Using Inheritance in Your Business Domain Layer.

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Additional Functionality for Entity Objects.

You will also need to perform the following tasks:

1. Create the base entity object and subtypes in the hierarchy, as described in How to Create Entity Objects in an Inheritance Hierarchy.
2. Create a custom Java implementation class for the base entity object, as described in Generating Custom Java Classes for an Entity Object.

To add a method common to all entity objects in a hierarchy:

1. In the Applications window, double-click the base entity object implementation class (for example, `CustomersImpl.java`).
2. In the source editor, add the method.

   For example, you could add the following method to the `CustomersImpl` class for the base `Customers` entity object:

   ```
   // In CustomersImpl.java
   public void performCustomerFunction() {
     System.out.println("## performCustomerFunction as Customer");
   }
   ```

Because this is the base entity object class, the methods you implement here are inherited by all subtype entity objects in the hierarchy.

Overriding Common Methods in a Subtype Entity Object

To override a method in a subtype entity object that is common to all entity objects in the hierarchy, you modify the common method inherited from the base entity object in the implementation class of the subtype entity object.

Before you begin:

It may be helpful to have an understanding of entity objects in an inheritance hierarchy. For more information, see Using Inheritance in Your Business Domain Layer.

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Additional Functionality for Entity Objects.

You will also need to perform the following tasks:

1. Create the base entity object and subtypes in the hierarchy, as described in How to Create Entity Objects in an Inheritance Hierarchy.
2. Create the common method in the base entity object, which your subtype entity object will override, as described in Adding Methods Common to All Entity Objects in the Hierarchy.
3. Create a a custom Java implementation class for the subtype entity object, as described in Generating Custom Java Classes for an Entity Object.

To override a method in a subtype entity object:

1. In the Applications window, double-click the subtype entity object implementation class (for example, `DomesticsImpl.java`).
2. With the subtype entity object implementation class open in the source editor, choose **Source > Override Methods** from the main menu.
3. In the Override Methods dialog, select the method you want to override (for example, the `performCustomerFunction()` method), and click **OK**.
4. In the source editor, customize the overridden method's implementation.

   For example, you could override the `performCustomerFunction()` method in the `DomesticsImpl` class for the `Domestics` subtype entity object and change the implementation to look like this:

   ```
   // In DomesticsImpl.java
   public void performCustomerFunction() {
   ```

```
  System.out.println("## performCustomerFunction as Domestics");
}
```

When working with instances of entity objects in a subtype hierarchy, sometimes you will process instances of multiple different subtypes. Because the `Domestics` and `Internationals` entity objects are special kinds of `Customers`, you can write code that works with all of them using the more generic `CustomersImpl` type that they all have in common. When doing this generic kind of processing of classes that might be one of a *family* of subtypes in a hierarchy, Java will always invoke the most specific override of a method available.

This means that invoking the `performCustomerFunction()` method on an instance of `CustomersImpl` that happens to really be the more specific `DomesticsImpl` subtype, will the result in printing out the following:

```
## performCustomerFunction as Domestics
```

instead of the default result that regular `CustomersImpl` instances would get:

```
## performCustomerFunction as Customer
```

Adding Methods Specific to a Subtype Entity Object

To add a method that is specific to a subtype entity object in the hierarchy, you simply add the method in the implementation class of the subtype using the source editor.

Before you begin:

It may be helpful to have an understanding of entity objects in an inheritance hierarchy. For more information, see Using Inheritance in Your Business Domain Layer.

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Additional Functionality for Entity Objects.

You will also need to perform the following tasks:

1. Create the base entity object and subtypes in the hierarchy, as described in How to Create Entity Objects in an Inheritance Hierarchy.
2. Create a custom Java implementation class for the subtype entity object, as described in Generating Custom Java Classes for an Entity Object.

To add a method specific to a subtype entity object:

1. In the Applications window, double-click the subtype entity object implementation class (for example, `InternationalsImpl.java`).
2. In the source editor, add the method.

   For example, you could add a `performInternationalFunction()` method to the `InternationalsImpl` class for the subtype `Internationals` entity object:

   ```
   // In InternationalsImpl.java
   public void performInternationalFunction() {
     System.out.println("## performInternationalFunction called");
   }
   ```

What You May Need to Know About Using Inheritance

When using inheritance, you can also introduce a new base entity, find subtype entities using a primary key, and create view objects with polymorphic entity usages.

Introducing a New Base Entity

For example, where the `Customers` entity object corresponds to a concrete kind of row in the `S_CUSTOMERS` table, it also plays the role of the base entity in the inheritance hierarchy. In other words, all of its attributes were common to all entity objects in the hierarchy. A situation might arise, however, where the `Customers` entity object required a property that was specific to customers, but not common to domestic or international customers.

In this case, you can introduce a new entity object (for example, `BaseCustomers`) to act as the base entity in the hierarchy. It would have all of the attributes common to all `Customers`, `Domestics`, and `Internationals` entity objects. Then each of the three entities that correspond to concrete rows that appear in the table could have some attributes that are inherited from `BaseCustomers` and some that are specific to the individual subtype. In the `BaseCustomers` type, so long as you mark the `CustomerTypeCode` attribute as a discriminator attribute, you can just leave the **Default Value** blank (or some other value that does *not* occur in the `CUSTOMER_TYPE_CODE` column in the table). Because you will not use instances of the `BaseCustomers` entity in the application, it doesn't matter what its discriminator default value is.

Subtype Entity Objects and the findByPrimaryKey() Method

When you use the `findByPrimaryKey()` method on an entity definition, it only searches the entity cache for the entity object type on which you call it. For example, this means that if you call `CustomersImpl.getDefinitionObject()` to access the entity definition for the `Customers` entity object when you call `findByPrimaryKey()` on it, you will only find entities in the cache that happen to be customers. Sometimes this is exactly the behavior you want.

However, if you want to find an entity object by primary key allowing the possibility that it might be a subtype in an inheritance hierarchy, then you can use the `findByPKExtended()` method from the `EntityDefImpl` class instead.

For example, if you have created subtypes of the `Customers` entity object, this alternative finder method would find an entity object by primary key whether it is a customer, domestic, or international. You can then use the Java `instanceof` operator to test which type you found, and then cast the `CustomersImpl` object to the more specific entity object type to work with features specific to that subtype.

View Objects with Polymorphic Entity Usages

When you create an entity-based view object with an entity usage corresponding to a base entity object in an inheritance hierarchy, you can configure the view object to query rows corresponding to multiple different subtypes in the base entity's subtype hierarchy. Each row in the view object will use the appropriate subtype entity object as the entity row part, based on matching the value of the discriminator attribute. See How to Create a Subtype View Object with a Polymorphic Entity Usage, for specific instructions on setting up and using these view objects.

5

Defining SQL Queries Using View Objects

This chapter describes how to create ADF view objects to create SQL queries that join, filter, sort, and aggregate data for use in an Oracle ADF application. It describes how view objects map their SQL-derived attributes to database table columns and static data sources, such as a flat file.
This chapter includes the following sections:

About View Objects

Use Oracle ADF view objects to encapsulate SQL queries and build data models of ADF application modules.

A **view object** is an **Oracle Application Development Framework** (Oracle ADF) component that encapsulates a SQL query and simplifies working with its results. There are several types of view objects that you can create in your data model project:

- Entity-based view objects when data updates will be performed or just to read data
- Non-entity backed view objects for cases not supported by entity objects (view objects with no entity usage definition are always read-only)
- Static data view objects for data defined by the view object itself
- Programmatically populated view objects (see Using Programmatic View Objects for Alternative Data Sources)

An **entity-based view object** can be configured to support updatable rows when you create view objects that map their attributes to the attributes of one or more existing entity objects. The mapped entity object is saved as an entity usage in the view object definition. In this way, entity-based view objects cooperate automatically with entity objects to enable a fully updatable data model. The entity-based view object usage queries just the data needed for the client-facing task and relies on its mapped entity objects and their entity cache of rows to automatically validate and save changes made to its view rows. An entity-based view object encapsulates a SQL query, it can be linked into master-detail hierarchies, and it can be used in the data model of an **application module**. The backing entity cache manages updates to the queried data so that every view object usage based on the same entity object points to the entity cache of rows rather than store duplicate data in the view object usage.

In contrast to entity-based view objects, *entity-less view objects* require you to write the query using the SQL query language. Non-entity backed view objects do not pick up entity-derived default values, they do not reflect pending changes, and they do not reflect updated reference information. The Create View Object wizard and overview editor for entity-based view objects, on the other hand, simplify this task by helping you to construct the SQL query declaratively. In addition, because entity-based view objects manage queried data using the entity cache, they have better performance over view objects with no entity usage which require special runtime handling of updates to their view object row cache. For these reasons, it is almost always preferable to create an entity-mapped view object, even when you want to create a view object just to read data.

There remain a few situations where it is still preferable to create a non-entity-mapped view object to read data, including SQL-based validation, Unions, and Group By queries. In this case, it is worth noting, that view objects with no entity usage definition are always read-only.

As an alternative to creating view objects that specify a SQL statement at design time, you can create entity-based view objects that contain no SQL statements. This capability of the **ADF Business Components** design time and runtime is known as **declarative SQL mode**. The business component developer who works with the wizard or editor for a view object in declarative SQL mode, requires no knowledge of SQL. In declarative SQL mode, the runtime query statement is based solely on the usage of attributes in the databound UI component.

When a view object has one or more underlying entity usages, you can create new rows, and modify or remove queried rows. The entity-based view object coordinates with underlying entity objects to enforce business rules and to permanently save the changes to the database. In addition, entity-based view objects provide these capabilities that do not exist with entity-less view objects:

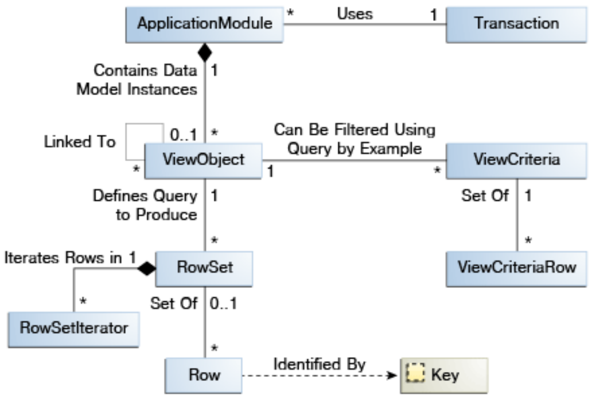
- Changes in cache (updates, inserts, deletes) managed by entities survive the view object's execution boundary.
- Changes made to relevant entity object attributes through other view objects in the same transaction are immediately reflected.
- Attribute values of new rows are initialized to the values from the underlying entity object attributes.
- Changes to foreign key attribute values cause reference information to get updated.
- Validation for row (entity) level is supported.
- Composition features, including validation, locking, ordered-updates are supported.
- Support for effective dating, change indicator, and business events.

View Object Use Cases and Examples

This chapter helps you understand these view object concepts as illustrated in Figure 5-1:

- You define a view object by providing a SQL query (either defined explicitly or declaratively).
- You use view object instances in the context of an application module that provides the database transaction for their queries.
- You can link a view object to one or more others to create master-detail hierarchies.
- At runtime, the view object executes your query and produces a set of rows (represented by a `RowSet` object).
- Each row is identified by a corresponding row key.
- You iterate through the rows in a row set using a **row set iterator**.
- You can filter the row set a view object produces by applying a set of Query-by-Example criteria rows.

Figure 5-1 A View Object Defines a Query and Produces a Row Set of Rows

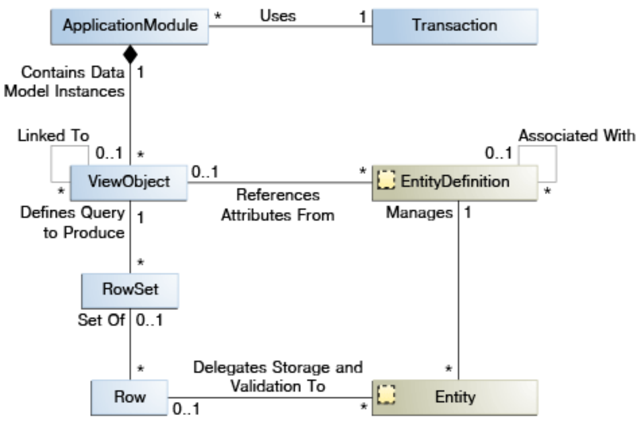


This chapter explains how instances of entity-based view objects contained in the data model of your application module enable clients to search for, update, insert, and delete **business services layer** information in a way that combines the full data shaping power of SQL with the clean, object-oriented encapsulation of reusable domain business objects. And all without requiring a line of code.This chapter helps you to understand these entity-based view object concepts as illustrated in Figure 5-2:

- You define an updatable view object by referencing attributes from one or more entity objects.
- You can use multiple, associated entity objects to simplify working with reference information.
- You can a define **view link** based on underlying **entity associations**.
- You use your entity-based view objects in the context of an application module that provides the transaction.

- At runtime, the view row delegates the storage and validation of its attributes to underlying entity objects.

Figure 5-2 View Objects and Entity Objects Collaborate to Enable an Updatable Data Model



Additional Functionality for View Objects

You may find it helpful to understand other **Oracle ADF** features before you start working with view objects. Following are links to other functionality that may be of interest.

- For additional information about using Groovy script wherever expressions are supported in view object definitions, see Using Groovy Scripting Language with Business Components.
- For details about using the interactive Oracle ADF Model Tester to validate view object query results, see Testing View Instance Queries.
- For details about the role of the entity cache plays in a transaction involving entity-based view objects, see What Happens at Runtime: How View Objects and Entity Objects Cooperate.
- For details about creating a data model consisting of view object instances, see Implementing Business Services with Application Modules.
- For a quick reference to the most common code that you will typically write, use, and override in your custom view object classes, see Most Commonly Used ADF Business Components Methods.
- For API documentation related to the `oracle.jbo` package, see the following Javadoc reference document:
  - *Java API Reference for Oracle ADF Model*

Populating View Object Rows from a Single Database Table

ADF Business Components view objects are utilized to retrieve data from a data source such as a database table using SQL queries.

View objects provide the means to retrieve data from a data source. In the majority of cases, the data source will be a database and the mechanism to retrieve data is the

SQL query. ADF Business Components can work with JDBC to pass this query to the database and retrieve the result.

When view objects use a SQL query, query columns map to view object attributes in the view object. The definition of these attributes, saved in the view object's XML definition file, reflect the properties of these columns, including data types and precision and scale specifications.

> **Performance Tip:**
>
> If the query associated with the view object contains query parameters whose values change from execution to execution, use bind variables. Using bind variables in the query allows the query to reexecute without needing to reparse the query on the database. You can add bind variables to the view object in the Query page of the overview editor for the view object. For more information, see See Working with Bind Variables.

Using the same Create View Object wizard, you can create view objects that either map to the attributes of existing entity objects or not. Only entity-based view objects automatically coordinate with mapped entity objects to enforce business rules and to permanently save data model changes. Additionally, you can disable the Updatable feature for entity-based view objects and work entirely declaratively to query read-only data. Alternatively, you can use the wizard or editor's custom SQL mode to work directly with the SQL query language, but the view object you create will not support the transaction features of the entity-based view object.

When you create entity-based view objects, you have the option to keep the view object definition entirely declarative and maintain a customizable view object. Or, you may switch to custom SQL query editing to create queries that entity objects alone cannot express, including Unions and Group By queries. Additionally, custom SQL view objects are useful in SQL-based validation queries used by the view object-based Key Exists validator. It is worth noting that, by definition, using custom SQL mode to define a SQL query means the view object is read only.

For information about the differences between entity-based view objects and entity-less view objects, see About View Objects.

How to Create an Entity-Based View Object

Creating an entity-based view object is the simplest way to create a view object. It is even easier than creating a custom SQL view object, since you don't have to type in the SQL statement yourself. An entity-based view object also offers significantly more runtime functionality than its custom SQL counterpart.

In an entity-based view object, the view object and entity object play cleanly separated roles:

- The view object is the data *source*: it retrieves the data using SQL.
- The entity object is the data *sink*: it handles validating and saving data changes.

Because view objects and entity objects have cleanly separated roles, you can build different view objects — projecting, filtering, joining, sorting the data in whatever way your user interfaces require, application after application — without any changes to the reusable entity object. In fact, it is possible that the development team responsible for the core business services layer of entity objects might be completely separate from another team responsible for the specific application modules and view objects needed to support the end-user environment. This relationship is enabled by metadata that the entity-based view object encapsulates. The metadata specifies how the `SELECT` list columns are related to the attributes of one or more underlying entity objects.

Your entity-based view object may be based on more than one database table. To use database joins to add multiple tables to the view object, see Working with Multiple Tables in Join Query Results.

Creating a View Object with All the Attributes of an Entity Object

When you want to allow the client to work with *all* of the attributes of an underlying entity object, you can use the Create Default View Object dialog to quickly create the view object from an entity object that you select in the Applications window. To create the default view object, select the entity object, right-click and choose **New Default View Object**. When you do not want the view object to contain all the attributes, you can use the Create View Object wizard and select only the attributes that you need, as described in Creating an Entity-Based View Object from a Single Table.

Before you begin:

It may be helpful to have an understanding of entity-based view objects. For more information, see How to Create an Entity-Based View Object.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for View Objects.

You will need to complete this task:

> Either create the desired entity objects yourself, as described in How to Create Multiple Entity Objects and Associations from Existing Tables or consult the developer responsible for creating entity objects and let them know the data that you expect to query.

To create a default entity-based view object:

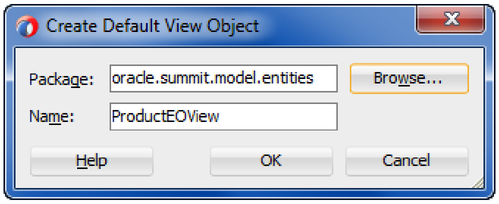
1. In the Applications window, right-click the entity object and choose **New Default View Object**.

   This context menu option lets you create a view object based on a single entity object that you select. If you need to add additional entity objects to the view object definition, you can use the Entity Objects page of the view object overview editor after you create the view object.

2. In the Create Default View Object dialog, provide a package and component name for the new view object.

   In the Create Default View Object dialog you can click **Browse** to select the package name from the list of existing packages. For example, in Figure 5-3, clicking **Browse** locates `oracle.summit.model.entities` package on the classpath for the `Model` project of the application.

Figure 5-3 Shortcut to Creating a Default View Object for an Entity Object



Package: oracle.summit.model.entities

Name: ProductEOView

The new entity-based view object created will be identical to one you could have created with the Create View Object wizard. By default, it will have a single entity usage referencing the entity object you selected in the Applications window, and will include all of its attributes. It will initially have neither a `WHERE` nor `ORDER BY` clause, and you may want to use the overview editor for the view object to:

- Remove unneeded attributes
- Refine its selection with a `WHERE` clause
- Order its results with an `ORDER BY` clause
- Customize any of the view object properties

Creating an Entity-Based View Object from a Single Table

To create an entity-based view object, use the Create View Object wizard, which is available from the New Gallery.

Before you begin:

It may be helpful to have an understanding of entity-based view objects. See, see How to Create an Entity-Based View Object.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. See, see Additional Functionality for View Objects.

You will need to complete this task:

> Create the desired entity objects, as described in How to Create Multiple Entity Objects and Associations from Existing Tables.

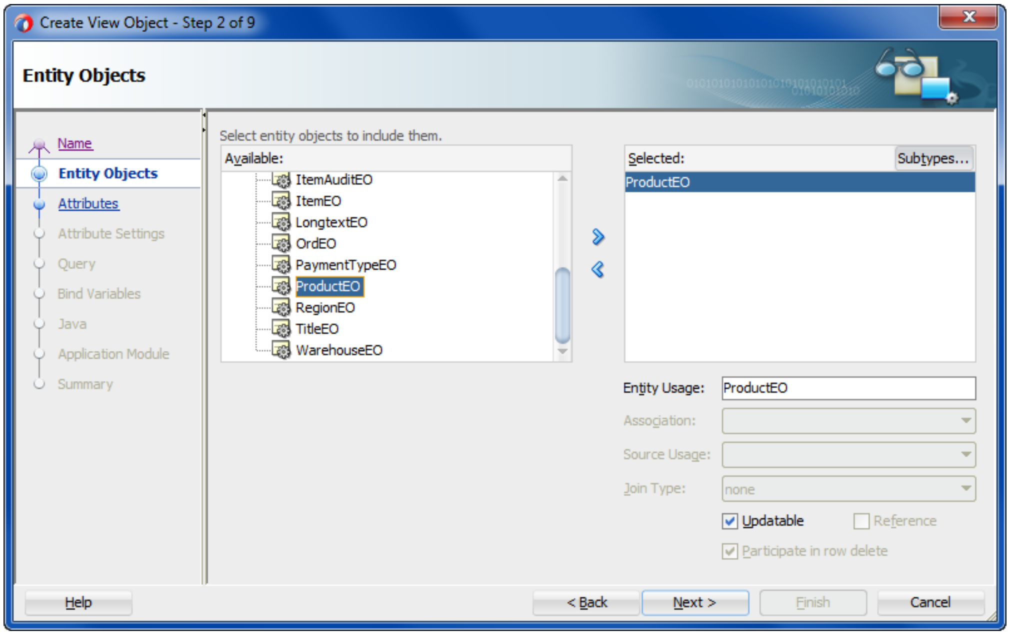
To create an entity-based view object from a single table:

1. In the Applications window, right-click the package of the data model project in which you want to create the view object and choose **New** and then **View Object**.
2. In the Create View Object wizard, on the Name page, enter a package name and a view object name. Keep the default setting **Entity** selected to indicate that you want this view object to manage data with its base entity object. Click **Next**.
3. On the Entity Objects page, select an entity object whose data you want to use in the view object. Click **Next**.

   An entry in this list is known as an **entity usage**, since it records the entity objects that the view object will be using. Each entry could also be thought of as an **entity reference**, since the view object references attributes from that entity. For information about working table joins to create additional entity usages, see Working with Multiple Tables in Join Query Results.

For example, Figure 5-4 shows the result after shuttling the `ProductEO` entity object into the **Selected** list.

Figure 5-4 Create View Object Wizard, Entity Objects Page



Create View Object - Step 2 of 9

Entity Objects

Select entity objects to include them.

Available:
- ItemAuditEO
- ItemEO
- LongtextEO
- OrdEO
- PaymentTypeEO
- ProductEO
- RegionEO
- TitleEO
- WarehouseEO

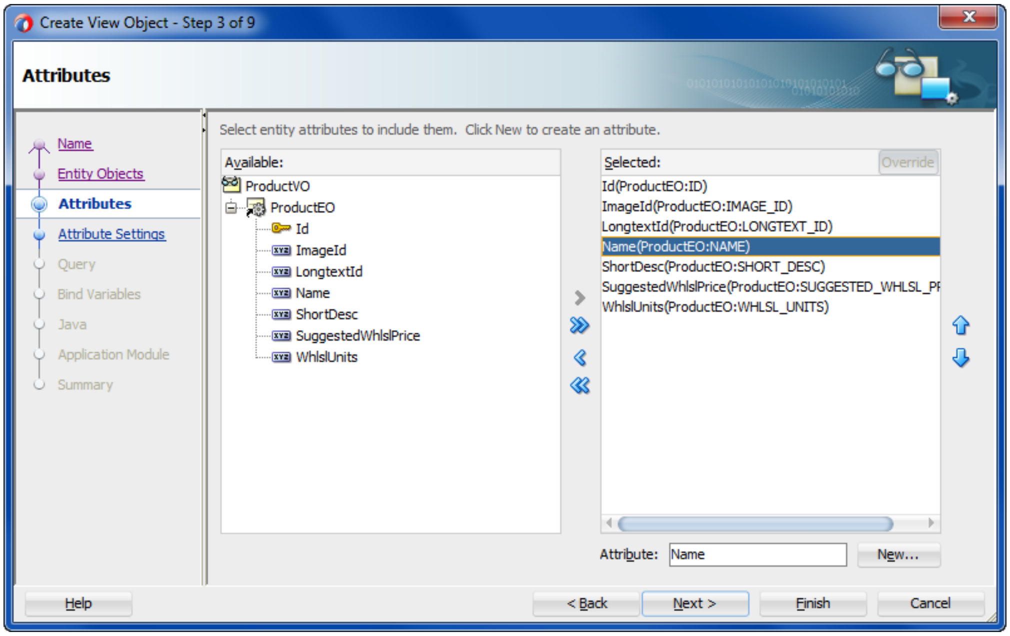
Selected:
- ProductEO

Entity Usage: ProductEO
Association:
Source Usage:
Join Type: none
☑ Updatable ☐ Reference
☑ Participate in row delete

4. On the Attributes page, select the attributes you want to include from each entity usage in the **Available** list and shuttle them to the **Selected** list. Click **Next**.

   For example, Figure 5-5 shows the attributes have been selected from the `ProductEO`.

Figure 5-5 Create View Object Wizard, Attributes Page



Create View Object - Step 3 of 9

Attributes

Select entity attributes to include them. Click New to create an attribute.

Available:
- ProductVO
  - ProductEO
    - Id
    - ImageId
    - LongtextId
    - Name
    - ShortDesc
    - SuggestedWhlslPrice
    - WhlslUnits

Selected:
- Id(ProductEO:ID)
- ImageId(ProductEO:IMAGE_ID)
- LongtextId(ProductEO:LONGTEXT_ID)
- Name(ProductEO:NAME)
- ShortDesc(ProductEO:SHORT_DESC)
- SuggestedWhlslPrice(ProductEO:SUGGESTED_WHLSL_PF
- WhlslUnits(ProductEO:WHLSL_UNITS)

Attribute: Name

5. On the Attribute Settings page, optionally, use the **Select Attribute** dropdown list to switch between the view object attributes in order to change their names or any of their initial settings.

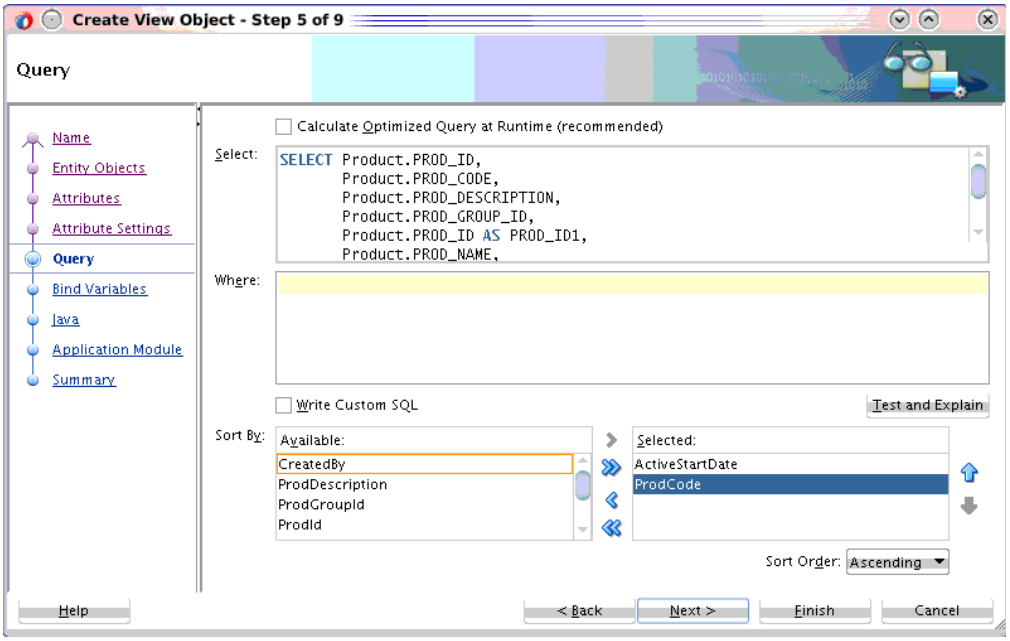
   See about any of the attribute settings, press F1 or click **Help**.

6. On the Query page, deselect **Calculate Optimized Query at Runtime** . Optionally add a `WHERE` clause and use the `Sort By` panel to shuffle entries from Available to Selected to add a sort criteria. JDeveloper automatically generates the SELECT statement based on the entity attributes you've selected.

   Do not include the `WHERE` keywords in the **Where** field values. The view object adds those keywords at runtime when it executes the query.

   For example, Figure 5-6 shows the `Sort By` clause, where the entries in the Selected column indicate the sort criteria.

Figure 5-6 Create View Object Wizard, Query Page



□ Calculate Optimized Query at Runtime (recommended)

Select:

```
SELECT  Product.PROD_ID,
        Product.PROD_CODE,
        Product.PROD_DESCRIPTION,
        Product.PROD_GROUP_ID,
        Product.PROD_ID AS PROD_ID1,
        Product.PROD_NAME,
```

Where:

□ Write Custom SQL

Sort By:

Available: CreatedBy, ProdDescription, ProdGroupId, ProdId

Selected: ActiveStartDate, ProdCode

Sort Order: Ascending

7. Click **Finish**.

What Happens When You Create an Entity-Based View Object

When you create a view object, JDeveloper creates the XML document file that represents the view object's declarative settings and saves it in the directory that corresponds to the name of its package. For example, the view object `ProductVO`, added to the `views` package, will have the XML file `./views/ProductVO.xml` created in the project's source path.

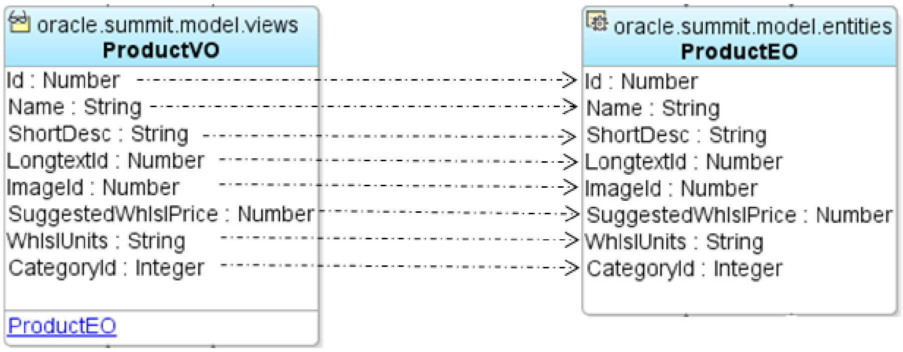
To view the view object settings, select the desired view object in the Applications window and open the Structure window. The Structure window displays the list of XML definitions, including the SQL query, the name of the entity usage, and the properties of each attribute. To open the XML definition in the editor, right-click the corresponding node and choose **Go to Source**.

Note:

If you configure JDeveloper preferences to generate default Java classes for ADF Business Components, the wizard will also create an optional custom view object class (for example, ProductVOImpl.java) and/or a custom view row class (for example, ProductVORowImpl.java). For details about setting preferences, see How to Customize Model Project Properties for ADF Business Components.

Figure 5-7 depicts the entity-based view object ProductVO and the singe entity usage referenced in its query statement. The dotted lines represent the metadata captured in the entity-based view object's XML document that map SELECT list columns in the query to attributes of the entity objects used in the view object.

Figure 5-7 View Object Encapsulates a SQL Query and Entity Attribute Mapping Metadata



What You May Need to Know About Non-Updatable View Objects

When you use the Create View Object wizard to create an entity-based view object, by default the attributes of the view object will be updatable. When you want to make all the attributes of the view object read-only, you can deselect **Updatable** when you add the entity object in the wizard. Later you may decide to convert the view object to one that permits updates to its SQL-mapped table columns. However, this cannot be accomplished by merely changing the attribute's **Updatable** property. You must delete the read-only view object and recreate the entity-based view object so that the intended attributes are updatable.

How to Edit a View Object

After you've created a view object, you can edit any of its settings in the overview editor for the view object.

> **Performance Tip:**
>
> How you configure the view object to fetch data plays a large role in the runtime performance of the view object. For information about the tuning parameters that you can edit to optimize performance, see What You May Need to Know About Optimizing View Object Runtime Performance.

Before you begin:

It may be helpful to have an understanding of view objects. For more information, see Populating View Object Rows from a Single Database Table.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for View Objects.

To edit a view object definition:

1. In the Applications window, double-click the view object that you want to edit.
2. In the overview editor, click the desired navigation tab.

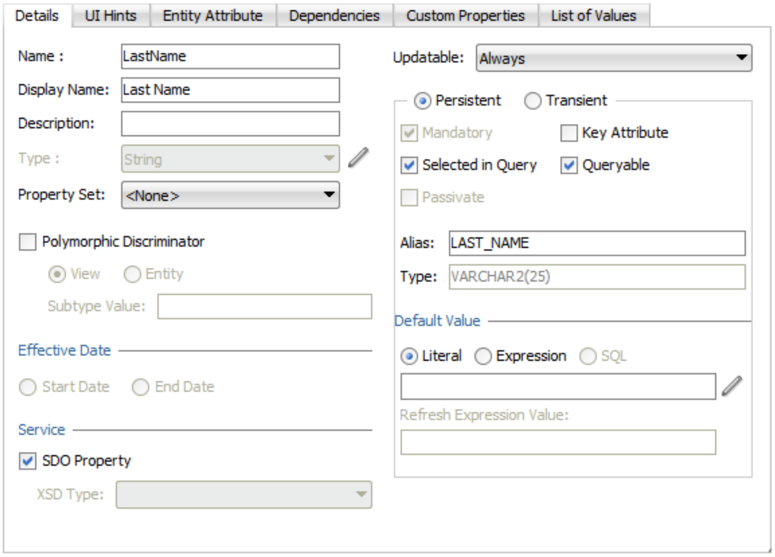
   The pages of the view object overview editor let you adjust the SQL query, change the attribute names, add named bind variables, add UI controls hints, control Java generation options, and edit other settings.

Overriding the Inherited Properties from Underlying Entity Object Attributes

One interesting aspect of entity-based view objects is that each attribute that relates to an underlying entity object attribute inherits that attribute's properties. Figure 5-8 shows the Details section of the view object editor's Attributes page with an inherited attribute selected. You can see that fields like the Java attribute type and the query column type are disabled and their values are inherited from the related attribute of the underlying entity object to which this view object is related. Some properties like the attribute's data type are inherited and cannot be changed at the view object level.

Other properties like `Queryable` and `Updatable` are inherited but can be overridden as long as their overridden settings are more restrictive than the inherited settings. For example, the attribute from underlying entity object might have an **Updatable** setting of **Always**. As shown Figure 5-8, the Details section of the Attributes page of the view object overview editor allows you to set the corresponding view object attribute to a more restrictive setting like **While New** or **Never**. However, if the attribute in the underlying entity object had instead an **Updatable** setting of **Never**, then the editor would not allow the view object's related attribute to have a less restrictive setting like **Always**.

Figure 5-8 View Object Attribute Properties Inherited from Underlying Entity Object



Name : LastName
Display Name: Last Name
Description:
Type : String
Property Set: <None>
☐ Polymorphic Discriminator
● View ○ Entity
Subtype Value:
○ Start Date ○ End Date
☑ SDO Property
XSD Type:

Updatable: Always
● Persistent ○ Transient
☑ Mandatory ☐ Key Attribute
☑ Selected in Query ☑ Queryable
☐ Passivate
Alias: LAST_NAME
Type: VARCHAR2(25)
● Literal ○ Expression ○ SQL
Refresh Expression Value:

Customizing View Object Attribute Display in the Overview Editor

When you edit view objects in the overview editor, you can customize the Attributes page of the overview editor to make better use of the attributes table displayed for the view object.

Customization choices that you make for the attributes table include the list of attribute properties to display as columns in the attributes table, the order that the columns appear (from left to right) in the attributes table, the sorting order of the columns, and the width of the columns. The full list of columns that you can choose to display correspond to the properties that you can edit for the attribute.

For example, you can add the **Updatable** property as a column to display in the attributes table when you want to quickly determine which attributes of your view object are updatable. Or, you can add the attributes' **Label** property as a column and see the same description as the end user. Or, you might want to view the list of attributes based on their entity usages. In this case, you can display the **Entity Usage** column and sort the entire attributes table on this column.

When you have set up the attributes table with the list of columns that you find most useful, you can apply the same set of columns to the attributes table displayed for other view objects by right-clicking the attributes table and choose **Apply to All View Objects**.

Before you begin:

It may be helpful to have an understanding of view objects. For more information, see Populating View Object Rows from a Single Database Table.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for View Objects.

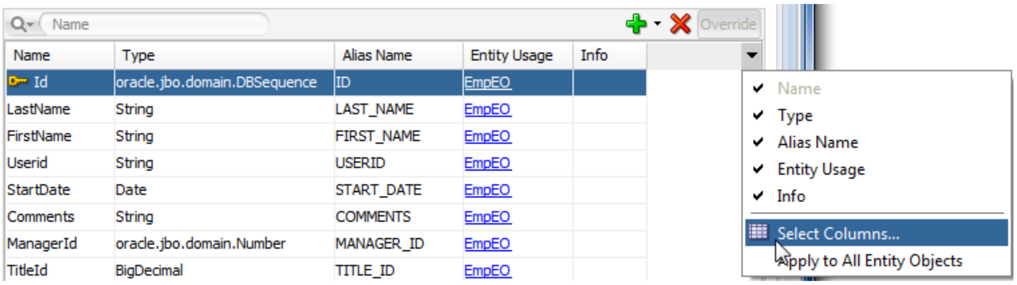
To customize the attributes table display:

1. In the Applications window, double-click the view object for which you want to customize the attributes table display.

2. In the overview editor, click the **Attributes** navigation tab.

3. In the Attributes page, click the dropdown menu icon to the right of the attribute column headers (just below the attributes table's button bar) and choose **Select Columns**.

Figure 5-9 Changing the Attribute Columns to DIsplay in the Overview Editor



Name	Type	Alias Name	Entity Usage	Info
Id	oracle.jbo.domain.DBSequence	ID	EmpEO	
LastName	String	LAST_NAME	EmpEO	
FirstName	String	FIRST_NAME	EmpEO	
Userid	String	USERID	EmpEO	
StartDate	Date	START_DATE	EmpEO	
Comments	String	COMMENTS	EmpEO	
ManagerId	oracle.jbo.domain.Number	MANAGER_ID	EmpEO	
TitleId	BigDecimal	TITLE_ID	EmpEO	

4. In the Select Columns dialog, perform any of the following actions:

   a. Click the left/right shuttle buttons to change the list of visible columns in the attributes table of the overview editor. The overview editor displays only those columns corresponding to the attribute properties that appear the **Selected** list.

   b. Click one of the **Move Selection** buttons to change the position of the columns in the attributes table of the overview editor. The overview editor displays the attribute properties arranged from left to right starting with the property at the top of the **Selected** list.

5. Click **OK**.

6. On the Attributes page of the overview editor, perform any of the following actions:

   a. Select any column header and drag to change the position of the column in the attributes table of the overview editor.

   b. Click any column header to sort all columns in the attributes table by the selected column.

      This feature is particularly useful when you want to focus on a particular column. For example, in the case of an entity-based view object, you can click the **Entity Usage** column header to group attributes in the attributes table by their underlying entity objects.

   c. Click any column header border and drag to adjust the width of the attributes table's column.

   d. Click the dropdown list to the right of the column headers and select among the list of displayed columns to change the visibility of a column in the current attributes table display.

      This feature lets you easily hide columns when you want to simplify the attributes table display in the current view object overview editor.

7. To extend the changes in the columns (including column list, column order, column sorting, and column width) to all other view object overview editors, click the

dropdown menu to the right of the column headers and choose **Apply to All View Objects**.

This feature allows you to easily compare the same attributes across view objects. The overview editor will apply the column selections (and order) that you make in the Select Columns dialog and the current attributes table's column sorting and column widths to all view objects that you edit. View objects that are currently displayed in an open overview editor are not updated with these settings; you must close the open view object overview editor and then reopen the view object to see these settings applied.

Modifying the Order of Attributes in the View Object Source File

After you create a view object definition, you may decide to change the order of the attributes queried by the view object. This view object editing feature allows you to easily change the order that the attributes will appear in the attributes table displayed on the Attributes page of the view object overview editor. Because this feature acts on specific attributes and alters the XML definition of the current view object, it does not apply to other view objects that you may edit. Alternatively, you can sort the display of attributes on the Attribute page of the view object overview editor without affecting the source file by clicking any column header in the overview editor's attributes table.

Before you begin:

It may be helpful to have an understanding of view objects. For more information, see Populating View Object Rows from a Single Database Table.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for View Objects.

To modify the order of attributes in the view object source file:

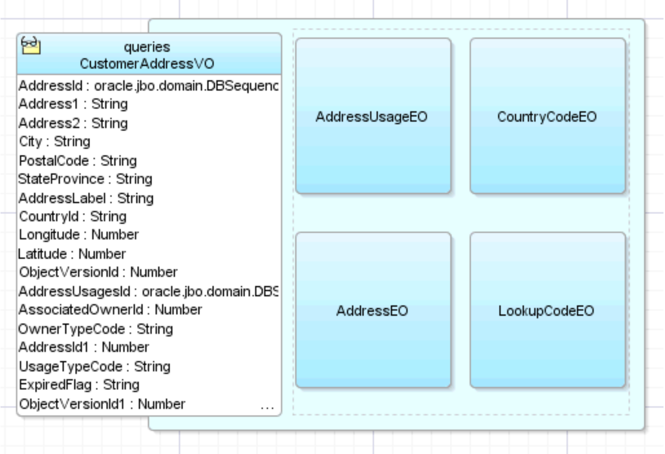
1. In the Applications window, double-click the view object that you want to modify.
2. In the overview editor, click the **Attributes** navigation tab and then click **Set Source Order**.
3. In the Set Source Order dialog, select the attribute you want to reposition and click one of the **Move Selection** buttons.
4. Click **OK**.

   This feature has no affect on other view objects that you may edit; it only affects the current view object.

How to Show View Objects in a Business Components Diagram

JDeveloper's UML diagramming lets you create a Business Components diagram to visualize your business services layer. In addition to supporting entity objects, JDeveloper's UML diagramming allows you to drop view objects onto diagrams as well to visualize their structure and entity usages. For example, if you create a new Business Components Diagram in the `oracle.summit.model` package, and drag the `CustomerAddressVO` view object from the Applications window onto the diagram, its entity usages would display, as shown in Figure 5-10. When viewed as an expanded node, the diagram shows a compartment containing the view objects entity usages.

Figure 5-10 View Object and Its Entity Usages in a Business Components Diagram



Before you begin:

It may be helpful to have an understanding of view objects. For more information, see Populating View Object Rows from a Single Database Table.

You may also find it helpful to understand how to create an entity diagram, see Creating a Diagram of Entity Objects for Your Business Layer.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for View Objects.

To create a business components diagram that models existing view objects:

1. In the Applications window, right-click the package in the data model project in which you want to create the business component diagram and choose **New** and then **Business Components Diagram**.
2. In the Create Business Components Diagram dialog, enter the name of the diagram and enter the package name in which the diagram will be created. Select any additional diagram features.
3. Click **OK**.
4. To add existing view objects to the diagram, select them in the Applications window and drop them onto the diagram surface.

Working with View Objects in Declarative SQL Mode

Create entity-based view objects as a no-SQL alternative and utilize ADF Business Components’ declarative SQL mode in design time and runtime.

At runtime, when ADF Business Components works with JDBC to pass a query to the database and retrieve the result, the mechanism to retrieve the data is the SQL query. As an alternative to creating view objects that specify a SQL statement at design time, you can create entity-based view objects that contain no SQL statements. This capability of the ADF Business Components design time and runtime is known as **declarative SQL mode**. When the data model developer works with the wizard or editor for a view object in declarative SQL mode, they require no knowledge of SQL. In declarative SQL mode, the ADF Business Components runtime generates the SQL query statements based on metadata that you must define for the view object, as follows:

- Generates `SELECT` and `FROM` lists based on the rendered web page's databound UI components' usage of one or more entity objects' attributes

  Specifying the runtime query statement based solely on databound UI component attribute usage is an optimization that you control at the level of each view object attribute by changing the attribute's `IsSelected` property setting. By default, the property setting is `IsSelected=true` for each attribute that you add to the view object in declarative SQL mode. The default setting specifies the added attribute will be selected in the SQL statement regardless of whether or not the attribute is exposed in the UI by a databound component. For details about changing the property setting to optimize the runtime query statement, see How to Create Declarative SQL View Objects.

- In the case of ADF RESTful web services, generates a SELECT clause based on the list of attributes specified using the URL query parameter fields.

  SQL SELECT statements executed by the ADF REST resource’s backing view object are based at runtime on how the view object was created. Only view objects that you create with declarative SQL mode enabled support optimized SQL SELECT statements formed exclusively by the list of attributes named by the query parameter fields. The SELECT statement executed by non-declarative view objects will contain all attributes of the view object definition. To gain this runtime optimization, it is therefore recommended that ADF Business Components developers create view objects for ADF REST resources using only declarative SQL mode.

- Generates a `WHERE` clause based on a **view criteria** that you add to the view object definition
- Generates an `ORDERBY` clause based on a sort criteria that you add to the view object definition.
- Augments the `WHERE` clause to support table joins based on named view criteria that you add to the view object definition
- Augments the `WHERE` clause to support master-detail view filtering based on a view criteria that you add to either the source or destination of a view link definition

Additionally, the SQL statement that a declarative SQL mode view object generates at runtime will be determined by the SQL platform specified in the Business Components page of the Project Properties dialog.

> **Note:**
>
> Currently, the supported platforms for runtime SQL generation are SQL92 (ANSI) style and Oracle style. For information about setting the SQL platform for your project, see How to Initialize the Data Model Project With a Database Connection.

Declarative SQL mode selection is supported in JDeveloper as the default setting for all view objects that you create in the Create View Object wizard. The wizard allows you to override the declarative SQL mode setting for any view object you create when you want to create custom SQL at design time.

When you work with custom SQL, the view object definitions you create at design time always contain the entire SQL statement based on the SQL platform required by your application module's defined database connection. Thus the capability of SQL independence does not apply to view objects that you create in custom SQL mode. For information about using the wizard and editor to customize view objects when SQL is desired at design time, see Populating View Object Rows from a Single Database Table.

How to Create Declarative SQL View Objects

All view objects that you create in JDeveloper rely on the same design time wizard and editor. However, when you enable declarative SQL mode, the wizard and editor change to support customizing the view object definition without requiring you to display or enter any SQL. For example, the Query page of the Create View Object wizard with declarative SQL mode enabled lacks the **Generated SQL** field present in normal mode.

Additionally, in declarative SQL mode, since the wizard and editor do not allow you to enter the `WHERE` clause, you provide equivalent functionality by defining a view criteria and sort criteria (using the Sort By panel) respectively. In declarative SQL mode, these criteria appear in the view object metadata definition and will be converted at runtime to their corresponding SQL clause. When the databound UI component has no need to display filtered or sorted data, you may omit the view criteria or sort criteria from the view object definition.

Otherwise, after you enable declarative SQL mode, the basic procedure to create a view object with ensured SQL independence is the same as you would follow to create any entity-based view object. For example, you must still ensure that your view object encapsulates the desired entity object metadata to support the intended runtime query. As with any entity-based view object, the columns of the runtime-generated `FROM` list must relate to the attributes of one or more of the view object's underlying entity objects. In declarative SQL mode, you automatically fulfill this requirement when working with the wizard or editor when you add or remove the attributes of the entity objects on the view object definition.

If you prefer to optimize the declarative SQL query so that the `SELECT` and `FROM` clauses of the SQL query statement are based solely on whether or not the attributes you add to the view object are rendered at runtime by a databound UI component, then you must disable the **Selected in Query** checkbox (sets `IsSelected=false` for the view object definition) for all added attributes. By default, the `IsSelected` property is `true` for any attribute that you add to the view object in declarative SQL mode. The default setting means the added attribute will be selected in the SQL statement regardless of whether or not the attribute is exposed by a databound UI component. When you create a new view object in declarative SQL mode, you can use the Attribute Settings page of the Create View Object wizard to change the setting for each attribute. If you need to alter this setting after you generate the view object, you can use the Properties window to change the **Selected in Query** setting for one or more attributes that you select in the Attributes page of the view object editor.

> **Performance Tip:**
>
> A view object instance configured to generate SQL statements dynamically will requery the database during page navigation if a subset of all attributes with the same list of key entity objects is used in the subsequent page navigation. Thus performance can be improved by activating a superset of all the required attributes to eliminate a subsequent query execution.

Before you begin:

It may be helpful to have an understanding of declarative SQL mode. For more information, see How to Create Declarative SQL View Objects.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for View Objects.

To create declarative SQL-based view objects:

1. In the Applications window, right-click the package in the data model project in which you want to create the view objects and choose **New** and then **View Object**.
2. In the Create View Object wizard, on the Name page, enter a package name and a view object name. Keep the default setting **Entity** selected to indicate that you want this view object to manage data with its base entity object. Click **Next**.

   Any other choice for the data selection will disable declarative SQL mode in the Create View Object wizard.
3. On the Entity Objects page, select the entity object whose data you want to use in the view object. Click **Next**.

   When you want to create a view object that joins entity objects, you can add secondary entity objects to the list. To create more complex entity-based view objects, see How to Create Joins for Entity-Based View Objects.
4. On the Attributes page, select at least one attribute from the entity usage in the **Available** list and shuttle it to the **Selected** list. Attributes you do not select will not be eligible for use in view criteria and sort criteria. Click **Next**.

   You should select any attribute that you intend to customize (in the Attribute Settings page) or any attribute that you intend to use in a view criteria or sort criteria (in the Query page). Additionally, the tables that appear in the `FROM` list of the runtime-generated query will be limited to the tables corresponding to the attributes of the entity objects you select.
5. On the Attribute Settings page, use the **Select Attribute** dropdown list to switch between the view object attributes and deselect **Selected in Query** for each attribute that you want to defer adding to the SQL statement until runtime.

   By default, the **Selected in Query** checkbox is enabled for all view object attributes that you add in declarative SQL mode. This default setting will generate a SQL statement with all added attributes selected. When you deselect the checkbox for an attribute, the `IsSelected` property is set to `false` and whether or not the attribute is selected will be determined at runtime by the databound UI component's usage of the attribute.
6. Optionally, change the attribute names or any of their initial settings. Click **Next**.

7. On the Query page, select **Calculate Optimized Query at Runtime** if it is not already displayed. The wizard changes to declarative SQL mode.

   The Create View Object wizard enables declarative SQL mode by default. The default mode allows you to create entity-based view objects that rely on no SQL in the design time. Declarative SQL mode is an alternative to custom SQL mode, which lets you create SQL statements directly in the Create View Object wizard. Deselecting **Calculate Optimized Query at Runtime** enables custom SQL mode.

8. Optionally, define **Where** and **Sort By** criteria (Using the Sort Panel) to filter and sort the data as required. At runtime, ADF Business Components automatically generates the corresponding SQL statements based on the criteria you create.

   Click **Edit** next to the **Where** field to define the view criteria you will use to filter the data. The view criteria you enter will be converted at runtime to a `WHERE` clause that will be enforced on the query statement. For information about specifying view criteria, see Working with Named View Criteria.

   In the **Sort By** panel select the desired attribute in the **Available** list and shuttle it to the **Selected** list. Add additional attributes to the **Selected** list when you want the results to be sorted by more than one column. Arrange the selected attributes in the list according to their sort precedence. Then for each sort attribute, assign whether the sort should be performed in ascending or descending order by selecting **Ascending** or **Descending** in **Sort Order**. Assigning the sort order to each attribute ensures that attributes ignored by the UI component still follow the intended sort order.

9. Click **Finish**.

How to Filter Declarative SQL-Based View Objects When Table Joins Apply

When you create an entity-based view object you can reference more than one entity object in the view object definition. In the case of view objects you create in declarative SQL mode, whether the base entity objects are activated from the view object definition will depend on the requirements of the databound UI component at runtime. If the UI component displays attribute values from multiple entity objects, then the SQL generated at runtime will contain a `JOIN` operation to query the appropriate tables.

Just as with any view object that you create, it is possible to filter the results from table joins by applying named view criteria. In the case of normal mode view objects, all entity objects and their attributes will be referenced by the view object definition and therefore will be automatically included in the view object's SQL statement. However, by delaying the SQL generation until runtime with declarative SQL mode, there is no way to know whether the view criteria should be applied.

> **Note:**
>
> In declarative SQL mode, you can define a view criteria to specify the `WHERE` clause (optional) when you create the view object definition. This type of view criteria when it exists by default will be applied at runtime. For a description of this usage of the view criteria, see How to Create Declarative SQL View Objects.

Because the UI component may not display sufficient attributes to cause a SQL `JOIN` for a view object with multiple entity objects defined in declarative SQL mode, named view criteria that you define to filter query results should be applied conditionally at runtime. In other words, named view criteria that you create for declarative SQL-based view objects need not be applied as required, automatic filters. To support declarative SQL mode, named view criteria that you apply to a view object created in declarative SQL mode can be set to apply only on the condition that the UI component is bound to the attributes referenced by the view criteria. The named view criteria once applied will, however, support the UI component's need to display a filtered result set.

Before you begin:

It may be helpful to have an understanding of SQL independence at runtime. For more information, see Working with View Objects in Declarative SQL Mode.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for View Objects.

You will need to complete these tasks:

- Create the view object with declarative SQL mode enabled, as described in How to Create Declarative SQL View Objects.
- Create a view criteria to filter the results from table joins, as described in How to Create Named View Criteria Declaratively.

  You will use the Edit View Criteria dialog to create the named view criteria by selecting one or more attributes from the joined entity objects. You then enable its conditional usage by setting **AppliedIfJoinSatisfied** in the Properties window.

To define a view criteria to filter only when the join is satisfied:

1. In the Applications window, double-click the declarative SQL view object that supports table joins.
2. In the overview editor, click the **View Criteria** navigation tab.
3. In View Criteria page, select the view criteria you created to filter the declarative SQL view object.

   The join view criteria must refer only to attributes from the joined entity objects.
4. In the Properties window, when you have selected one or more attributes from the joined entity objects, select **true** from the **AppliedIfJoinSatisfied** dropdown list.

   The property value **true** means you want the view criteria to be applied only on the condition that the UI component requires the attributes referenced by the view criteria. The default value **false** means that the view criteria will automatically be applied at runtime. In the case of declarative SQL mode-based view objects, the value **true** ensures that the query filter will be appropriate to needs of the view object's databound UI component.

How to Filter Master-Detail Related View Objects with Declarative SQL Mode

Just as with normal mode view objects, you can link view objects that you create in declarative SQL mode to other view objects to form master-detail hierarchies of any complexity. The steps to create the view links are the same as with any other entity-based view object, as described in How to Create a Master-Detail Hierarchy

Based on Entity Associations. However, in the case of view objects that you create in declarative SQL mode, you can further refine the view object results in the Source SQL or Destination SQL dialog for the view link by selecting a previously defined view criteria in the Create View Link wizard or the overview editor for the view link.

Before you begin:

It may be helpful to have an understanding of SQL independence at runtime. For more information, see Working with View Objects in Declarative SQL Mode.

You may also find it helpful to understand functionality that can be added using other view object features. For more information, see Additional Functionality for View Objects.

You will need to complete these tasks:

1. Create the master and detail view objects with declarative SQL mode enabled, as described in How to Create Declarative SQL View Objects.
2. Define the desired view criteria for either the source (master) view object or the destination (detail) view object, as described in How to Filter Declarative SQL-Based View Objects When Table Joins Apply.
3. Create the view link, as described in How to Create a Master-Detail Hierarchy Based on Entity Associations.

To filter a view link source or view link destination:

1. In the Applications window, double-click the view link you created to form the master-detail hierarchy.
2. In the overview editor, click the **Query** navigation tab.
3. In the Query page, expand the **Source** or **Destination** section and from the **View Criteria** dropdown list, select the desired view criteria.

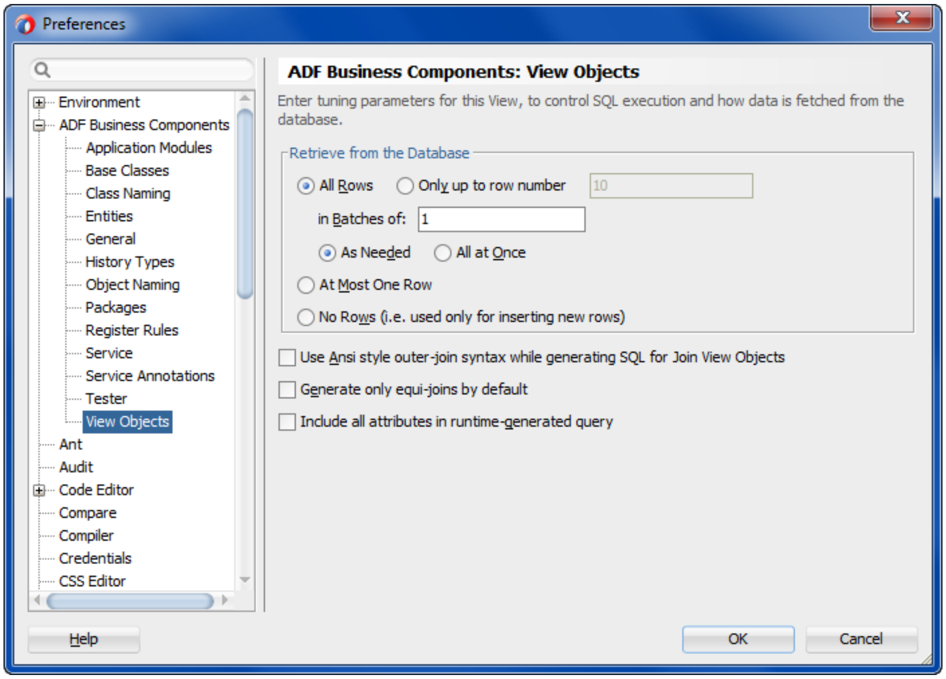
   The dropdown list will be empty if no view criteria exist for the view object.

   > **Tip:**
   >
   > If the overview editor does not display a dropdown list for view criteria selection, then the view objects you selected for the view link were not created in declarative SQL mode. For view objects created in normal or custom SQL mode, you must edit the `WHERE` clause to filter the data as required.

How to Support Programmatic Execution of Declarative SQL Mode View Objects

Typically, when you define a declarative SQL mode view object, the attributes that get queried at runtime will be determined by the requirements of the databound UI component as it is rendered in the web page. This is the runtime-generation capability that makes view objects independent of the design time database's SQL platform. However, you may also need to execute the view object programmatically without exposing it to an ADF binding in the UI. In this case, you can enable the **Include all attributes in runtime-generated query** option to ensure that a programmatically executed view object has access to all of the entity attributes. Figure 5-11 shows the global preference **Include all attributes in runtime-generated query** set to enabled.

Figure 5-11 Preferences Dialog with Declarative SQL Mode Enabled



ADF Business Components: View Objects

Enter tuning parameters for this View, to control SQL execution and how data is fetched from the database.

Retrieve from the Database

● All Rows ○ Only up to row number 10

in Batches of: 1

● As Needed ○ All at Once

○ At Most One Row

○ No Rows (i.e. used only for inserting new rows)

□ Use Ansi style outer-join syntax while generating SQL for Join View Objects

□ Generate only equi-joins by default

□ Include all attributes in runtime-generated query

> **Note:**
>
> Be careful to limit the use of the **Include all attributes in runtime-generated query** option to programmatically executed view objects. If you expose the view object with this setting enabled to a databound UI component, the runtime query will include all attributes.

The **Include all attributes in runtime-generated query** option can be specified as a global preference setting or as a setting on individual view objects. Both settings may be used in these combinations:

- Enable the global preference so that every view object you create includes all attributes in the runtime query statement.
- Enable the global preference, but disable the setting on view objects that will not be executed programmatically and therefore should not include all attributes in the runtime query statement.
- Disable the global preference (default), but enable the setting on view objects that will be executed programmatically and therefore should include all attributes in the runtime query statement.

Forcing Attribute Queries for All Declarative SQL Mode View Objects

You can enable the global preference to force attribute queries for all declarative SQL mode view objects as the default mode, or you can leave the option disabled and enable the option directly in the overview editor for a specific view object.

Before you begin:

It may be helpful to have an understanding of how the **Include all attributes in runtime-generated query** option supports programmatic execution of declarative SQL mode view objects. For more information, see How to Support Programmatic Execution of Declarative SQL Mode View Objects.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for View Objects.

To set the global preference to include all attributes in the query:

1. In the main menu, choose **Tools** and then **Preferences**.
2. In the Preferences dialog, expand **ADF Business Components** and select **View Objects**.
3. Select **Include all attributes in runtime-generated query** to force all attributes of the view object's underlying entity objects to participate in the query and click **OK**.

   Enabling this option sets a flag in the view object definition but you will still need to add entity object selections and entity object attribute selections to the view object definition.

Forcing Attribute Queries for Specific Declarative SQL Mode View Objects

When you want to force all attributes for a specific view object to participate in the view object query, you can specify this in the **Tuning** section of the General page of the overview editor. The overview editor only displays the **Include all attributes in runtime-generated query** option if you have created the view object in declarative SQL mode.

Before you begin:

It may be helpful to have an understanding of how the **Include all attributes in runtime-generated query** option supports programmatic execution of declarative SQL mode view objects. For more information, see How to Support Programmatic Execution of Declarative SQL Mode View Objects.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for View Objects.

You will need to complete this task:

> Create the view object in the Create View Object wizard and be sure that you have enabled declarative SQL mode, as described in How to Create Declarative SQL View Objects.

To set the view object-specific preference to include all attributes in the query:

1. In the Applications window, double-click the declarative SQL view object.

2. In the overview editor, click the **General** navigation tab.

   You can verify that the view object was created in declarative SQL mode in the overview editor. In the overview editor, click the **Query** navigation tab and then in the Query page, verify that the **Mode** dropdown list shows the selection **Declarative**.

3. In the General page, expand the **Tuning** section and select **Include all attributes in runtime-generated query**.

   Enabling this option forces all attributes of the view object's underlying entity objects to participate in the query. When enabled, it sets a flag in the view object definition but you will still need to add entity object selections and entity object attribute selections to the view object definition.

What Happens When You Create a View Object in Declarative SQL Mode

When you create the view object in declarative SQL mode, three properties get added to the view object's metadata: `SelectListFlags`, `FromListFlags`, and `WhereFlags`. Properties that are absent in declarative SQL mode are the normal mode view object's `SelectList`, `FromList`, and `Where` properties, which contain the actual SQL statement (or, for custom SQL mode, the `SQLQuery` element). The following example shows the three view object metadata flags that get enabled in declarative SQL mode to ensure that SQL will be generated at runtime instead of specified as metadata in the view object's definition.

```
<ViewObject
  xmlns="http://xmlns.oracle.com/bc4j"
  Name="CustomerCardStatus"
  SelectListFlags="1"
  FromListFlags="1"
  WhereFlags="1"
   ...
```

Similar to view objects that you create in either normal or custom SQL mode, the view object metadata also includes a `ViewAttribute` element for each attribute that you select in the Attribute page of the Create View Object wizard. However, in declarative SQL mode, when you "select" attributes in the wizard (or add an attribute in the overview editor), you are not creating a `FROM` or `SELECT` list in the design time. The attribute definitions that appear in the view object metadata only determine the list of potential entities and attributes that will appear in the runtime-generated statements. For information about how ADF Business Components generates these SQL lists, see What Happens at Runtime: Declarative SQL Mode Queries.

The following example shows the additional features of declarative SQL mode view objects, including the optional declarative `WHERE` clause (`DeclarativeWhereClause` element) and the optional declarative `ORDERBY` clause (`SortCriteria` element).

```
<DeclarativeWhereClause>
    <ViewCriteria
        Name="CustomerStatusWhereCriteria"
        ViewObjectName="oracle.summit.model.views.CustomerCardStatus"
        Conjunction="AND"
        Mode="3"
        AppliedIfJoinSatisfied="false">
        <ViewCriteriaRow
            Name="vcrow60">
            <ViewCriteriaItem
                Name="CardTypeCode"
                ViewAttribute="CardTypeCode"
                Operator="STARTSWITH"
                Conjunction="AND"
                Required="Optional">
            <ViewCriteriaItemValue
                Value=":cardtype"
                IsBindVarValue="true"/>
            </ViewCriteriaItem>
        </ViewCriteriaRow>
    </ViewCriteria>
    </DeclarativeWhereClause>
    <SortCriteria>
       <Sort
        Attribute="CustomerId"/>
       <Sort
        Attribute="CardTypeCode"/>
    </SortCriteria>
```

What Happens at Runtime: Declarative SQL Mode Queries

At runtime, when a declarative SQL mode query is generated, ADF Business Components determines which attributes were defined from the metadata ViewCriteria element and SortCriteria element. It then uses these attributes to generate the WHERE and Sort By clauses. Next, the runtime generates the FROM list based on the tables corresponding to the entity usages defined by the metadata ViewAttribute elements. Finally, the runtime builds the SELECT statement based on the attribute selection choices the end user makes in the UI. As a result, the view object in declarative SQL mode generates all SQL clauses entirely at runtime. The runtime-generated SQL statements will be based on the platform that appears in the project properties setting. Currently, the runtime supports SQL92 (ANSI) style and Oracle style platforms.

Note that the binding style of SQL queries will be determined by the SQL flavor chosen for the application. In practice, all queries must use the same binding style (named or positional) or SQL validation errors may result. To avoid this issue, application developers must ensure that all ADF applications that they deploy to the same managed server all use a single value for their chosen SQL platform. For more information, see How to Initialize the Data Model Project With a Database Connection.

What You May Need to Know About Working Programmatically with Declarative SQL Mode View Objects

As a convenience to developers, the view object implementation API allows individual attributes to be selected and deselected programmatically. This API may be useful in combination with the view objects you create in declarative SQL mode and intend to execute programmatically. The following example shows how to call selectAttributeDefs() on the view object when you want to add a subset of attributes to those already configured with SQL mode enabled.

```
ApplicationModule am = Configuration.createRootApplicationModule(amDef, config);
ViewObjectImpl vo = (ViewObjectImpl) am.findViewObject("CustomerVO");
vo.resetSelectedAttributeDefs(false);
vo.selectAttributeDefs(new String[] {"FirstName, "LastName"});
vo.executeQuery();
```

The call to `selectAttributeDefs()` adds the attributes in the array to a private member variable of `ViewObjectImpl`. A call to `executeQuery()` transfers the attributes in the private member variable to the actual select list. It is important to understand that these `ViewObjectImpl` attribute calls are not applicable to the client layer and are only accessible inside the `Impl` class of the view object on the middle tier.

Additionally, you might call `unselectAttributeDefs()` on the view object when you want to deselect a small subset of attributes after *enabling* the **Include all attributes in runtime-generated query** option. Alternatively, you can call `selectAttributeDefs()` on the view object to select a small subset of attributes after *disabling* the **Include all attributes in runtime-generated query** option.

> **Caution:**
>
> Be careful not to expose a declarative SQL mode view object executed with this API to the UI since only the value of the **Include all attributes in runtime-generated query** option will be honored.

Creating View Objects Populated With Static Data

Create ADF Business Components static data view objects to access a small set of static data, and avoid unnecessary querying of the database.

ADF Business Components lets you create view objects in your data model project with rows that you populate at design time. Typically, you create static data view objects when you have a small amount of data to maintain and you do not expect that data to change frequently. The decision whether to use a lookup table from the database or whether to use a static data view object based on a list of hardcoded values depends on the size and nature of the data. The static data view object is useful when you have no more than 100 entries to list. Any larger number of rows should be read from the database with a conventional table-based view object. The static data view object has the advantage of being easily translatable since attribute values are stored in a resource bundle. However, all of the rows of a static data view object will be retrieved at once and therefore, using no more than 100 entries yields the best performance.

> **Best Practice:**
>
> When you need to create a view object to access a small list of static data, you should use the static data view object rather than query the database. The static data view object is ideal for lists not exceeding 100 rows of data. Because the Create View Object wizard saves the data in a resource message file, these data are easily translatable.

Static data view objects are useful as an LOV data source when it is not desirable to query the database to supply the list of values. Suppose your order has the following statuses: open, closed, pending. You can create a static data view object with these values and define an LOV on the static data view object's status attribute. Because the wizard stores the values of the status view object in a translatable resource file, the UI will display the status values using the resource file corresponding to the application's current locale.

How to Create Static Data View Objects with Data You Enter

You use the Create View Object wizard to create static data view objects. The wizard lets you define the desired attributes (columns) and enter as many rows of data as necessary. The wizard displays the static data table as you create it.

> **Note:**
>
> Because the data in a static data view object does not originate in database tables, the view object will be read-only.

You can also use the Create View Object wizard to create the attributes based on data from a comma-separated value (CSV) file format like a spreadsheet file.

Before you begin:

It may be helpful to have an understanding of static data view objects. For more information, see Creating View Objects Populated With Static Data.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for View Objects.

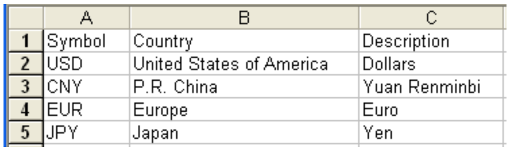
To manually create attributes for a static data view object:

1. In the Applications window, right-click the package in the data model project in which you want to create the static data view object and choose **New** and then **View Object**.
2. In the Create View Object wizard, on the Name page, enter a package name and a view object name. Select **Static list** to indicate that you want to supply static data for this view object. Click **Next**.
3. On the Attributes page, click **New** to add an attribute that corresponds to the columns in the static list table. In the New View Object Attribute dialog, enter a name and select the attribute type. Click **OK** to return to the wizard, and click **Next**.
4. On the Attribute Settings page, do nothing and click **Next**.
5. On the Static List page, click the **Add** button to enter the data directly into the wizard page. The attributes you defined will appear as the columns for the static data table.
6. On the Application Module pages, do nothing and click **Next**.
7. On the Summary page, click **Finish**.

How to Create Static Data View Objects with Data You Import

Using the Import feature of the Create View Object wizard, you can create a static data view object with attributes based on data from a comma-separated value (CSV) file format like a spreadsheet file. The wizard will use the first row of a CSV flat file to identify the attributes and will use the subsequent rows of the CSV file for the data for each attribute. For example, if your application needs to display choices for international currency, you might define the columns Symbol, Country, and Description in the first row and then add rows to define the data for each currency type, as shown in Figure 5-12.

Figure 5-12 Sample Data Ready to Import from CSV Flat File



	A	B	C
1	Symbol	Country	Description
2	USD	United States of America	Dollars
3	CNY	P.R. China	Yuan Renminbi
4	EUR	Europe	Euro
5	JPY	Japan	Yen

Before you begin:

It may be helpful to have an understanding of static data view objects. For more information, see Creating View Objects Populated With Static Data.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for View Objects.

To create attributes of a static data view object based on a flat file:

1. In the Applications window, right-click the project in which you want to create the static data view object and choose **New** and then **View Object**.
2. In the Create View Object wizard, on the Name page, enter a package name and a view object name. Select **Static list** to indicate that you want to supply static data for this view object. Click **Next**.
3. On the Attributes page, if the CSV flat file does not define column names in the first row of the file, click **New** to add an attribute that corresponds to the column names in the static data table. In the New View Object Attribute dialog, enter a name and select the attribute type. Click **OK** to return to the wizard, and click **Next**.

   The attributes you add must match the order of the data columns defined by the flat file. If you create fewer attributes than columns, the wizard will ignore extra columns during import. Conversely, if you create more attributes than columns, the wizard will define extra attributes with value null.
4. On the Attribute Settings page, do nothing and click **Next**.
5. On the Static List page, click **Import** to locate the CSV file and display the data in the wizard. Verify the data and edit the values as needed.

   If you created the attribute names yourself and the first row of the CSV file displays the column names in the first row, the column names will be displayed in the wizard as data and should be deleted. To edit an attribute value, double-click in the value field.
6. Optionally, click the **Add** button or **Remove** button to change the number of rows of data. Click **Next**.

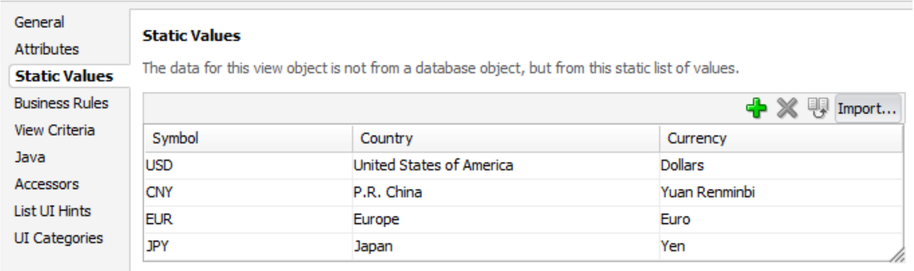
   To enter values for the attributes of a new row, double-click in the value field.
7. On the Application Module page, do nothing and click **Next**.

8. On the Summary page, click **Finish**.

What Happens When You Create a Static Data View Object

When you create a static data view object, the overview editor for the view object displays the rows of data that you defined in the wizard. You can use the editor to define additional data, as shown in Figure 5-13.

Figure 5-13 Static Values Page Displays Data



Static Values

The data for this view object is not from a database object, but from this static list of values.

Symbol	Country	Currency
USD	United States of America	Dollars
CNY	P.R. China	Yuan Renminbi
EUR	Europe	Euro
JPY	Japan	Yen

The generated XML definition for the static data view object contains one transient attribute for each column of data. For example, if you import a CSV file with data that describes international currency, your static data view object might contain a transient attribute for `Symbol`, `Country`, and `Description`, as shown in the following example.

```
<ViewObject
...
<!-- Transient attribute for first column -->
  <ViewAttribute
    Name="Symbol"
    IsUpdateable="false"
    IsSelected="false"
    IsPersistent="false"
    PrecisionRule="true"
    Precision="255"
    Type="java.lang.String"
    ColumnType="VARCHAR2"
    AliasName="Symbol"
    SQLType="VARCHAR"/>
<!-- Transient attribute for second column -->
  <ViewAttribute
    Name="Country"
    IsUpdateable="false"
    IsPersistent="false"
    PrecisionRule="true"
    Precision="255"
    Type="java.lang.String"
    ColumnType="VARCHAR"
    AliasName="Country"
    SQLType="VARCHAR"/>
<!-- Transient attribute for third column -->
  <ViewAttribute
    Name="Description"
    IsUpdateable="false"
    IsPersistent="false"
    PrecisionRule="true"
    Precision="255"
    Type="java.lang.String"
    ColumnType="VARCHAR"
    AliasName="Description"
    SQLType="VARCHAR"/>
  <StaticList
    Rows="4"
    Columns="3"/>
<!-- Reference to file that contains static data -->
  <ResourceBundle>
    <PropertiesBundle
      PropertiesFile="model.ModelBundle"/>
  </ResourceBundle>
</ViewObject>
```

Because the data is static, the overview editor displays no Query page and the generated XML definition for the static data view object contains no query statement. Instead, the `<ResourceBundle>` element in the XML definition references a resource bundle file. The following example shows the reference to the file as `PropertiesFile="model.ModelBundle"`. The resource bundle file describes the rows of data and also lets you localize the data. When the default resource bundle type is used, the file `ModelBundle.properties` appears in the data model project, as shown in the following example.

```
model.ViewObj.SL_0_0=USD
model.ViewObj.SL_0_1=United States of America
model.ViewObj.SL_0_2=Dollars
model.ViewObj.SL_1_0=CNY
model.ViewObj.SL_1_1=P.R. China
model.ViewObj.SL_1_2=Yuan Renminbi
model.ViewObj.SL_2_0=EUR
model.ViewObj.SL_2_1=Europe
model.ViewObj.SL_2_2=Euro
model.ViewObj.SL_3_0=JPY
model.ViewObj.SL_3_1=Japan
model.ViewObj.SL_3_2=Yen
```

How to Edit Static Data View Objects

When you need to make changes to the static list table, double-click the view object in the Applications window to open the overview editor for the view object. You can add and delete attributes (columns in the static list table), add or delete rows (data in the static list table), sort individual rows, and modify individual attribute values. The editor will update the view object definition file and save the modified attribute values in the message bundle file. For information about localizing message bundles, see Working with Resource Bundles.

What You May Need to Know About Static Data View Objects

The static data view object has a limited purpose in the application module's data model. Unlike entity-based view objects, static data view objects will not be updatable. You use the static data view object when you want to display read-only data to the end user and you do not want to create a database table for the small amount of data the static list table contains.

Adding Calculated and Transient Attributes to a View Object

ADF Business Components view objects can include SQL-calculated attributes and transient attributes that do not map to any entity object attributes.

In addition to having attributes that map to underlying entity objects, your view objects can include calculated attributes that don't map to any entity object attribute value. The two kinds of calculated attributes are known as:

- **SQL-calculated attributes**, when their value is retrieved as an expression in the SQL query's SELECT list
- **Transient attributes,** when their value is not retrieved as part of the query

A view object can include an entity-mapped attribute which itself is a transient attribute at the entity object level.

How to Add a SQL-Calculated Attribute

You use the overview editor for the view object to add a SQL-calculated attribute.

Before you begin:

It may be helpful to have an understanding of calculated attributes. For more information, see Adding Calculated and Transient Attributes to a View Object.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for View Objects.

You will need to complete this task:

> Create the desired view objects, as described in How to Create an Entity-Based View Object, and How to Create a Custom SQL Mode View Object.

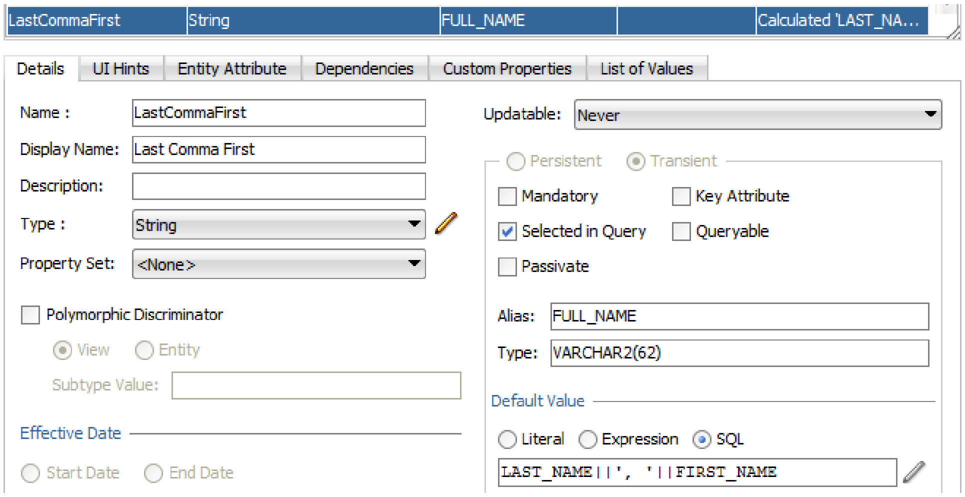
To add a SQL-calculated attribute to a view object:

1. In the Applications window, double-click the view object for which you want to define a SQL-calculated attribute.
2. In the overview editor, click the **Attributes** navigation tab and then click the **Create new attribute** button.
3. In the New View Object Attribute dialog, enter a name for the attribute.
4. In the overview editor, in the Attributes page, select the new attribute.
5. Click the **Details** tab, set the Java type to an appropriate value.

   For example, a calculated attribute that concatenates a first name and a last name would have the type `String`, as shown in Figure 5-14.
6. In the **Default Value** section, select **SQL** and provide a SQL expression in the expression field.

   For example, to change the order of first name and last name, you could write the expression `LAST_NAME||', '||FIRST_NAME`, as shown in Figure 5-14.

Figure 5-14 New SQL-Calculated Attribute



LastCommaFirst	String	FULL_NAME		Calculated 'LAST_NA...

Name : LastCommaFirst

Display Name: Last Comma First

Description:

Type : String

Property Set: <None>

☐ Polymorphic Discriminator

● View ○ Entity

Subtype Value:

○ Start Date ○ End Date

Updatable: Never

○ Persistent ● Transient

☐ Mandatory ☐ Key Attribute

☑ Selected in Query ☐ Queryable

☐ Passivate

Alias: FULL_NAME

Type: VARCHAR2(62)

○ Literal ○ Expression ● SQL

LAST_NAME||', '||FIRST_NAME

7. Consider changing the SQL column alias to match the name of the attribute.
8. Verify the database query column type and adjust the length (or precision/scale) as appropriate.

What Happens When You Add a SQL-Calculated Attribute

When you add a SQL-calculated attribute in the overview editor for the view object, JDeveloper updates the XML document for the view object to reflect the new attribute. The entity-mapped attribute's `<ViewAttribute>` tag looks like the sample shown in the following example. The entity-mapped attribute inherits most of it properties from the underlying entity attribute to which it is mapped.

```
<ViewAttribute
   Name="LastName"
   IsNotNull="true"
   EntityAttrName="LastName"
   EntityUsage="EmpEO"
   AliasName="LAST_NAME" >
</ViewAttribute>
```

Whereas, in contrast, a SQL-calculated attribute's `<ViewAttribute>` tag looks like sample shown in the following example. As expected, the tag has no `EntityUsage` or `EntityAttrName` property, and includes datatype information along with the SQL expression.

```
<ViewAttribute
   Name="LastCommaFirst"
   IsUpdatable="false"
   IsPersistent="false"
   PrecisionRule="true"
   Type="java.lang.String"
   ColumnType="VARCHAR2"
   AliasName="FULL_NAME"
   SQLType="VARCHAR" >
   Precision="62"
   Expression="LAST_NAME||', '||FIRST_NAME">
</ViewAttribute>
```

What You May Need to Know About SQL-Calculated Attributes

The view object includes the SQL expression for your SQL-calculated attribute in the `SELECT` list of its query at runtime. The database is the one that evaluates the expression, and it returns the result as the value of that column in the query. The value is reevaluated each time you execute the query.

How to Add a Transient Attribute

Transient attributes are often used to provide subtotals or other calculated expressions that are not stored in the database.

Before you begin:

It may be helpful to have an understanding of transient attributes. For more information, see Adding Calculated and Transient Attributes to a View Object.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for View Objects.

Transient attributes that you define may be evaluated using the Groovy Expression Language. Groovy lets you insert expressions and variables into strings. The expression is saved as part of the view object definition. For more information, see Using Groovy Scripting Language With Business Components.

You should be familiar with the limitations of using transient attributes. For more information, see What You May Need to Know About View Object Queries with Primary Keys Defined by Transient Attributes.

You will need to complete this task:

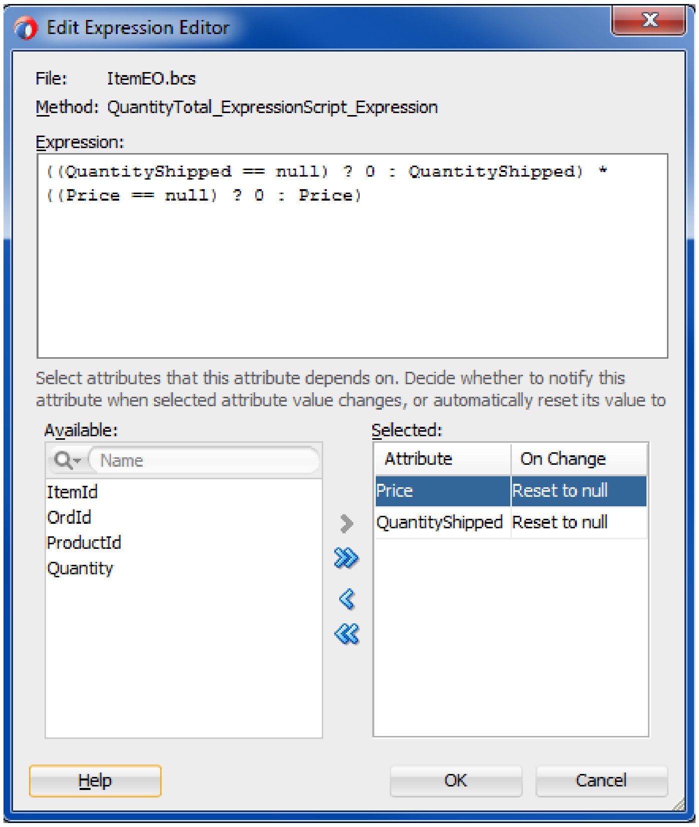
> Create the desired view objects, as described in How to Create an Entity-Based View Object, and How to Create a Custom SQL Mode View Object.

To add a transient attribute to a view object:

1. In the Applications window, double-click the view object for which you want to define a transient attribute.
2. In the overview editor, click the Attributes navigation tab and then click the **Create new attribute** button.
3. In the New View Object Attribute dialog, enter a name for the attribute and select the Java attribute type from the **Type** dropdown, then click **OK**.
4. On the Attributes page of the overview editor, click the Details tab and select the **Transient** option.
5. If the transient attribute's default value is to be calculated by an expression, under Default Value, select **Expression** checkbox and click the icon beside to open the Edit Expression Editor dialog.
6. In the Edit Expression Editor dialog, enter a default value expression that calculates the default value of the attribute.

   Transient attribute expressions may derive the value from persistent attributes that are defined by the view object. If you need to refer to an entity attribute, that attribute must be added to the view object definition.

Figure 5-15 Edit Expression Editor



File: ItemEO.bcs

Method: QuantityTotal_ExpressionScript_Expression

Expression:

```
((QuantityShipped == null) ? 0 : QuantityShipped) *
((Price == null) ? 0 : Price)
```

Select attributes that this attribute depends on. Decide whether to notify this attribute when selected attribute value changes, or automatically reset its value to

Available:

ItemId
OrdId
ProductId
Quantity

Selected:

Attribute	On Change
Price	Reset to null
QuantityShipped	Reset to null

7. Specify the attributes on which this attribute is dependent. Move these attributes from Available to Selected.

8. Click **OK** to save the expression that you entered in the Edit Expression Editor dialog.

9. You can optionally provide a condition for when to recalculate the expression in the Refresh Expression Value field. The procedure to define the recalculation expression is the same as the procedure to define the default value expression. The recalculation expression if evaluated as true, recalculates the default value expression.

   For example, the following Refresh Expression Value Groovy expression causes the attribute to be recalculated when either the `Country` attribute or the `RegionId` attribute are changed:

   ```
   return (adf.object.isAttributeChanged("Country") ||
   adf.object.isAttributeChanged("RegionId"));
   ```

> **Note:**
>
> When either the value expression or the optional recalculate expression that you define references an attribute from the base entity object, you must define this as a dependency in the Edit Expression Editor dialog. Locate each attribute in the Available list and shuttle it to the Selected list.

How to Add a Transient Attribute Defined by an Entity Object

A view object can include an entity-mapped attribute which itself is a transient attribute at the entity object level.

Before you begin:

It may be helpful to have an understanding of transient attributes. For more information, see Adding Calculated and Transient Attributes to a View Object.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for View Objects.

You will need to complete this task:

> Create the desired view objects, as described in How to Create an Entity-Based View Object, and How to Create a Custom SQL Mode View Object.

To add a transient attribute from an entity object to an entity-based view object:

1. In the Applications window, double-click the view object for which you want to add a transient attribute based on an entity usage.
2. In the overview editor, click the **Attributes** navigation tab and then click the **Create new attribute** button and choose **Add Attribute from Entity** to view the list of available entity-derived attributes.
3. In the Attributes dialog, move the desired transient attribute from the **Available** list into the **Selected** list.
4. Click **OK**.

How to Add a Validation Rule to a Transient Attribute

Attribute-level validation rules are triggered for a particular view object transient attribute when either the end user or the program code attempts to modify the attribute's value. Since you cannot determine the order in which attributes will be set, attribute-level validation rules should be used only when the success or failure of the rule depends exclusively on the candidate value of that single attribute.

The process for adding a validation rule to a view object transient attribute is similar to the way you create any declarative validation rule, and is done using the Add Validation Rule dialog. You can open this dialog from the overview editor for the view object by clicking the **Add Validation Rule** button in the Validation Rules section of the Attributes page. You must first select the transient attribute from the attributes list and the transient attribute must be defined as updatable. Validation rules cannot be defined for read-only transient attributes.

Before you begin:

It may be helpful to have an understanding of attribute UI hints. For more information, see Defining UI Hints for View Objects.

You may also find it helpful to understand the different types of validation rules you can define. For more information, see Using the Built-in Declarative Validation Rules.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for View Objects.

You will need to complete this task:

> Create the desired view objects, as described in How to Create an Entity-Based View Object, and How to Create a Custom SQL Mode View Object.

To add a validation rule for a transient attribute:

1. In the Applications window, double-click the desired view object.
2. In the overview editor, click the **Attributes** navigation tab.
3. In the Attributes page, select the transient attribute, and then click the **Details** tab and verify that **Updatable** displays **Always**.

   Validation rules for transient attributes must be updatable. The value **Never** specifies a read-only attribute.
4. In the Attributes page, click the **Validation Rules** tab and then click the **Add Validation Rule** icon.
5. In the Add Validation Rule dialog, select the type of validation rule desired from the **Type** dropdown list.
6. In the **Rule Definition** tab, use the dialog settings to configure the new rule.

   The controls will change depending on the kind of validation rule you select. For more information about the different validation rules, see Using the Built-in Declarative Validation Rules.
7. Click the **Failure Handling** tab and enter or select the error message that will be shown to the user if the validation rule fails. For more information, see Creating Validation Error Messages.
8. Click **OK**.

What Happens When You Add a Transient Attribute

When you add a transient attribute in the overview editor for a view object, JDeveloper updates the XML document for the view object to reflect the new attribute. A transient attribute's <ViewAttribute> tag in the XML is similar to the SQL-calculated one, but it lacks an Expression property.

When you base a transient attribute on a Groovy expression, a <TransientExpression> tag is created within the appropriate attribute. The following example shows XML code for a Groovy expression. The Groovy expression is added for the SalesRepName attribute of the OrdVO view object. The XML definition shows that the .bcs file where this Groovy expression script resides in is part of this XML definition: CodeSourceName=”OrdVORow”. Therefore, the .bcs file name is OrdVORow.bcs. There is also an operations.xml file, which in this example is OrdVOOperations.xml; this operations.xml file has an URI which points the user to the .bcs file.

```
<ViewAttribute
    Name="SalesRepName"
    IsUpdateable="false"
    IsSelected="false"
    IsPersistent="false"
    PrecisionRule="true"
    Type="java.lang.String"
    ColumnType="CHAR"
    AliasName="VIEW_ATTR"
    SQLType="VARCHAR">
    <TransientExpression
       Name="ExpressionScript"
       trustMode="untrusted"
       CodeSourceName="OrdVORow"/>
  </ViewAttribute>
```

What You May Need to Know About Transient Attributes and Calculated Values

A transient attribute is a placeholder for a data value. If you change the **Updatable** property of the transient attribute to **Always**, the end user can enter a value for the attribute. If you want the transient attribute to display a calculated value, then you'll typically leave the **Updatable** property set to **Never** and write a Groovy Language expression on the Attribute page of the overview editor for the view object.

Alternatively, if you want to write Java code to calculate the transient attribute, you need to enable a custom view row class and choose to generate accessor methods, in the Java dialog that you open by clicking the **Edit** button on the Java page of the overview editor for the view object. Then you would write Java code inside the accessor method for the transient attribute to return the calculated value. The following example shows the `CustomerVORowImpl.java` view row class contains the Java code to return a calculated value in the `getFirstDotLast()` method.

```
// In CustomerVORowImpl.java
public String getFirstDotLast() {
  // Commented out this original line since we're not storing the value
  // return (String) getAttributeInternal(FIRSTDOTLAST);
  return getFirstName().substring(0,1)+". "+getLastName();
}
```

What You May Need to Know About View Object Queries with Primary Keys Defined by Transient Attributes

In general, because a transient attribute will force reevaluation of the primary key attribute with each access, defining the primary key with a transient attribute may not be suitable for all use cases. In particular, Oracle recommends that you avoid using a transient attribute to define the primary key of any view object that will be bound to an **ADF Faces** component that may produce multiple queries, such as an ADF Tree component. In such situations, the performance of the application will be degraded by the need to reevaluate the transient attribute as the primary key.

Additionally, you should be aware that when your programmatic view object query relies on transient attributes for it primary key, it is possible for the user to receive a null pointer exception when they scroll the UI out of the view object's cached rows. In this case, since the view object query is not generated from a database table, your view object implementation must override the `ViewObjectImpl` classes' `retrieveByKey()` method to return the rows (or return an empty array when no rows are found).

Overriding this method will allow **ADF Business Components** to execute findByKey() to first find the requested rows from the cache. When that fails (because the primary key is a transient attribute), ADF Business Components will execute your retrieveByKey() override to perform the operations you defined to retrieve the rows that match the key coming in. The default implementation of this method tries to issue a database query to get the row(s) from the database:

```
protected Row[] retrieveByKey(ViewRowSetImpl rs, Key key,
                                  int maxNumOfRows, boolean skipWhere)
```

The method has these arguments:

maxNumOfRows is the maxNumOfRows you passed into the call to findByKey(). It may be 1 .. *n* or -1. *n* means that it's looking for *n* many rows whose key matches the one that got passed in. -1 means match all rows. Note that it is possible for the view object to have more than one row that matches the key when the key is a partial key and the view object is based on multiple entity objects.

skipWhere controls whether findByKey() should apply the same WHERE clause as the base view object. If the base view object has a WHERE clause DEPTNO = 10, if skipWhere is false, you're supposed to apply the same WHERE clause when looking for the row(s) from the backing store. If skipWhere is true, then don't bother with the WHERE clause from the base view object.

Limiting View Object Rows Using Effective Date Ranges

You can define an ADF view object to query data in a particular date range and limit view rows to fall in the purview of this date range.

Applications that need to query data over a specific date range can generate **date-effective row sets**. To define a date-effective view object you must create an entity-based view object that is based on an date-effective entity object. User control over the view object's effective date usage is supported by metadata on the view object at design time. At runtime, ADF Business Components generates the query filter that will limit the view rows to an effective date.

How to Create a Date-Effective View Object

Whether or not the query filter for an effective date will be generated depends on the value of the **Effective Dated** property displayed in the Properties window for the view object (to view the property, click the **General** tab in the overview editor for the view object and expand the **Name** section in the Properties window).

The overview editor for the view object does not display the date-effective query clause in the WHERE clause. You can use the Test Query dialog to view the clause. A typical query filter for effective dates looks like this:

```
(:Bind_SysEffectiveDate BETWEEN CustomerVO.EFFECTIVE_START_DATE AND
CustomerVO.EFFECTIVE_END_DATE)
```

At runtime, the bind value for the query can be obtained from a property of the **root application module** or can be assigned directly to the view object. In order to set the effective date for a transaction, use code similar to the following snippet:

```
am.setProperty(ApplicationModule.EFF_DT_PROPERTY_STR, new
Date("2008-10-01));
```

If you do not set `EFF_DT_PROPERTY_STR` on the application module, the current date is used in the query filter, and the view object returns the effective rows filtered by the current date.

The view object has its own transient attribute, `SysEffectiveDate`, that you can use to set the effective date for view rows. This transient attribute is generated in the view object by JDeveloper when you set the view object to be date-effective. Otherwise, the `SysEffectiveDate` attribute value for new rows and defaulted rows is derived from the application module. ADF Business Components propagates the effective date from the view row to the entity object during DML operations only.

Before you begin:

It may be helpful to have an understanding of data-effective row sets. For more information, see Limiting View Object Rows Using Effective Date Ranges.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for View Objects.

You will need to complete these tasks:

1. Create an effective dated entity object, as described in How to Store Data Pertaining to a Specific Point in Time.
2. Use the Create View Object wizard to create the entity-based view object, as described in How to Create an Entity-Based View Object.

   The view object you create should be based on the effective dated entity object you created. In the Attributes page of the wizard, be sure to add the date-effective attributes that specify the start date and end date on the entity object to the **Selected** list for the view object.

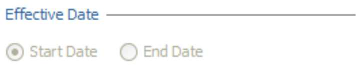
To enable effective dates for a view object using the SysEffectiveDate attribute:

1. In the Applications window, double-click the view object you created based on the effective dated entity object.
2. In the overview editor, click the **General** navigation tab.
3. In the Properties window, expand the **Name** section, and verify that the **Effective Dated** dropdown menu displays **True**.

   If the **Name** section is not displayed in the Properties window, click the **General** navigation tab in the overview editor to set the proper focus.
4. In the overview editor, click the **Attributes** navigation tab and select the attribute for the start date, and then click the **Details** tab and verify that **Effective Date** is enabled and that **Start** is selected, as shown in Figure 5-16.

   Verify that the attribute for the end date is also enabled correctly, as shown in the figure. Note that these fields appear grayed out to indicate that they cannot be edited for the view object.

Figure 5-16 View Object Overview Editor Displays Effective Date Setting



No additional steps are required once you have confirmed that the view object has inherited the desired attributes from the date-effective entity object.

How to Create New View Rows Using Date-Effective View Objects

Creating (inserting) date-effective rows is similar to creating or inserting ordinary view rows. The start date and end date can be specified as follows:

- The user specifies the effective date on the application module. The start date is set to the effective date, and the end date is set to end of time.

  End of time is defined for ADF Business Components as December 31st, 4712.

- The user specifies values for the start date and the end date (advanced).

In either case, during entity validation, the new row is checked to ensure that it does not introduce any gaps or overlaps. During post time, ADF Business Components will acquire a lock on the previous row to ensure that the gap or overlaps are not created upon the row insert.

How to Update Date-Effective View Rows

Date-effective rows are updated just as non date-effective rows are updated, using a `Row.setAttribute()` call. However, for the desired operation to take effect, an effective date mode must be set on the row before the update. ADF Business Components supports various modes to initiate the row update.

To set the update mode, invoke the `Row.setEffectiveDateMode(int mode)` method with one of the following mode constants defined by the `oracle.jbo.Row` class.

- `CORRECTION` (Correction Mode)

  The effective start date and effective end dates remain unchanged. The values of the other attributes may change. This is the standard row update behavior.

- `UPDATE` (Update Mode)

  The effective end date of the latest row will be set to the effective date. All user modifications to the row values are reverted on this row. A new row with the modified values is created. The effective start date of the new row is set to the effective date plus one day, and the effective end date is set to end of time. The new row will appear after the transaction is posted to the database.

- `OVERRIDE` (Update Override Mode)

  The effective end date of the modified row will be set to the effective date. The effective start date of the next row is set to effective date plus one day and the effective end date of the next row is set to end of time.

- `CHANGE_INSERT` (Change Insert Mode)

  Similar to update mode, but change insert mode operates on a past row not the latest row. The effective end date of the modified row should be set to the effective date. All user modifications to the row values are reverted on this row. A new row with the modified values will be created. The effective start date of the new row is set to effective date plus one day, and the effective end date is set to effective start date of the next row minus one day. The new row will appear after the transaction is posted to the database.

How to Delete Date-Effective View Rows

ADF Business Components supports various modes to initiate the row deletion. You can mark view rows for deletion by using API calls like `RowSet.removeCurrentRow()` or `Row.remove()`.

To set the deletion mode, invoke the `Row.setEffectiveDateMode(int mode)` method with one of the following mode constants defined by the `oracle.jbo.Row` class.

- `DELETE` (Delete Mode)

  The effective end date of the row is set to the effective date. The operation for this row is changed from delete to update. All rows with the same noneffective date key values and with an effective start date greater than the effective date are deleted.

- `NEXT_CHANGE` (Delete Next Change Mode)

  The effective end date of the row is set to the effective end date of the next row with the same noneffective date key values. The operation for this row is changed from delete to update. The next row is deleted.

- `FUTURE_CHANGE` (Delete Future Change Mode)

  The effective end date of the row is set to the end of time. The operation for this row is changed from delete to update. All future rows with the same noneffective date key values are deleted.

- `ZAP` (Zap Mode)

  All rows with the same non-effective date key values are deleted.

The effective date mode constants are defined on the row interface as well.

What Happens When You Create a Date-Effective View Object

When you create a date-effective view object, the view object inherits the transient attribute `SysEffectiveDate` from the entity object to store the effective date for the row. Typically, the insert/update/delete operations modify the transient attribute while Oracle ADF decides the appropriate values for effective start date and effective end date.

The query displayed in the overview editor for the date-effective view object does not display the `WHERE` clause needed to filter the effective date range. To view the full query for the date-effective view object, including the `WHERE` clause, edit the query and click **Test and Explain** in the Query page of the overview editor. The following sample shows a typical query and query filter for effective dates:

```
SELECT OrdersVO.ORDER_ID,        OrdersVO.CREATION_DATE,
       OrdersVO.LAST_UPDATE_DATE
FROM ORDERS OrdersVO
WHERE (:Bind_SysEffectiveDate BETWEEN OrdersVO.CREATION_DATE AND
                                      OrdersVO.LAST_UPDATE_DATE)
```

The following example shows sample XML entries that are generated when you create an date-effective view object.

```
<ViewObject
...
```

```
<!-- Property that enables date-effective view object. -->
  IsEffectiveDated="true">
  <EntityUsage
    Name="Orders1"
    Entity="model.OrdersDatedEO"
    JoinType="INNER JOIN"/>
<!-- Attribute identified as the start date -->
  <ViewAttribute
    Name="CreationDate"
    IsNotNull="true"
    PrecisionRule="true"
    IsEffectiveStartDate="true"
    EntityAttrName="CreationDate"
    EntityUsage="Orders1"
    AliasName="CREATION_DATE"/>
<!-- Attribute identified as the end date -->
  <ViewAttribute
    Name="LastUpdateDate"
    IsNotNull="true"
    PrecisionRule="true"
    IsEffectiveEndDate="true"
    EntityAttrName="LastUpdateDate"
    EntityUsage="Orders1"
    AliasName="LAST_UPDATE_DATE"/>
<!-- The SysEffectiveDate transient attribute -->
  <ViewAttribute
    Name="SysEffectiveDate"
    IsPersistent="false"
    PrecisionRule="true"    Type="oracle.jbo.domain.Date"
    ColumnType="VARCHAR2"
    AliasName="SysEffectiveDate"
    Passivate="true"
    SQLType="DATE"/>
</ViewObject>
```

What You May Need to Know About Date-Effective View Objects and View Links

Effective dated associations and view links allow queries to be generated that take the effective date into account. The effective date of the driving row is passed in as a bind parameter during the query execution.

While it is possible to create a noneffective dated association between two entities when using the Create Association wizard or Create View Link wizard, JDeveloper will by default make the association or link effective dated if one of the ends is effective dated. However, when the association or view link exists between an effective dated and a noneffective dated object, then at runtime ADF Business Components will inspect the effective dated nature of the view object or entity object before generating the query clause and binding the effective date. The effective date is first obtained from the driving row. If it is not available, then it is obtained from the property EFF_DT_PROPERTY_STR of the root application module. If you do not set EFF_DT_PROPERTY_STR for the application module, the current date is used in the query filter on the driving row and applied to the other side of the association or view link.

Working with Multiple Tables in Join Query Results

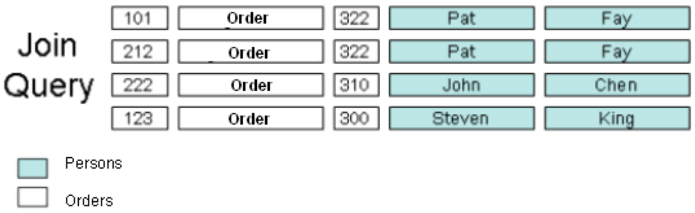
Use the Create View Object Wizard to define join queries declaratively that involves multiple tables.

Many queries you will work with will involve multiple tables that are related by foreign keys. In this scenario, you join the tables in a single view object query to show additional descriptive information in each row of the main query result. You use the Create View Object wizard to define the query using declarative options. Whether your view object is read-only or entity-based determines how you can define the join:

- When you work with entity-based view objects, the Create View Object wizard uses an existing association defined between the entities to automatically build the view object's join `WHERE` clause. You can declaratively specify the type of join you want to result from the entity objects. Inner join (equijoin) and outer joins are both supported.
- When you work with non-entity based view objects, you can use the SQL Builder dialog to build the view object's join `WHERE` clause. In this case, you must select the columns from the tables that you want to join.

Figure 5-17 illustrates the rows resulting from two tables queried by a view object that defines a join query. The join is a single flattened result.

Figure 5-17   Join Query Result



101	Order	322	Pat	Fay
212	Order	322	Pat	Fay
222	Order	310	John	Chen
123	Order	300	Steven	King

How to Create Joins for Entity-Based View Objects

It is extremely common in business applications to supplement information from a *primary* business domain object with secondary reference information to help the end user understand what foreign key attributes represent. Take the example of the `ItemEO` entity object. It contains foreign key attribute of type `Number` like:

- `ItemId`, representing the product to which the order item pertains

Showing an end user exclusively these "raw" numerical values is not very helpful. Ideally, reference information from the view object's related entity objects should be displayed to improve the application's usability. One typical solution involves performing a join query that retrieves the combination of the primary and reference information. This is equivalent to populating "dummy" fields in each queried row with reference information based on extra queries against the lookup tables.

When the end user can *change* the foreign key values by editing the data, this presents an additional challenge. Luckily, entity-based view objects support easily including reference information that's always up to date. The key requirement to leverage this feature is the presence of associations between the entity object that act as the view object's primary entity usage and the entity objects that contribute reference information.

To include reference entities in a join view object, use the Create View Object wizard. The Create View Object wizard lets you specify the type of join:

- **Inner Join**

  Select when you want the view object to return all rows between two or more entity objects, where each entity defines the same primary key column. The inner join view object will not return rows when a primary key value is missing from the joined entities.
- **Outer Join**

  Select when you want the view object to return all rows that exist in one entity object, even though corresponding rows do not exist in the joined entity object. Both left and right outer join types are supported. The left and right designation refers to the source (left) and destination (right) entity object named in an association. For details about changing the default inner join to an outer join, see How to Modify a Default Join Clause to Be an Outer Join When Appropriate. For details about cases where a matching outer entity row is not found (none exists), see What You May Need to Know About Outer Joins.

Both inner joins and outer joins are supported with the following options:

- **Reference**

  Select when you want the data from the entity object to be treated as reference information for the view object. Automatic lookup of the data is supported and attribute values will be dynamically fetched from the entity cache when a controlling key attribute changes. For details about how this setting affects runtime behavior, see What Happens at Runtime: View Row Creation.
- **Updatable**

  Deselect when you want to prevent the view object from modifying any entity attributes in the entity object. By default, the first entity object (primary) in the **Selected** list is updatable and subsequent entity objects (secondary) are not updatable. To understand how to create a join view object with *multiple* updatable entity usages, see How to Create a Subtype View Object with a Polymorphic Entity Usage.
- **Participate in row delete**

  Select when you have defined the entity as updatable and you want the action of removing rows in the UI to delete the participating reference entity object. This option is disabled for the primary entity and is used mainly on the outer joined entity. For example, while it may be possible to delete an order item, it should not be possible to delete the order when a remove row is called from the join view object. For details about how this setting affects view row deletion for various join types, see What You May Need to Know About Entity Based Joins and Participates in Row Delete and What You May Need to Know About Composition Associations and Joins.

Before you begin:

It may be helpful to have an understanding of how the type of view object effects joins. For more information, see Working with Multiple Tables in Join Query Results.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for View Objects.

You may also find it helpful to understand Oracle Database features related to join queries. See the SQL Queries and Subqueries in the *Oracle Database SQL Language Reference*.

You will need to complete this task:

> Create the desired entity objects, as described in How to Create Multiple Entity Objects and Associations from Existing Tables.

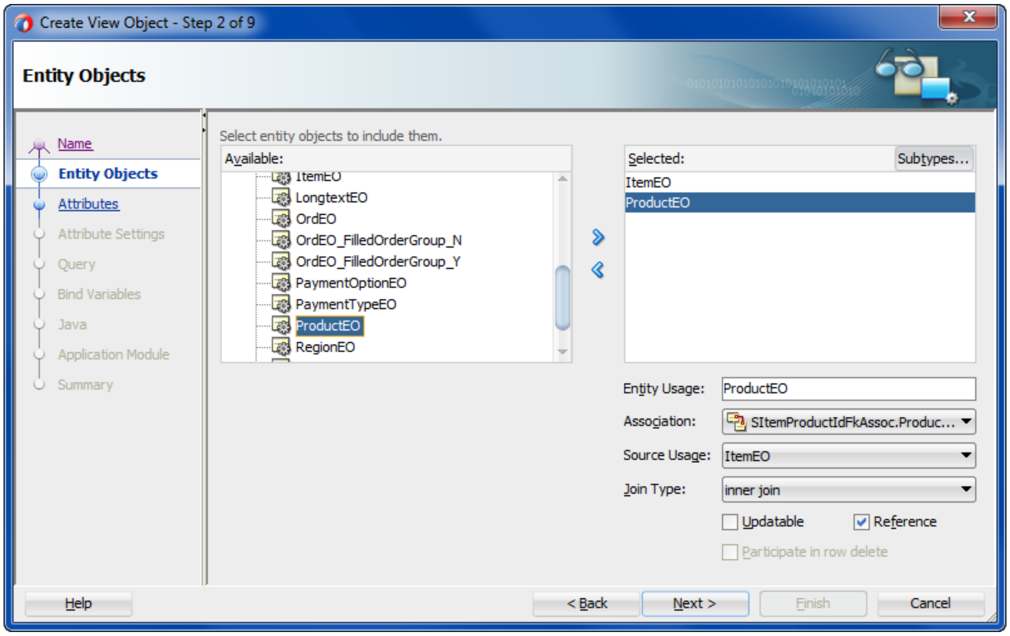
To create a view object that joins entity objects:

1. In the Applications window, right-click the package in the data model project in which you want to create the view object and choose **New** and then **View Object**.
2. In the Create View Object wizard, on the Name page, enter a package name and a view object name. Keep the default setting **Entity** enabled to indicate that you want this view object to manage data with its base entity object. Click **Next**.
3. In the Entity Objects page, the first entity usage in the **Selected** list is known as the *primary* entity usage for the view object. Select the primary entity object from the **Available** list and shuttle it to the **Selected** list.

   The list is not limited to a single, primary entity usage.
4. To add additional, secondary entity objects to the view object, select them in the **Available** list and shuttle them to the **Selected** list.

   The **Association** dropdown list shows you the name of the association that relates the selected secondary entity usage to the primary one. For example, Figure 5-18 shows the result of adding one secondary reference entity usage, `ProductEO`, in addition to the primary `ItemEO` entity usage. The association that relates to this secondary entity usage is `SItemProductIdFkAssoc.ProductEO`.

Figure 5-18 Create View Object Wizard, Entity Objects Page



Selected: ItemEO, ProductEO

Entity Usage: ProductEO

Association: SItemProductIdFkAssoc.Produc...

Source Usage: ItemEO

Join Type: inner join

☐ Updatable ☑ Reference

☐ Participate in row delete

5. Optionally, use the **Entity Usage** field to give a more meaningful name to the entity usage when the default name is not clear.

6. If you add multiple entity usages for the same entity, use the **Association** dropdown list to select which association represents that usage's relationship to the primary entity usage. Click **Next**.

   For each secondary entity usage, the **Reference** option is enabled to indicate that the entity provides reference information and that it is not the primary entity. The **Updatable** option is disabled. This combination represents the typical usage. However, when you want to create a join view object with multiple, updatable entity usages, see How to Create a Subtype View Object with a Polymorphic Entity Usage.

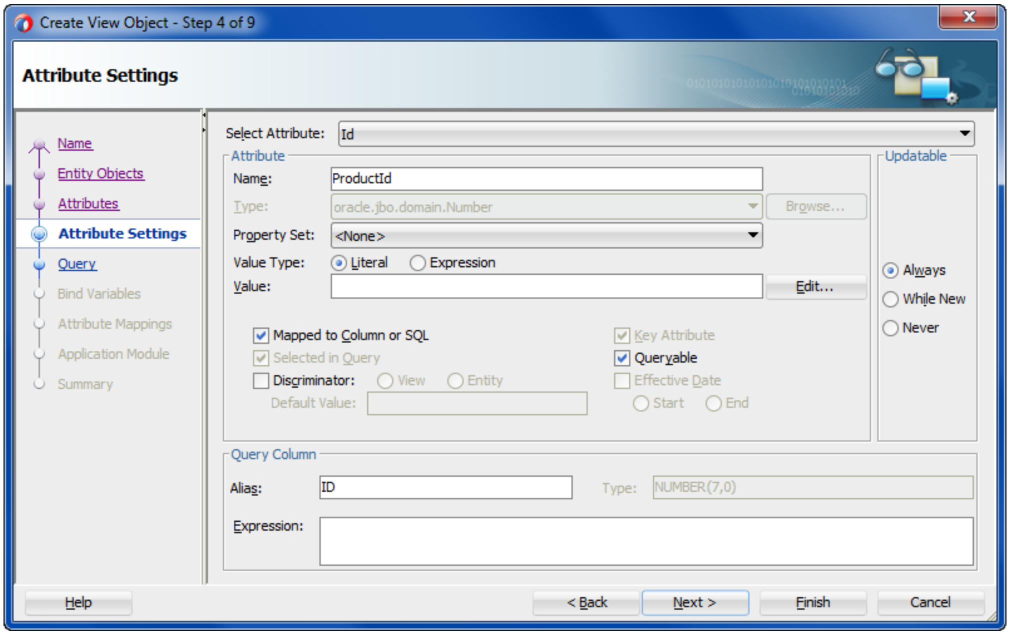
   Secondary entity usages that are updatable can also have the **Participate in row delete** option enabled. This will allow secondary entity attributes to appear `NULL` when the primary entity is displayed.

7. On the Attributes page, select the attributes you want each entity object usage to contribute to the view object. Click **Next**.

8. On the Attribute Settings page, you can rename an attribute when the names are not as clear as they ought to be.

   The same attribute name often results when the reference and secondary entity objects derive from the same table. Figure 5-19 shows the attribute `Id` in the **Select Attribute** dropdown list, which has been renamed to `ProductId` in the **Name** field.

Figure 5-19 Create View Object Wizard, Attribute Settings Page



Select Attribute: Id

Name: ProductId

Type: oracle.jbo.domain.Number

Property Set: <None>

Value Type: ● Literal ○ Expression

Value:

☑ Mapped to Column or SQL

☑ Selected in Query

□ Discriminator: ○ View ○ Entity

Default Value:

☑ Key Attribute

☑ Queryable

□ Effective Date

○ Start ○ End

Updatable: ● Always ○ While New ○ Never

Alias: ID

Type: NUMBER(7,0)

Expression:

9. Click **Finish**.

How to Modify a Default Join Clause to Be an Outer Join When Appropriate

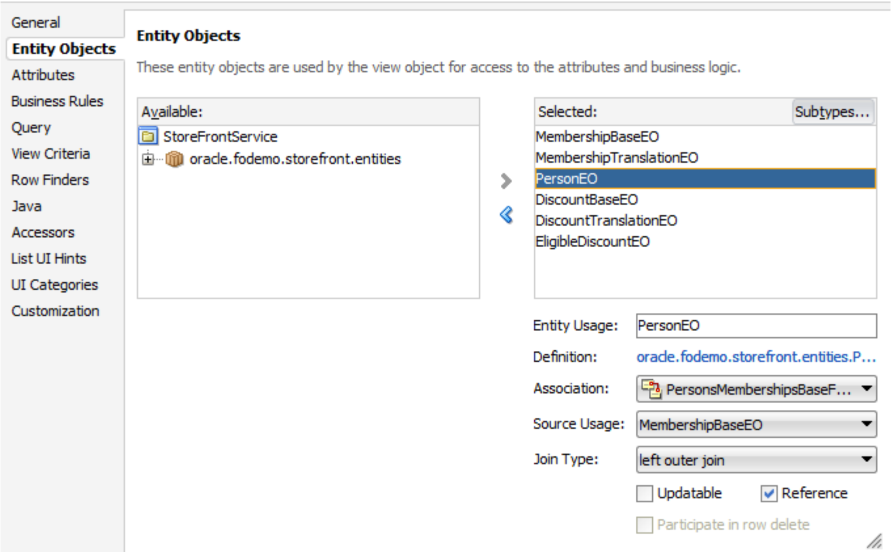
When you add a secondary entity usage to a view object, the entity usage is related to an entity usage that precedes it in the list of selected entities. This relationship is established by an entity association displayed in the **Association** dropdown list in the Entity Objects page of the overview editor for the view object. You use the **Association** dropdown list in the editor to select the entity association that relates the secondary entity usage to the desired preceding entity usage in the **Selected** list. The name of the preceding entity usage is identified in the **Source Usage** dropdown list.

When JDeveloper creates the `WHERE` clause for the join between the table for the primary entity usage and the tables for related secondary entity usages, by default it always creates inner joins. You can modify the default inner join clause to be a left or right outer join when appropriate. The left designation refers to the source entity object named in the selected association. This is the entity identified in the **Source Usage** dropdown list. The right designation refers to the current secondary entity usage that you have selected in the **Selected** list.

In the left outer join, you will include all rows from the left table (related to the entity object named in the **Source Usage** list) in the join, even if there is no matching row from the right table (related to the current secondary entity object selection). The right outer join specifies the reverse scenario: you will include all rows from the right table (related to the entity object named in the **Selected** list) in the join, even if there is no matching row from the left table (related to the current secondary entity object selection).

For example, assume that a person is not yet assigned a membership status. In this case, the `MembershipId` attribute will be `NULL`. The default inner join condition will not retrieve these persons from the `MEMBERSHIPS_BASE` table. Assuming that you want persons without membership status to be viewable and updatable through the `MembershipDiscountsVO` view object, you can use the Entity Objects page in the overview editor for the view object to change the query into an left outer join to the `MEMBERSHIPS_BASE` table for the possibly null `MEMBERSHIP_ID` column value. When you add the person entity to the view object, you would select the **left outer join** as the join type. As shown in Figure 5-20, the association `PersonsMembershipsBaseFkAssoc` identifies a source usage `MembershipBaseEO` on the left side of the join and the selected `PersonEO` entity usage on the right side. The view object `MembershipDiscountsVO` joins the rows related to both of these entity objects and defines a left outer join for `PersonEO` to allow the view object to return rows from the table related to `MembershipBaseEO` even if they do not have a match in the table related to `PersonEO`.

Figure 5-20   Setting an Outer Join to Return NULL Rows from Joined Entities



Entity Objects

These entity objects are used by the view object for access to the attributes and business logic.

Entity Usage: PersonEO

Definition: oracle.fodemo.storefront.entities.P...

Association: PersonsMembershipsBaseF...

Source Usage: MembershipBaseEO

Join Type: left outer join

☐ Updatable ☑ Reference

☐ Participate in row delete

The view object's updated `WHERE` clause includes the addition `(+)` operator on the right side of the equals sign for the related table whose data is allowed to be missing in the left outer join:

```
PersonEO.MEMBERSHIP_ID = MembershipBaseEO.MEMBERSHIP_ID(+)
```

Before you begin:

It may be helpful to have an understanding of how the type of view object effects joins. For more information, see Working with Multiple Tables in Join Query Results.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for View Objects.

You will need to complete this task:

> Create the desired entity objects, as described in How to Create Multiple Entity Objects and Associations from Existing Tables.

To change an inner join type to an outer join:

1. In the Applications window, double-click the view object that you want to modify.
2. In the overview editor, click the **Entity Objects** navigation tab.

   The entity object you select represents the table on the right side of the join.
3. In the Entity Objects page, in the **Selected** list, select the entity object that you want to change the join type for.

   The entity object you select represents the table on the right side of the join.
4. In the **Association** dropdown list, if only one association is defined, leave it selected; otherwise, select among the list of entity object associations that relate the secondary entity object to the desired entity object.

   The entity usage that represents the joined table will be displayed in the **Source Usage** dropdown list. The entity object in the **Source Usage** dropdown list that you choose through the association selection represents the table on the left side of the join.

5. In the **Join Type** dropdown list, decide how you want the view object to return rows from the joined entity objects:

   - **left outer join** will include rows from the left table in the join, even if there is no matching row from the right table.
   - **right outer join** will include rows from the right table in the join, even if there is no matching row from the left table.

   The **Source Usage** dropdown list is the left side of the join and the current entity usage in the **Selected** list is the right side.

How to Select Additional Attributes from Reference Entity Usages

After adding secondary entity usages, you can use the overview editor for the view object to select the specific, additional *attributes* from these new usages that you want to include in the view object.

> **Tip:**
>
> The overview editor lets you sort attributes displayed in the Attributes page by their entity usages. By default, the attributes table displays attributes in the order they appear in the underlying entity object. To sort the attributes by entity usage, click the header for the **Entity Usage** column of the attributes table. If the **Entity Usage** column does not appear in the attributes table, click the dropdown menu on the top-right corner of the table (below the button bar) and choose **Select Columns** to add the column to the **Selected** list.

Before you begin:

It may be helpful to have an understanding of how the type of view object effects joins. For more information, see Working with Multiple Tables in Join Query Results.

You may also find it helpful to understand Oracle Database features related to join queries. For more information, see the SQL Queries and Subqueries in the *Oracle Database SQL Language Reference*.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for View Objects.

You will need to complete this task:

Create the desired entity objects, as described in How to Create Multiple Entity Objects and Associations from Existing Tables.

To select attributes from a secondary entity usage:

1. In the Applications window, double-click the entity-based view object that you want to modify.

2. In the overview editor, click the **Attributes** navigation tab and then click the **Create new attribute** button and choose **Add Attribute from Entity** to view the list of available entity-derived attributes.
3. In the Attributes dialog, select the desired attribute and add it to the **Selected** list.

   Note that even if you didn't intend to include them, JDeveloper automatically verifies that the primary key attribute from each entity usage is part of the **Selected** list. If it's not already present in the list, JDeveloper adds it for you. When you are finished, the overview editor Query page shows that JDeveloper has included the new columns in the `SELECT` statement.
4. Click **OK**.

How to Remove Unnecessary Key Attributes from Reference Entity Usages

The view object attribute corresponding to the primary key attribute of the primary entity usage acts as the primary key for identifying the view row. When you add secondary entity usages, JDeveloper marks the view object attributes corresponding to their primary key attributes as part of the view row key as well. When your view object consists of a *single* updatable primary entity usage and a number of reference entity usages, the primary key attribute from the primary entity usage is enough to uniquely identify the view row. Further key attributes contributed by secondary entity usages are not necessary and you should disable their **Key Attribute** settings.

For example, based on the view object with primary entity usage `ShippingOptionEO`, you could disable the **Key Attribute** property for the `ShippingOptionTranslationEO` entity usage so that this property is no longer selected for this additional key attribute: `ShippingTranslationsId`.

However, when your view object consists of *multiple* updatable primary entity usages, then the primary key to identify the view row must consist at the minimum of *all* the key attributes of *all* updatable entity usages included in the view object.

Before you begin:

It may be helpful to have an understanding of how the type of view object effects joins. For more information, see Working with Multiple Tables in Join Query Results.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for View Objects.

You will need to complete this task:

Create the desired entity objects, as described in How to Create Multiple Entity Objects and Associations from Existing Tables.

To remove unnecessary key attributes:

1. In the Applications window, double-click the entity-based view object that you want to modify.
2. In the overview editor, click the **Attributes** navigation tab.
3. In the Attributes page, in the attributes table, select the key attribute (identified by the key icon in the **Name** column), and then click the **Details** tab and deselect the **Key Attribute** property.

How to Hide the Primary Key Attributes from Reference Entity Usages

Since you generally won't want to display the primary key attributes that were automatically added to the view object, you can set the attribute's **Display Hint** property to **Hide**.

Before you begin:

It may be helpful to have an understanding of how the type of view object effects joins. For more information, see Working with Multiple Tables in Join Query Results.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for View Objects.

You will need to complete this task:

Create the desired entity objects, as described in How to Create Multiple Entity Objects and Associations from Existing Tables.

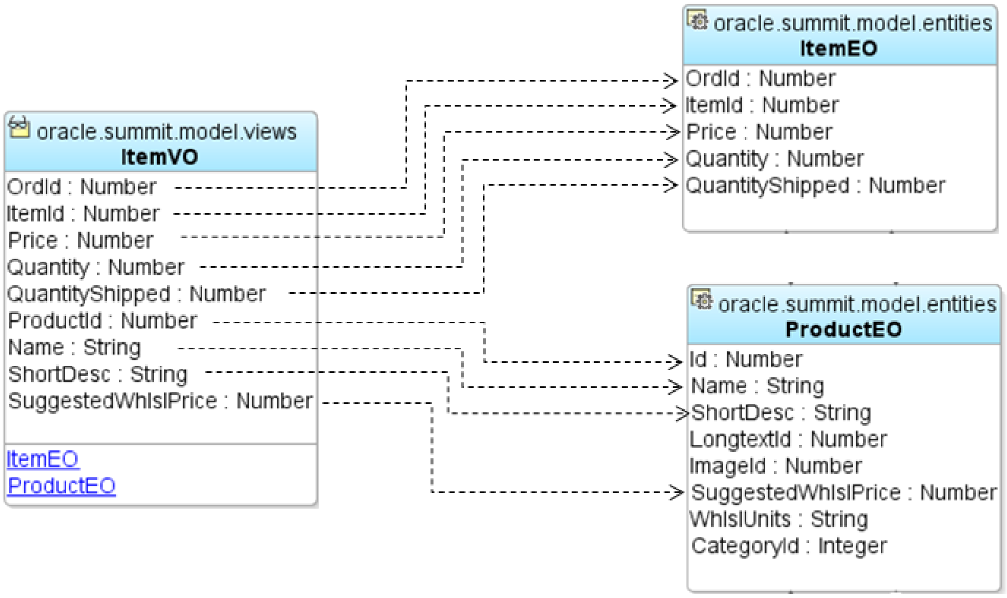
To hide the primary key attribute:

1. In the Applications window, double-click the entity-based view object that you want to modify.
2. In the overview editor, click the **Attributes** navigation tab.
3. In the Attributes page, select the primary key attribute (identified by the key icon in the **Name** column), and then click the **UI Hints** tab and select **Hide** for the **Display** type.

What Happens When You Reference Multiple Entities in a View Object

Figure 5-7 depicts the entity-based view object `ItemVO` and the two entity usages referenced in its query statement. The dotted lines represent the metadata captured in the entity-based view object's XML document that map `SELECT` list columns in the query to attributes of the entity objects used in the view object. The query of the entity-based view object joins data from a primary entity usage (`ItemEO`) with that from the secondary reference entity usage (`ProductEO`).

Figure 5-21 View Object Maps Attributes of Multiple Entity Objects



When you create a join view object to include one or more secondary entity usages by reference, JDeveloper updates the view object's XML document to include information about the additional entity usages. For example, the `ItemVO.xml` file for the view object includes an additional reference entity usage. You will see this information recorded in the multiple <`EntityUsage`> elements. For example, Figure 5-5 shows an entity usage entry that defines the primary entity usage.

```
<EntityUsage
   Name="ItemEO"
   Entity="oracle.summit.model.entities.ItemEO"/>
```

The secondary reference entity usages will have a slightly different entry, including information about the association that relates it to the primary entity usage, like the entity usage shown in the following example.

```
<EntityUsage
   Name="ProductEO"
   Entity="oracle.summit.model.entities.ProductEO"
   Association="oracle.summit.model.entities.assoc.SItemProductIdFkAssoc"
   AssociationEnd="oracle.summit.model.entities.assoc.
                    SItemProductIdFkAssoc.ProductEO"
   SourceUsage="oracle.summit.model.views.ItemVO.ItemEO"
   ReadOnly="true"
   Reference="true"
   Participant="false"
   JoinType="INNER JOIN/>
```

Each attribute entry in the XML file indicates which entity usage it references. For example, the entry for the `OrdId` attribute in the following example shows that it's related to the `ItemEO` entity usage, while the `Name` attribute is related to the `ProductEO` entity usage.

```
   <ViewAttribute
      Name="OrdId"
      IsNotNull="true"
      EntityAttrName="OrdId"
      EntityUsage="ItemEO"
      AliasName="ORD_ID" >
   </ViewAttribute>
...
   <ViewAttribute
      Name="Name"
      IsUpdatable="true"
      IsNotNull="true"
      EntityAttrName="Name"
      EntityUsage="ProductEO"
      AliasName="NAME" >
   </ViewAttribute>
```

The Create View Object wizard uses this association information at design time to automatically build the view object's join WHERE clause. It uses the information at runtime to enable keeping the reference information up to date when the end user changes foreign key attribute values.

What You May Need to Know About Outer Joins

If you specify an outer join and no matching outer entity row is found, the database return null for that portion of the query. For example, if you join Dept entity and Emp entity through DeptEmpView with an outer join and Dept 40 has no employees. Then, when the row for Dept 40 is reached, database returns [40, null] and ADF Business Components will create an entity row for Dept 40 and an blank Emp entity row. If the Emp entity in the view object is set as **Updatable**, you can set attribute values into this blank entity row. Later if you commit the transaction, a new entity row will be inserted.

What You May Need to Know About Entity Based Joins and Entity Cardinality

Suppose you have Customer and Order entity objects and you create view object joins based on the entity association CustOrderIdFKAssoc, where Customer has cardinality many and Order has cardinality 1.

If you were to create a 1-many join OrderCustVO view object that joins Order entity (the primary entity with cardinality of 1) to Customer entity (secondary entity with cardinality of many), the application may return the same Customer entity row for many Order rows.

If you were to create a many-1 join CustOrderVO view object that joins Customer (primary, many) to Order (secondary, one), the application may return the same Customer entity row with many Order entity rows.

In the OrderCustVO (1-many) case, where Customer is a reference entity, when the end user deletes a row, the application only deletes the Order entity row and Customer is unaffected. Similarly, in the CustOrderView (m-1 case), where Order is a reference entity, when the end user deletes a row, the application only deletes the Customer entity row and Order is unaffected.

But, in the CustOrderVO (many-1) case, when the end user deletes one row, the application may remove multiple rows from the join view instance. For example, suppose you have three orders (A, B, C) for customer 10. If the end user deletes the first of these joined view rows [10, A], the application will actually remove three view rows: [10, A], [10, B], and [10, C]. At the entity level, only `Customer` 10 entity row is removed because `Order` is a reference entity.

What You May Need to Know About Entity Based Joins and Participates in Row Delete

Suppose you create a 1-many join (like `OrderCustVO` view object) and a many-1 join (like `CustOrderVO` view object) where the secondary entity object has **Participate in Row Delete** enabled (and `Customer` 10 has `Orders` A, B, C).

If you were to create a 1-many join `OrderCustVO` and the `Customer` entity object has **Participate in Row Delete** enabled, when the end user deletes [A, 10], the application will delete `Order` A entity row first, followed by the `Customer` 10 entity row. When this `Customer` 10 is deleted, the application actually remove all view rows that refer to that entity row. This mean [A, 20] and [A, 30] will also be removed (3 view rows in total).

If you were to create a many-1 join `CustOrderVO` and the `Order` entity object has **Participate in Row Delete** enabled, when the end user deletes [10, A], the application will delete `Customer` 10 entity row first. The application will then remove all view rows that refer to that entity row. This means [20, A] and [30, A] will be removed from the view. Then, the application will delete the `Order` A entity row.

What You May Need to Know About Composition Associations and Joins

Suppose the association between entity objects `Order` and `OrderItem` is a composition association and you create a view object joins based on these entities, where `Order` A contains `Items` 1, 2, 3.

In the `ItemOrdVO` view object case, where `Order` (secondary) is a reference entity, when the end user deletes [1, A], the application will only delete the 1 `OrderItem` entity row and there will be no issue. However, when the `Order` entity object has **Participate in Row Delete** enabled, when the end user deletes [1, A], the application will also attempt to delete the `Item` 1 entity row, followed by deleting `Order` A. However, due to the composition relationship, this will fail and the application will throw `RemoveWithDetailsException` because `Order` A still has `OrderItem` 2 and `OrderItem` 3 that belong to `Order` A.

In the `OrdItemVO` view object case, where Item (secondary) entity is a reference entity, when the end user deletes [A, 1], the application will also attempt to delete the `Order` A entity row. However, due to the composition relationship, this will fail and the application will throw `RemoveWithDetailsException` because `Order` A still has `Item` 1, `Item` 2, and `Item` 3 that belong to `Order` A.

If the `Item` (secondary) entity object has **Participate in Row Delete** enabled, removing [A, 1] will produce the same exception since `Order` A has child rows.

The situation is different if the entity association has **Implement Cascade Delete** under **Composition Association** enabled.

In the `ItemOrderVO` view object case, where `Order` is a reference entity, when the end user deletes [1, A], the application only deletes the `Item` A entity row (with no exception thrown). If the `Order` entity object has **Participate in Row Delete** enabled, when the end user deletes [1, A], the application will delete `Item` A first, followed by deletion of `Order` A. Because cascade delete is enabled, deletion of `Order` A will delete all the remaining children of `Order` A, including `Item` 2 and `Item` 3. This will result in removal of view rows [2, A] and [3, A] from the view instance.

Similarly, in the `OrderItemVO` view object case, where `Item` is a reference entity, when the end user deletes [A, 1], the application deletes `Order` A. Because **Implement Cascade Delete** is enabled, the application will also delete all the children of `Order` A, including `Item` 1, 2, and 3. This will result in the removal of view rows [A, 2], [A, 2], and [A, 3] from the view instance.

If the `Item` entity object has **Participate in Row Delete** enabled, the same behavior is produced: all three view rows are removed.

How to Create Joins for Read-Only View Objects

To create a read-only view object joining two tables, use the Create View Object wizard.

Before you begin:

It may be helpful to have an understanding of how the type of view object effects joins. For more information, see Working with Multiple Tables in Join Query Results.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for View Objects.

To create a read-only view object joining two tables:

1. In the Applications window, right-click the project in which you want to create the view object and choose **New** and then **View Object**.
2. If you have not yet created a database connection for the data model project, in the Initialize Business Components Project dialog, select the database connection or choose **New** to create a connection. Click **OK**.

   If this is the first component you're creating in the project, the Initialize Business Components Project dialog appears to allow you to select a database connection.
3. In the Create View Object wizard, on the Name page, enter a package name and a view object name. Select **Custom SQL query** to indicate that you want this view object to manage data with read-only access. Click **Next**.
4. On the Query page, use one of the following techniques to create the SQL query statement that joins the desired tables:
   - Type or paste any valid SQL statement into the **Select** box.
   - Click **Query Builder** to open the SQL Statement dialog and build the query statement, as described in How to Use the SQL Statement Dialog with Read-Only View Objects.
5. After entering or building the query statement, click **Next**.
6. On the Bind Variables page, do one of the following:
   - If the query does not reference any bind variables, click **Next** to skip Step 3.
   - To add a bind variable and work with it in the query, see How to Add WHERE Clause Bind Variables to a View Object Definition.
7. Click **Next** on the Attribute Mappings page and the Attribute page.

8. On the Attribute Settings page, click **Finish**.

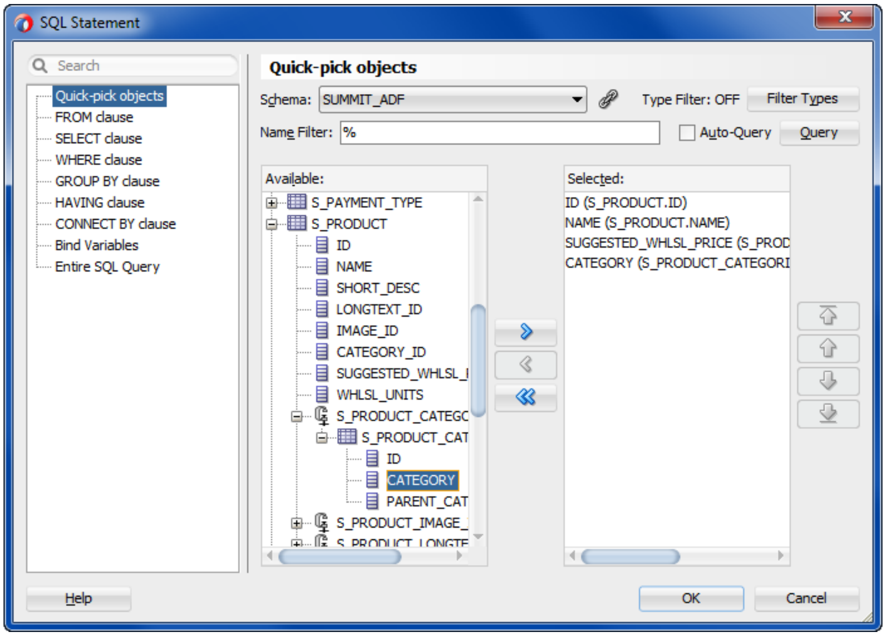
How to Test the Join View

To test the new view object, edit the application module and on the Data Model page add an instance of the new view object to the data model. Then, use the Oracle ADF Model Tester to verify that the join query is working as expected. For details about editing the data model and running the Oracle ADF Model Tester, see Testing View Object Instances Using the Oracle ADF Model Tester.

How to Use the SQL Statement Dialog with Read-Only View Objects

The Quick-pick objects page of the SQL Statement dialog lets you view the tables in your schema, including the foreign keys that relate them to other tables. To include columns in the select list of the query, shuttle the desired columns from the **Available** list to the **Selected** list. For example, Figure 5-22 shows the result of selecting the `ID`, `NAME`, and `SUGGESTED_WHLSL_PRICE` columns from the `S_PRODUCT` table, along with the `CATEGORY` column from the `S_PRODUCT_CATEGORIES` table. The column from the second table appears, beneath the `S_PRODUCT_CATEGORY_ID_FK` foreign key in the **Available** list. When you select columns from tables joined by a foreign key, the SQL Statement dialog automatically determines the required join clause for you.

Figure 5-22 View Object SQL Statement Dialog to Define a Join



Schema: SUMMIT_ADF
Type Filter: OFF
Name Filter: %
□ Auto-Query

Selected:
ID (S_PRODUCT.ID)
NAME (S_PRODUCT.NAME)
SUGGESTED_WHLSL_PRICE (S_PROD
CATEGORY (S_PRODUCT_CATEGORI

Optionally, use the `WHERE` clause page of the SQL Statement dialog to define the expression. To finish creating the query, click **OK** in the SQL Statement dialog. The Query page of the overview editor will show a query like the one shown in the following example.

```
SELECT
    S_PRODUCT.ID ID,
    S_PRODUCT.NAME NAME,
    S_PRODUCT.SUGGESTED_WHLSL_PRICE SUGGESTED_WHLSL_PRICE,
    S_PRODUCT_CATEGORIES.CATEGORY CATEGORY
FROM
    S_PRODUCT JOIN S_PRODUCT_CATEGORIES ON S_PRODUCT.CATEGORY_ID =
                                     S_PRODUCT_CATEGORIES.ID
```

You can use the Attributes page of the Create View Object wizard to rename the view object attribute directly as part of the creation process. Renaming the view object here saves you from having to edit the view object again, when you already know the attribute names that you'd like to use. As an alternative, you can also alter the default Java-friendly name of the view object attributes by assigning a column alias, as described in Attribute Names for Calculated Expressions in Custom SQL Mode.

Working with View Objects and Custom SQL

You can define entity objects in the Normal mode and custom SQL mode depending on whether you require full control over the SELECT or FROM query clause.

When defining entity-based view objects in Normal mode, you can fully specify the WHERE and SORT BY clauses, whereas, by default, the FROM clause and SELECT list are automatically derived. The names of the tables related to the participating entity usages determine the FROM clause, while the SELECT list is based on the:

- Underlying column names of participating entity-mapped attributes
- SQL expressions of SQL-calculated attributes

When you require full control over the SELECT or FROM clause in a query, you can enable custom SQL mode.

> **Tip:**
>
> The view object editors and wizard in the JDeveloper provide full support for generating SQL from choices that you make. For example, two such options allow you to declaratively define outer joins and work in declarative SQL mode (where no SQL is generated until runtime).

How to Create a Custom SQL Mode View Object

When you need full control over the SQL statement, the Create View Object wizard lets you specify this using custom SQL mode. Primarily, the custom SQL view object that you create will be useful when you need to write Unions or Group By queries. Additionally, you can create a custom SQL view object if you need to create SQL-based validation queries used by the view object-based Key Exists validator, provided that you have marked a key attribute.

See about the tradeoffs between working with entity-based view objects and custom SQL view objects that are not based on entity objects, see Using Entity-Based View Objects for Read-Only Data.

To create a custom SQL view object, use the Create View Object wizard, which is available from the New Gallery.

Before you begin:

It may be helpful to have an understanding of view objects. See, see Populating View Object Rows from a Single Database Table.

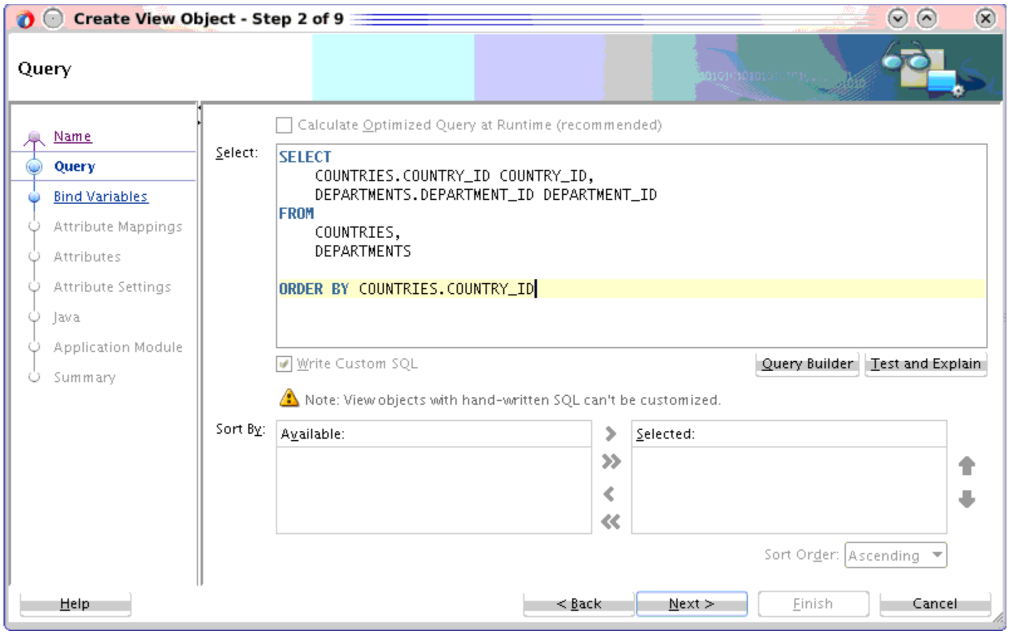
You may also find it helpful to understand functionality that can be added using other Oracle ADF features. See, see Additional Functionality for View Objects.

To create a custom SQL view object:

1. In the Applications window, right-click the package in which you want to create the view object and choose **New** and then **View Object**.
2. If you have not yet created a database connection for the data model project, in the Initialize Business Components Project dialog, select the database connection or choose **New** to create a connection. Click **OK**.

   If this is the first component you're creating in the project, the Initialize Business Components Project dialog appears to allow you to select a database connection.
3. In the Create View Object wizard, on the Name page, enter a package name and a view object name. Select **Custom SQL query** to indicate that you want this view object to manage data without the benefit of entity objects. Click **Next**.
4. On the Query page, use one of the following techniques:
   - Type or paste any valid SQL statement into the **Select** text box. The query statement can use a `WHERE` clause and an `Order By` clause. For example, Figure 5-23shows a query statement that uses a `WHERE` clause and an `Order By` clause.
   - Click **Query Builder** to open the SQL Statement dialog and build the query statement.

Figure 5-23 Create View Object Wizard, Query Page



```
SELECT
    COUNTRIES.COUNTRY_ID COUNTRY_ID,
    DEPARTMENTS.DEPARTMENT_ID DEPARTMENT_ID
FROM
    COUNTRIES,
    DEPARTMENTS

ORDER BY COUNTRIES.COUNTRY_ID
```

☑ Write Custom SQL

Note: View objects with hand-written SQL can't be customized.

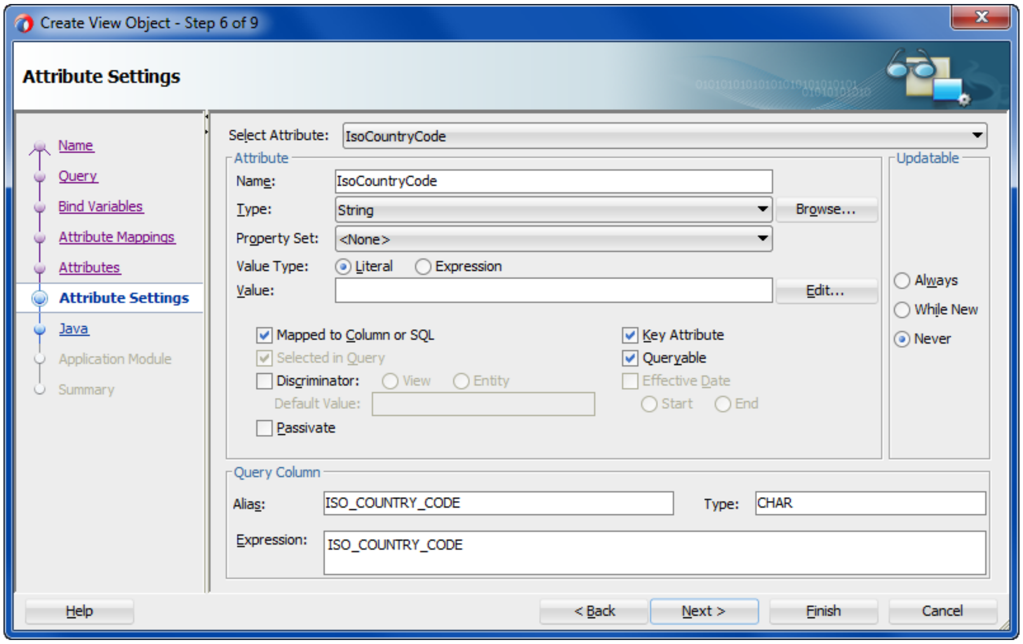
Sort Order: Ascending

> **Note:**
>
> If the Entity Objects page displays instead of the Query page, go back to Step 1 of the wizard and ensure that you've selected **Custom SQL query**.

5. After entering or building the query statement, click **Next**.
6. On the Bind Variables page, do one of the following:
   - If the query does not reference any bind variables, click **Next** to skip this page.
   - To add a bind variable and work with it in the query, see How to Add WHERE Clause Bind Variables to a View Object Definition.
7. On the Attribute Settings page, from the **Select Attribute** dropdown, select the attribute that corresponds to the primary key of the queried table and confirm that the **Key Attribute** checkbox is selected.

   The Create View Object wizard auto-detects key attributes from the database and enables the **Key Attribute** checkbox, as shown in Figure 5-24. Failure to define the key attribute can result in unexpected runtime behavior for **ADF Faces** components with a **data control** based on the view object collection.

Figure 5-24 Create View Object Wizard, Attribute Settings Page



Create View Object - Step 6 of 9

Attribute Settings

Select Attribute: IsoCountryCode

Attribute

Name: IsoCountryCode

Type: String

Property Set: <None>

Value Type: ● Literal ○ Expression

Value:

☑ Mapped to Column or SQL
☑ Selected in Query
☐ Discriminator: ○ View ○ Entity
Default Value:
☐ Passivate

☑ Key Attribute
☑ Queryable
☐ Effective Date ○ Start ○ End

Updatable

○ Always
○ While New
● Never

Query Column

Alias: ISO_COUNTRY_CODE

Type: CHAR

Expression: ISO_COUNTRY_CODE

8. Click **Finish**.

> **Note:**
>
> In the ADF Business Components wizards and editors, the default convention is to use camel-capped attribute names, beginning with a capital letter and using uppercase letters in the middle of the name to improve readability when the name comprises multiple words.

What Happens When You Create a Custom SQL View Object

When you create a view object using custom SQL mode, JDeveloper first parses the query to infer the following from the columns in the SELECT list:

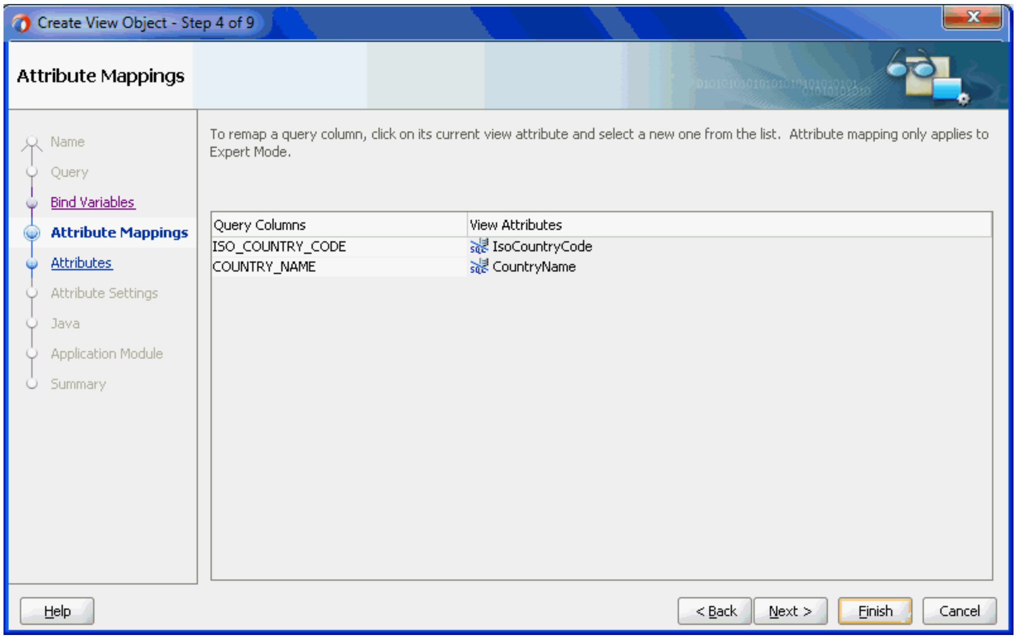
- The Java-friendly view attribute names (for example, `CountryName` instead of `COUNTRY_NAME`)

  By default, the wizard creates Java-friendly view object attribute names that correspond to the `SELECT` list column names, as shown in Figure 5-25.

  For information about using view object attribute names to access the data from any row in the view object's result set by name, see Testing View Object Instances Programmatically.

- The SQL and Java data types of each attribute

Figure 5-25 Create View Object Wizard, Attribute Mappings Page



To remap a query column, click on its current view attribute and select a new one from the list. Attribute mapping only applies to Expert Mode.

Query Columns	View Attributes
ISO_COUNTRY_CODE	IsoCountryCode
COUNTRY_NAME	CountryName

Each part of an underscore-separated column name like *`SOME_COLUMN_NAME`* is turned into a camel-capped word (like *`SomeColumnName`*) in the attribute name. While the view object attribute names correspond to the underlying query columns in the `SELECT` list, the attribute names at the view object level need not match necessarily.

Tip:

You can rename the view object attributes to any names that might be more appropriate without changing the underlying query.

JDeveloper then creates the XML document file that represents the view object's declarative settings and saves it in the directory that corresponds to the name of its package. For example, the XML file created for a view object named `CountriesVO` in the `lookups` package is `./lookups/CountriesVO.xml` under the project's source path.

To view the view object settings, select the desired view object in the Applications window and open the Structure window. The Structure window displays the list of XML definitions, including the SQL query and the list of attributes. To open the XML definition in the editor, right-click the corresponding node and choose **Go to Source**.

How to Customize SQL Statements in Custom SQL Mode

To enable custom SQL mode, select **Write Custom SQL** on the Query page of the Create View Object wizard. You can also modify the SQL statement of an existing entity-based view object in the view object overview editor. In the overview editor, navigate to the Query page and select **Write Custom SQL**.

What Happens When You Enable Custom SQL Mode

When you enable custom SQL mode, the **Select** text box in the Query page of the overview editor or wizard becomes fully editable, displaying the full SQL statement. Using this text box, you can change every aspect of the SQL query.

While you can add the ORDER BY clause directly to the **Select** text box when you are in custom SQL mode, it is still useful to keep the ORDER BY portion of the clause separate, since the runtime engine might need to append additional WHERE clauses to the query. Keeping the ORDER BY clauses separated will ensure they are applied correctly.

What You May Need to Know About Custom SQL Mode

When you define a SQL query using custom SQL mode, you type a SQL language statement directly into the editor. Using this mode places some responsibility on the Business Components developer to understand how the view object handles the metadata resulting from the query definition. Review the following information to familiarize yourself with the behavior of editing queries that you create in custom SQL mode.

Attribute Names for Calculated Expressions in Custom SQL Mode

If your SQL query includes a calculated expression, use a SQL alias to assist the Create View Object wizard in naming the column with a Java-friendly name. The following example shows a SQL query that includes a calculated expression.

```
select CUSTOMER_ID, EMAIL,
       SUBSTR(FIRST_NAME,1,1)||'. '||LAST_NAME
```

```
from CUSTOMERS
order by EMAIL
```

The following example uses a SQL alias USER_SHORT_NAME to assist the Create View Object wizard in naming the column with a Java-friendly name. The wizard will display **UserShortName** as the name of the attribute derived from this calculated expression.

```
select CUSTOMER_ID, EMAIL,
       SUBSTR(FIRST_NAME,1,1)||'. '||LAST_NAME AS USER_SHORT_NAME
from CUSTOMERS
order by EMAIL
```

Attribute Mapping Assistance in Custom SQL Mode

The automatic cooperation of a view object with its underlying entity objects depends on correct attribute-mapping metadata saved in the XML document. This information relates the view object attributes to corresponding attributes from participating entity usages. JDeveloper maintains this attribute mapping information in a fully automatic way for normal entity-based view objects. However, when you decide to use custom SQL mode with a view object, you need to pay attention to the changes you make to the SELECT list. That is the part of the SQL query that directly relates to the attribute mapping. Even in custom SQL mode, JDeveloper continues to offer some assistance in maintaining the attribute mapping metadata when you do the following to the SELECT list:

- Reorder an expression without changing its column alias

  JDeveloper reorders the corresponding view object attribute and maintains the attribute mapping.

- Add a new expression

  JDeveloper adds a new SQL-calculated view object attribute with a corresponding camel-capped name based on the column alias of the new expression.

- Remove an expression

  JDeveloper converts the corresponding SQL-calculated or entity-mapped attribute related to that expression to a transient attribute.

However, if you rename a column alias in the SELECT list, JDeveloper has no way to detect this, so it is treated as if you removed the old column expression and added a new one of a different name.

After making any changes to the SELECT list of the query, visit the Attribute Mappings page of the overview editor to ensure that the attribute-mapping metadata is correct. The table on this page, which is disabled for view objects in normal mode, becomes enabled for custom SQL mode view objects. For each view object attribute, you will see its corresponding SQL column alias in the table. By clicking into a cell in the **View Attributes** column, you can use the dropdown list that appears to select the appropriate entity object attribute to which any entity-mapped view attributes should correspond.

> **Note:**
>
> If the view attribute is SQL-calculated or transient, a corresponding attribute with a "SQL" icon appears in the **View Attributes** column to represent it. Since neither of these type of attributes are related to underlying entity objects, no entity attribute related information is required for them.

Custom Edits in Custom SQL Mode

When you disable custom SQL mode for a view object, it will return to having its `SELECT` and `FROM` clause be derived again. JDeveloper warns you that doing this might cause your custom edits to the SQL statement to be lost. If this is what you want, after acknowledging the alert, your view object's SQL query reverts back to the default.

Changes to SQL Expressions in Custom SQL Mode

Consider a `Products` view object with a SQL-calculated attribute named `Shortens` whose SQL expression you defined as `SUBSTR(NAME,1,10)`. If you switch this view object to custom SQL mode, the **Select** text box will show a SQL query similar to the one shown in the following example.

```
SELECT Products.PROD_ID,
       Products.NAME,
       Products.IMAGE,
       Products.DESCRIPTION,
       SUBSTR(NAME,1,10) AS SHORT_NAME
FROM PRODUCTS Products
```

If you go back to the attribute definition for the `Shortens` attribute and change the **SQL Expression** field from `SUBSTR(NAME,1,10)` to `SUBSTR(NAME,1,15)`, then the change will be saved in the view object's XML document. Note, however, that the SQL query in the **Select** text box will remain as the original expression. This occurs because JDeveloper never tries to *modify* the text of a custom SQL statement. In custom SQL mode, the developer is in full control. JDeveloper attempts to adjust metadata as a result of some kinds of changes you make yourself to the custom SQL statement, but it does not perform the reverse. Therefore, if you change view object metadata, the custom SQL statement is not updated to reflect it.

Therefore, you need to update the expression in the custom SQL statement itself. To be completely thorough, you should make the change *both* in the attribute metadata *and* in the custom SQL statement. This would ensure — if you (or another developer on your team) ever decides to toggle custom SQL mode *off* at a later point in time — that the automatically derived `SELECT` list would contain the correct SQL-derived expression.

> **Note:**
>
> If you find you had to make numerous changes to the view object metadata of a custom SQL mode view object, you can avoid having to manually translate any effects to the SQL statement by copying the text of your customized query to a temporary backup file. Then, you can disable custom SQL mode for the view object and acknowledge the warning that you will lose your changes. At this point JDeveloper will rederive the correct generated SQL statement based on all the new metadata changes you've made. Finally, you can enable custom SQL mode once again and reapply your SQL customizations.

SQL Calculations that Change Entity Attributes in Custom SQL Mode

If you need to present altered versions of entity-mapped attribute data, you must introduce a new SQL-calculated attribute with the appropriate expression to handle the task. In custom SQL mode, when editing the SELECT list expression that corresponds to *entity-mapped* attributes, you must not introduce SQL calculations into SQL statements that change the value of the attribute when retrieving the data.

To illustrate the problem that will occur if you introduce SQL calculations into SQL statements, consider the query for a simple entity-based view object named Products shown in the following example.

```
SELECT Products.PROD_ID,
       Products.NAME,
       Products.IMAGE,
       Products.DESCRIPTION
FROM PRODUCTS Products
```

Imagine that you wanted to limit the name column to display only the first ten characters of the name of a product query. The correct way to do that would be to introduce a new SQL-calculated field, such as ShortName with an expression like SUBSTR(Products.NAME,1,10). One way you should *avoid* doing this is to switch the view object to custom SQL mode and change the SELECT list expression for the entity-mapped NAME column to the include the SQL-calculate expression, as shown in the following example.

```
SELECT Products.PROD_ID,
       SUBSTR(Products.NAME,1,10) AS NAME,
       Products.IMAGE,
       Products.DESCRIPTION
FROM PRODUCTS Products
```

This alternative strategy would initially appear to work. At runtime, you see the truncated value of the name as you are expecting. However, if you modify the row, when the underlying entity object attempts to lock the row it does the following:

- Issues a SELECT FOR UPDATE statement, retrieving all columns as it tries to lock the row.
- If the entity object successfully locks the row, it compares the original values of all the persistent attributes in the entity cache as they were last retrieved from the database with the values of those attributes just retrieved from the database during the lock operation.

- If any of the values differs, then the following error is thrown:

  ```
  (oracle.jbo.RowInconsistentException)
  JBO-25014: Another user has changed the row with primary key [...]
  ```

If you see an error like this at runtime even though you are the *only* user testing the system, it is most likely due to your inadvertently introducing a SQL function in your custom SQL view object that changed the selected value of an entity-mapped attribute. In the above example, the `SUBSTR(Products.NAME,1,10)` function introduced causes the original selected value of the `Name` attribute to be truncated. When the row-lock SQL statement selects the value of the `NAME` column, it will select the entire value. This will cause the comparison shown in the above example to fail, producing the "phantom" error that another user has changed the row.

The same thing would happen with `NUMBER`-valued or `DATE`-valued attributes if you inadvertently apply SQL functions in custom SQL mode to truncate or alter their retrieved values for entity-mapped attributes.

Formatting of the SQL Statement in Custom SQL Mode

When you change a view object to custom SQL mode, its XML document changes from storing parts of the query in separate XML attributes, to saving the entire query in a single `<SQLQuery>` element. The query is wrapped in an XML `CDATA` section to preserve the line formatting you may have done to make a complex query be easier to understand.

Query Clauses in Custom SQL Mode

If your custom SQL view object:

- Contains a `ORDERBY` clause specified in the **Order By** field of the Query page of the overview editor at design time, or
- Has a dynamic `WHERE` clause or `ORDERBY` clause applied at runtime using `setWhereClause()` or `setOrderByClause()`

Then its query gets nested into an inline view before applying these clauses. For example, suppose your custom SQL query was defined like the one shown in the following example.

```
select CUSTOMER_ID, EMAIL, FIRST_NAME, LAST_NAME
from CUSTOMERS
union all
select CUSTOMER_ID, EMAIL, FIRST_NAME, LAST_NAME
from INACTIVE_CUSTOMERS
```

Then, at runtime, when you set an additional `WHERE` clause like `email = :TheUserEmail`, the view object nests its original query into an inline view like the one shown in the following example.

```
SELECT * FROM(
select CUSTOMER_ID, EMAIL, FIRST_NAME, LAST_NAME
from CUSTOMERS
union all
select CUSTOMER_ID, EMAIL, FIRST_NAME, LAST_NAME
from INACTIVE_CUSTOMERS) QRSLT
```

And, the view object adds the dynamic `WHERE` clause predicate at the end, so that the final query the database sees looks like the one shown in the following example.

```
SELECT * FROM(
select CUSTOMER_ID, EMAIL, FIRST_NAME, LAST_NAME
from CUSTOMERS
union all
select CUSTOMER_ID, EMAIL, FIRST_NAME, LAST_NAME
from INACTIVE_CUSTOMERS) QRSLT
WHERE email = :TheUserEmail
```

This query "wrapping" is necessary in general for custom SQL queries, because the original query could be arbitrarily complex, including SQL `UNION`, `INTERSECT`, `MINUS`, or other operators that combine multiple queries into a single result. In those cases, simply "gluing" the additional runtime `WHERE` clause onto the end of the query text could produce unexpected results. For example, the clause might apply only to the *last* of several `UNION`'ed statements. By nesting the original query verbatim into an inline view, the view object guarantees that your additional `WHERE` clause is correctly used to filter the results of the original query, regardless of how complex it is.

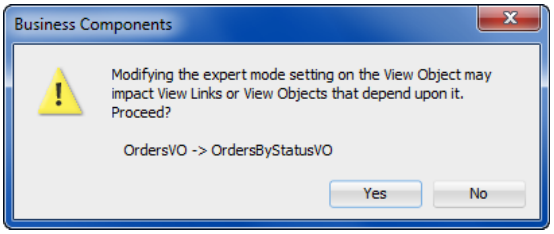
Inline View Wrapping at Runtime

Inline view wrapping of custom SQL view objects, limits a dynamically added `WHERE` clause to refer only to columns in the `SELECT` list of the original query. To avoid this limitation, when necessary you can disable the use of the inline view wrapping by calling `setNestedSelectForFullSql(false)`.

Custom SQL Affect on Dependent Objects

When you modify a view object query to be in custom SQL mode after you have already created the view links that involve that view object or after you created other view objects that extend the view object, JDeveloper will warn you with the alert shown in Figure 5-26. The alert reminds you that you should revisit these dependent components to ensure their SQL statements still reflect the correct query.

Figure 5-26 Proactive Reminder to Revisit Dependent Components



Business Components

Modifying the expert mode setting on the View Object may impact View Links or View Objects that depend upon it. Proceed?

OrdersVO -> OrdersByStatusVO

For example, if you were to modify the `OrdersVO` view object to use custom SQL mode, because the `OrdersByStatusVO` view object extends it, you need to revisit the extended component to ensure that its query still logically reflects an extension of the modified parent component.

Custom SQL Affect on View Object Query Auto Refresh

When you create a view object query statement in custom SQL mode and you wish to enable automatic query updates using Oracle Database change notification, you will need to test the query statement for compatibility with query result change notification. You can test the query in the Test Query dialog by clicking the **Test and Explain** button in the Query page of the view object overview editor. The Change Notification tab in the Test Query dialog will confirm whether the query can be registered for change notification. If the query statement is not compatible, you will need to revise the query as required by Oracle Database query result change notification registration requirements. For more information, see How to Automatically Refresh the View Object of the View Accessor.

SQL Types of Attributes in Custom SQL Mode

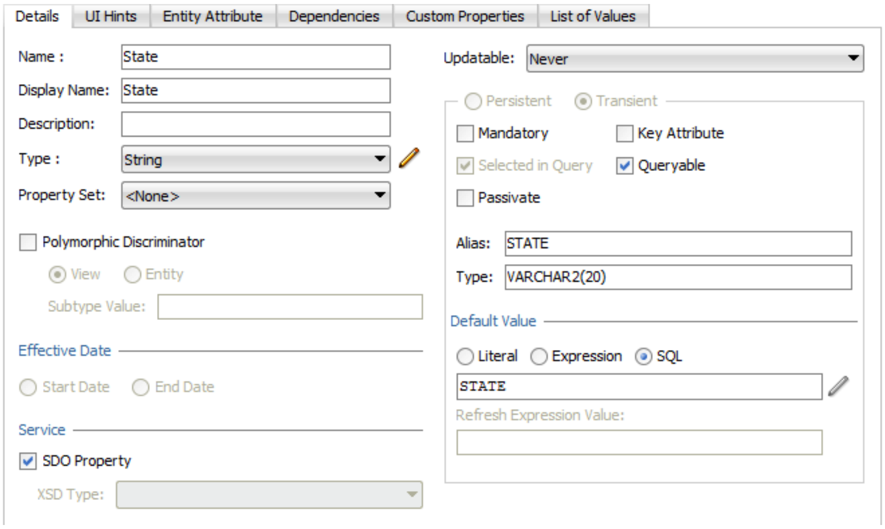
JDeveloper tries to infer the SQL type of the mapped column for view object attributes that you create in custom SQL mode automatically. The **Type** property of the attribute records the SQL type of the column, including the length information for `VARCHAR2` columns and the precision and scale information for `NUMBER` columns. However, for some SQL expressions the inferred value might default to `VARCHAR2(255)`.

You can update the **Type** value for this type of attribute to reflect the correct length if you know it. For example, `VARCHAR2(20)` which shows as the **Type** for the `State` attribute in Figure 5-27 means that it has a maximum length of 20 characters. For a `NUMBER` column, you would indicate a **Type** of `NUMBER(7,2)` for an attribute that you want to have a precision of 7 digits and a scale of 2 digits after the decimal.

Performance Tip:

Your SQL expression can control how long the describe from the database says the column is. Use the `SUBSTR()` function around the existing expression. For example, if you specify `SUBSTR(`*`yourexpression`*`, 1, 15)`, then the describe from the database will inform JDeveloper that the column has a maximum length of 15 characters.

Figure 5-27 Custom Attribute Settings in the Details Section of SQL-Derived Attributes



Name : State
Display Name: State
Description:
Type : String
Property Set: <None>
□ Polymorphic Discriminator
● View ○ Entity
Subtype Value:
Effective Date
○ Start Date ○ End Date
Service
☑ SDO Property
XSD Type:

Updatable: Never
○ Persistent ● Transient
□ Mandatory □ Key Attribute
☑ Selected in Query ☑ Queryable
□ Passivate
Alias: STATE
Type: VARCHAR2(20)
Default Value
○ Literal ○ Expression ● SQL
STATE
Refresh Expression Value:

In the case of entity-based view objects, you must edit the **Type** property in the Details section of the Attributes page of the overview editor that you display for the entity object, as described in How to Indicate Data Type, Length, Precision, and Scale.

Working with Named View Criteria

You can define ADF view criteria to filter data from view object rows. You can create specific usages of the view object by defining different named view criteria.

A view criteria lets you specify filter information for the rows of a view object collection. The view criteria object is a row set of one or more view criteria groups, whose attributes mirror those in the view object. The view criteria definition comprises query conditions that augment the `WHERE` clause of the target view object. However, unlike the `WHERE` clause defined by the view object query statement, which applies to all instances of the view object, the view criteria query condition is added to specific view object instances. This allows you to create specific usages of the target view object definition using query conditions that apply to the individual attributes of the target view object.

View criteria definitions support Query-by-Example operators and therefore allows the user to enter conditions such as "`OrderId > 304`", for example.

The Edit View Criteria dialog lets you create view criteria and save them as part of the view object's definition, where they appear as named view criteria. You use the View Criteria page of the overview editor to define view criteria for specific view objects. View criteria that you define at design time can participate in these scenarios where filtering results is desired at runtime:

- Supporting Query-by-Example search forms that allow the end user to supply values for attributes of the target view object.

  For example, the end user might input the value of a customer name and the date to filter the results in a web page that displays the rows of the `CustomerOrders` view object. The web page designer will see the named view criteria in the JDeveloper Data Controls panel as well as ADF Business Components data control, and, from them, easily create a search form. For more information about the utilizing the named view criteria in the Data Controls panel, see Creating Query Search Forms.

- Supporting row finder operations that the application may use to perform row look ups using any non-key attribute.

  For example, it is usually preferable to allow the end user to make row updates without the need to know the row key value (often an ID). In this case, a row finder that you apply to a view criteria supports locating a row using easily identifiable attribute values, such as by user name and user email address. See Working with Row Finders.

- Filtering the **list of values (LOV)** components that allow the end user may select from one attribute list (displayed in the UI as an LOV component).

  The web page designer will see the attributes of the view object in the JDeveloper Data Controls panel and, from them, easily create LOV controls. For information about utilizing LOV-enabled attributes in the Data Controls panel, see Creating a Selection List.

- Validating attribute values using a **view accessor** with a view criteria applied to filter the view accessor results.

For information about create view accessor validators, see How to Validate Against the Attribute Values Specified by a View Accessor.

- Creating the application module's data model from a single view object definition with a unique view criteria applied for each view instance.

  The single view object query modified by view criteria is useful with look up data that must be shared across the application. In this case, a base view object definition queries the lookup table in the database and the view criteria set the lookup table's `TYPE` column to define application-specific views. To define view instances in the data model using the view criteria you create for a base view object definition, see How to Define the WHERE Clause of the Lookup View Object Using View Criteria.

Additionally, view criteria have full API support, and it is therefore possible to create and apply view criteria to view objects programmatically.

How to Create Named View Criteria Declaratively

View criteria have a number of uses in addition to applying them to declarative queries at runtime. In all usages, the named view criteria definition consists of a set of attribute requirements that you specify to filter individual view object results. The features of the view criteria definition that you can use will depend on its intended usage.

To define view criteria for the view object you wish to filter, you open the view object in the overview editor and click the **Create** button in the **View Criteria** page. A dedicated editor (the Create View Criteria dialog) helps you to build a `WHERE` clause using attribute names instead of the target view object's corresponding SQL column names. You may define multiple named view criteria for each view object.

Before you work with the Create View Criteria dialog to create named view criteria, familiarize yourself with the usages described in Working with Named View Criteria. The chapter references provide additional details that will help you to anticipate using the appropriate features of the Create View Criteria dialog. For example, when you create a view criteria to specify the searchable attributes of a search form, the view criteria condition defines a simple list of attributes (a subset of the view object's attributes) to be presented to the user, but then the view criteria definition requires that you specify UI hints (model-level properties) to control the behavior of those attributes in the search form. The Create View Criteria dialog displays all the UI hints in a separate tabbed page that you select for the view criteria you are defining. Whereas, when your view criteria is intended to specify view instances in the data model, you can define arbitrarily complex query filter conditions, but you can ignore the UI hints features displayed by the Create View Criteria dialog.

Each view criteria definition consists of the following elements:

- An arbitrary number of view criteria groups or an arbitrary number of references to another named view criteria already defined for the current view object.
- An arbitrary number of view criteria items (grouped by view criteria groups) consisting of an attribute name, an attribute-appropriate operator, and an operand. Operands can be a literal value when the filter value is defined or a bind variable that can optionally utilize a scripting expression that includes dot notation access to attribute property values.

  Expressions are based on the Groovy scripting language, as described in Using Groovy Scripting Language with Business Components.

When you define a view criteria, you control the source of the filtered results. You can limit the results of the filtered view object to:

- Just the database query results.
- Just the in-memory results of the view object query. For example, new rows added by the user.
- Both the database and the in-memory results of the view object query.

Filtering on both database tables and the view object's in-memory results allows you to filter rows that were created in the transaction but not yet committed to the database.

View criteria expressions you construct in the Edit View Criteria dialog use logical conjunctions to specify how to join the selected criteria item or criteria group with the previous item or group in the expression:

- `AND` conjunctions specify that the query results meet both joined conditions. This is the default for each view criteria item you add.
- `OR` conjunctions specify that the query results meet either or both joined conditions. This is the default for view criteria groups.

Additionally, you may create nested view criteria when you want to filter rows in the current view object based on criteria applied to view-linked detail views. A nested view criteria group consists of an arbitrary number of nested view criteria items. You can use nested view criteria when you want to have more controls over the logical conjunctions among the various view criteria items. The nested criteria place restrictions on the rows that satisfy the criteria under the nested criteria's parent view criteria group. For example, you might want to query both a list of employees with (`Salary > 3000`) and belonging to (`DeptNo = 10` or `DeptNo = 20`). You can define a view criteria with the first group with one item for (`Salary > 3000`) and a nested view criteria with the second group with two items `DeptNo = 10` and `DeptNo =20`.

Before you begin:

It may be helpful to have an understanding of the ways you can use view criteria. The usage you intend will affect the best practices for creating named view criteria. For more information about the supported usages, see Working with Named View Criteria.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for View Objects.

You will need to complete these tasks:

- Create the desired view objects, as described in How to Create an Entity-Based View Object, and How to Create a Custom SQL Mode View Object.
- If the view criteria will use a bind variable in the operand, create the bind variable, as described in How to Add View Criteria Bind Variables to a View Object Definition.

To define a named view criteria:

1. In the Applications window, double-click the view object for which you want to create the named view criteria.
2. In the overview editor, click the **View Criteria** navigation tab and then click the **Create New View Criteria** button.

3. In the Create View Criteria dialog, enter the name of the view criteria to identify its usage in your application.

4. In the **Query Execution Mode** dropdown list, decide how you want the view criteria to filter the view object query results.

   You can limit the view criteria to filter the database table specified by the view object query, the in memory row set produced by the query, or both the database table and the in-memory results.

   Choosing **Both** may be appropriate for situations where you want to include rows created as a result of enforced view link consistency. In this case, in-memory filtering is performed after the initial fetch. For details about view link consistency, see Maintaining New Row Consistency Between View Objects Based on the Same Entity.

5. In the Criteria Definition tab, click one of these **Add** buttons to define the view criteria.

   - **Add Item** to add a single criteria item. The editor will add the item to the hierarchy beneath the current group or view criteria selection. By default each time you add an item, the editor will choose the next attribute to define the criteria item. You can change the attribute to any attribute that the view object query defines.

   - **Add Group** to add a new group that will compose criteria items that you intend to add to it. When you add a new group, the editor inserts the `OR` conjunction into the hierarchy. You can change the conjunction as desired.

   - **Add Criteria** to add a view criteria that you intend to define. This selection is an alternative to adding a named criteria that already exists in the view object definition. When you add a new view criteria, the editor inserts the `AND` conjunction into the hierarchy. You can change the conjunction as desired. Each time you add another view criteria, the editor nests the new view criteria beneath the current view criteria selection in the hierarchy. The root node of the hierarchy defines the named view criteria that you are currently editing.

     Search forms that the UI designer will create from view criteria are not able to use directly nested view criteria. For more information about defining nested expressions for use with search forms, see What You May Need to Know About Nested View Criteria Expressions.

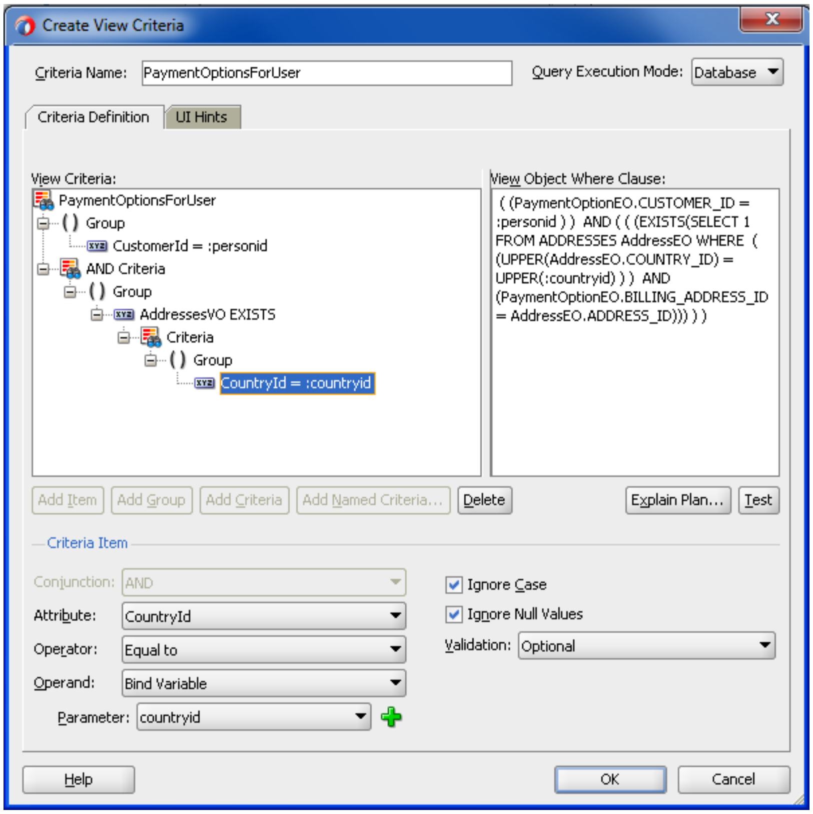
   - **Add Named Criteria** to add a view criteria that the view object defines. The named criteria must appear in the overview editor for the view object you are defining the view criteria.

6. Select a view criteria item node in the view criteria hierarchy and define the added node in the **Criteria Item** section.

   - Choose the desired **Attribute** for the criteria item. By default the editor adds the first one in the list.

     Optionally, you can add a nested view criteria inline when a view link exists for the current view object you are editing. The destination view object name will appear in the **Attribute** dropdown list. Selecting a view object lets you filter the view criteria based on view criteria items for the nested view criteria based on a view link relationship. For example, `EmpVO` is linked to the `PaymentOptionsVO` and a view criteria definition for `PaymentOptionsVO` will display the destination view object `AddressVO`. You could define the nested view criteria to filter payment options based on the `CountryId` attribute of the current customer, as specified by the `CustomerId` criteria item, as shown in Figure 5-28.

Figure 5-28   Edit View Criteria Dialog with Nested View Criteria Specified



Criteria Name: PaymentOptionsForUser

Query Execution Mode: Database

View Criteria:

```
PaymentOptionsForUser
  ( ) Group
      CustomerId = :personid
  AND Criteria
    ( ) Group
      AddressesVO EXISTS
        Criteria
          ( ) Group
              CountryId = :countryid
```

View Object Where Clause:

```
( (PaymentOptionEO.CUSTOMER_ID =
:personid ) )  AND ( ( (EXISTS(SELECT 1
FROM ADDRESSES AddressEO WHERE  (
(UPPER(AddressEO.COUNTRY_ID) =
UPPER(:countryid) ) )  AND
(PaymentOptionEO.BILLING_ADDRESS_ID
= AddressEO.ADDRESS_ID))) ) )
```

Criteria Item

Conjunction: AND

Attribute: CountryId

Operator: Equal to

Operand: Bind Variable

Parameter: countryid

☑ Ignore Case

☑ Ignore Null Values

Validation: Optional

- Choose the desired **Operator**.

  The list displays only the operators that are appropriate for the selected attribute or view object. In the case of a view object selection, the **exists** operator applies to a view criteria that you will define (or reference) as an operand. In the case of String and Date type attributes, the **Between** and **Not between** operators require you to supply two operand values to define the range. In the case of Date type attributes, you can select operators that test for a date or date range (with date values entered in the format YYYY-MM-DD). For example, for December 16th, 2010, enter 2010-12-16.

  JDeveloper does not support the `IN` operator. However, you can create a view criteria with the `IN` operator using the API, as described in How to Create View Criteria Programmatically.

7. Choose the desired **Operand** for the view criteria item selection.

   - Select **Literal** when you want to supply a value for the attribute or when you want to define a default value for a user-specified search field for a Query-by-Example search form. When the view criteria defines a query search form for the user interface, you may leave the **Value** field empty. In this case, the user will supply the value. You may also provide a value that will act as a search field default value that the user will be able to override. The value you supply in the **Value** field can include wildcard characters `*` or `%`.

   - Select **Bind Variable** when you want the value to be determined at runtime using a bind variable. If the variable was already defined for the view criteria, select it from the **Parameter** dropdown list. Otherwise, click **New** to display the New Variable dialog that lets you name a new bind variable on the view criteria.

When you select an existing bind variable from the dropdown list for use by the view criteria, you will be able to select bind variables that have been defined for the view object query or for the view criteria. The difference between these two sets of bind variables is that query bind variables (ones that you define in the Query page of the overview editor) are typically used within the view object query statement and therefore support different validation use cases than view criteria-defined bind variables. For view criteria, typically you always select a bind variable that is defined in the View Criteria page of the overview editor.

For further discussion about view criteria use cases for bind variables and literals, see What You May Need to Know About Bind Variables in View Criteria.

8. For each item, group, or nested view criteria that you define, optionally change the default conjunction to specify how the selection should be joined.
   - **AND** conjunction specifies that the query results meet both joined conditions. This is the default for each view criteria item or view nested view criteria that you add.
   - **OR** conjunction specifies that the query results meet either or both joined conditions. This is the default for view criteria groups.
9. Verify that the view criteria definition is valid by doing one of the following:
   - Click **Explain Plan** to visually inspect the view criteria's generated `WHERE` clause.
   - Click **Test** to allow JDeveloper to verify that the `WHERE` clause is valid.
10. To disable case-sensitive filtering of the attribute based on the case of the runtime-supplied value, leave **Ignore Case** selected.

    The criteria item can be a literal value that you define or a runtime parameter that the end user supplies. This option is supported for attributes of type String only. The default disables case sensitive searches.
11. In the **Validation** dropdown list, decide whether the view criteria item is a required or an optional part of the attribute value comparison in the generated `WHERE` clause.
    - **Selectively Required** means that the `WHERE` clause will ignore the view criteria item at runtime if no value is supplied and there exists at least one criteria item in the same view criteria group that has a criteria value. Otherwise, an exception is thrown.
    - **Optional** means the view criteria (or search field) is added to the `WHERE` clause only if the value is non-`NULL`. The default **Optional** for each new view criteria item means no exception will be generated for null values.
    - **Required** means that the `WHERE` clause will fail to execute and an exception will be thrown when no value is supplied for the criteria item.
12. If the view criteria uses a bind variable as the operand, decide whether the `IS NULL` condition is the generated in the `WHERE` clause. This field is enabled only if you have selected **Optional** for the validation of the bind variable.
    - Leave **Ignore Null Values** selected (default) when you want to permit the view criteria to return a result even when the bind variable value is not supplied

at runtime. For example, suppose you define a view criteria to allow users to display a cascading list of countries and states (or provinces) through a bind variable that takes the `countryID` as the child list's controlling attribute. In this case, the default behavior for the view criteria execution returns the list of *all* states if the user makes no selection in the parent LOV (an empty `countryId` field). The generated `WHERE` clause would look similar to `(((CountryEO.COUNTRY_ID =:bvCountryId) OR (:bvCountryId IS NULL)))`, where the test for a null value guarantees that the child list displays a result even when the bind variable is not set. When validation is set to **Required** or **Selectively Required**, the view criteria expects to receive a value and thus this option to ignore null values is disabled.

- Deselect **Ignore Null Values** when you expect the view criteria to return a null result when the bind variable value is not supplied at runtime. In the example of the cascading lists, the view criteria execution returns *no* states if the user makes no selection with an empty `countryID` field. In this case, the generated `WHERE` clause would look similar to `((CountryEO.COUNTRY_ID=:bvCountryId))`, where the test for null is not performed, which means the query is expected to function correctly with a null value bind variable. Note that the combination of optional validation with null values is only possible for bind variables that are required. Required bind variables are those defined in the Query page of the overview editor and used in the query `WHERE` clause.

  Note that the validation settings **Required** or **Selectively Required** also remove the null value condition but can be used in combination with a deselected **Ignore Null Values** setting to support a different use case. For more details about the interaction of these settings, see What You May Need to Know About Bind Variables in View Criteria.

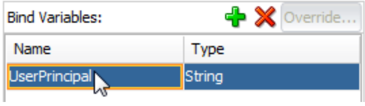
13. Click **OK**.
14. If any of the view criteria conditions you defined reference a bind variable, then in the View Criteria page, select the bind variable and then in the **Details** section, click the **Value** tab and optionally specify a default value for the bind variable:
    - When you want the value to be determined at runtime using an expression, enter a Groovy scripting language expression, select the **Expression** value type and enter the expression in the **Value** field.
    - When you want to define a default value, select the **Literal** value type and enter the literal value in the **Value** field.
15. Leave **Updatable** selected when you want the bind variable value to be defined through the user interface.

    The **Updatable** checkbox controls whether the end user will be allowed to change the bind variable value through the user interface. If a bind variable is not updatable (it may not be changed either programmatically or by the end user), deselect **Updatable**.
16. In the View Criteria page, click the **UI Hints** tab and specify hints like **Label**, **Format Type**, **Format** mask, and others.

    The view layer will use bind variable UI hints when you build user interfaces like search pages that allow the user to enter values for the named bind variables. Note that formats are only supported for bind variables defined by the `Date` type or any numeric data type.
17. If you created bind variables that you do not intend to reference in the view criteria definition, select the bind variable and click the **Delete** button.

Confirm that your bind variable has been named in the view criteria by moving your cursor over the bind variable name field, as shown in Figure 5-29. JDeveloper identifies unreferenced bind variables by displaying the name field with an orange border.

Figure 5-29 Orange Border Reminds to Reference the Bind Variable



What Happens When You Create a Named View Criteria

The Create View Criteria dialog in JDeveloper lets you easily create view criteria and save them as named definitions. These named view criteria definitions add metadata to the XML document file that represents the target view object's declarative settings. Once defined, named view criteria appear by name in the View Criteria page of the overview editor for the view object.

To view the view criteria, expand the desired view object in the Applications window, select the XML file under the expanded view object, open the Structure window, and expand the View Criteria node. Each view criteria definition for a view object contains one or more `<ViewCriteriaRow>` elements corresponding to the number of groups that you define in the Create View Criteria dialog. The following example shows the `ProductsVO.xml` file with the `<ViewCriteria>` definition `FindByProductNameCriteria` and a single `<ViewCriteriaRow>` that defines a developer-defined search form for products using the bind variable `:bvProductName`. Any UI hints that you selected to customize the behavior of a developer-defined search from will appear in the `<ViewCriteria>` definition as attributes of the `<CustomProperties>` element. For details about specific UI hints for view criteria, see How to Set User Interface Hints on View Criteria to Support Search Forms.

```
<ViewObject
    xmlns="http://xmlns.oracle.com/bc4j"
    Name="ProductsVO"
    ... >
  <SQLQuery>
    ...
  </SQLQuery>
  ...
  <ViewCriteria
    Name="FindByProductNameCriteria"
    ViewObjectName="oracle.summit.model.views.ProductsVO"
    Conjunction="AND">
    <Properties>
      <CustomProperties>
        <Property
          Name="mode"
          Value="Basic"/>
        <Property
          Name="autoExecute"
          Value="false"/>
        <Property
          Name="showInList"
          Value="true"/>
        <Property
          Name="displayName"
          Value="Find Products By Name"/>
        <Property
          Name="displayOperators"
          Value="InAdvancedMode"/>
        <Property
          Name="allowConjunctionOverride"
          Value="true"/>
      </CustomProperties>
    </Properties>
    <ViewCriteriaRow
      Name="vcrow87">
      <ViewCriteriaItem
        Name="ProductName"
        ViewAttribute="ProductName"
        Operator="CONTAINS"
        Conjunction="AND"
        Value=":bvProductName"
        UpperColumns="1"
        IsBindVarValue="true"
        Required="Optional"/>
    </ViewCriteriaRow>
  </ViewCriteria>
  ...
</ViewObject>
```

Additionally, when you create view objects and specify them as instances in an application module, JDeveloper automatically creates a **data control** to encapsulate the collections (view instances) that the application module contains. JDeveloper then populates the Data Controls panel with these collections and any view criteria that you have defined, as shown in How View Objects Appear in the Data Controls Panel.

What You May Need to Know About Bind Variables in View Criteria

The view criteria filter that you define using a bind variable expects to obtain its value at runtime. This can be helpful in a variety of user interface scenarios. To support a particular use case, familiarize yourself with the combination of the **Validation** and **Ignore Null Values** settings shown in Table 5-1.

Table 5-1 Use Cases for Bind Variable Options in View Criteria

Validation	Ignore Null Values	Use Cases	Notes
`Optional`	`True` (Default)	Configure cascading **List of Values (LOV)** where the parent LOV value is optional. Generate an optional search field in a search form.	This combination generates the SQL query `(ProductName = :bind) OR (:bind IS NULL)`. When used for cascading LOVs, no selection in the parent LOV returns all rows in the child LOV. To ensure that all rows are not returned prior to executing a search, a view criteria item with a literal operand (not a bind variable) type is preferred. When a bind variable on a view criteria with optional validation usage is not resolved, the result will be all rows returned.

Table 5-1 (Cont.) Use Cases for Bind Variable Options in View Criteria

Validation	Ignore Null Values	Use Cases	Notes
`Optional`	`False` (must select a required bind variable that has been defined in a query `WHERE` clause to achieve this combination)	Configure cascading LOVs where the parent LOV value is required.	This combination generates the SQL query (`ProductName = :bind`). When used for cascading LOVs, no selection in the parent LOV returns no rows in the child LOV. Avoid this combination for search forms, because when the user leaves the search field blank the search will attempt to find rows where this field is explicitly NULL. A better way to achieve this is for the user to explicitly select the "IS NULL" operator in advanced search mode.
`Required`	`False` (default and cannot be modified for this combination)	Generate a required search field in a search form.	This combination generates the SQL query `ProductName = :bind`. Avoid this setting for cascading LOVs, because no selection in the parent LOV will cause a validation error. To ensure that a non-NULL search result exists prior to executing a search, a view criteria item with a literal operand (not a bind variable) type is preferred. When a bind variable on a view criteria with required validation usage is not resolved, the result will be no rows returned.

What You May Need to Know About Nested View Criteria Expressions

Search forms that the UI designer will create from view criteria are not able to work with all types of nested expressions. Specifically, search forms do not support expressions with directly nested view criteria. This type of nested expression defines one view criteria as a direct child of another view criteria. Query search forms do support nested expressions where you nest the view criteria as a child of a criteria item which is itself a child of a view criteria. For more information about using view criteria to create search forms, see Implicit and Named View Criteria.

How to Set User Interface Hints on View Criteria to Support Search Forms

Named view criteria that you create for view object collections can be used by the web page designer to create Query-by-Example search forms. Web page designers select your named view criteria from the JDeveloper Data Controls panel to create search forms for the Fusion web application. In the web page, the search form utilizes an ADF Faces query search component that will be bound initially to the named view criteria selected in the Data Controls panel. At runtime, the end user may select among all other named view criteria that appear in the Data Controls panel. Named view criteria that the end user can select in a search form are known as **developer-defined system searches**. The query component automatically displays these system searches in its **Saved Search** dropdown list. For more information about creating search forms and using the ADF query search component, see Creating Query Search Forms.

> **Note:**
>
> By default, any named view criteria you create in the Edit View Criteria dialog will appear in the Data Controls panel. As long as the **Show In List** option appears selected in the UI Hints page of the Edit View Criteria dialog, JDeveloper assumes that the named view criteria should be available as a developer-defined system search. When you want to create a named view criteria that you do not want the end user to see in search forms, deselect the **Show In List** option in the dialog. For example, you might create a named view criteria only for an LOV-enabled attribute and so you would need to deselect **Show In List**.

Because developer-defined system searches are created in the data model project, the UI Hints page of the Edit View Criteria dialog lets you specify the default properties for the query component's runtime usage of individual named view criteria. At runtime, the query component's behavior will conform to the selections you make for the following system search properties:

Search Region Mode: Select the mode that you want the query component to display the system search as. The **Basic** mode has all features of **Advanced** mode, except that it does not allow the end user to dynamically modify the displayed search criteria fields. The default is **Basic** mode for a view criteria you define in the Edit View Criteria dialog.

Query Automatically: Select when you want the query associated with the named view criteria to be executed and the results displayed in the web page. Any developer-defined system search with this option enabled will automatically be executed when the end user selects it from the query component's **Saved Search** list. Deselect when the web page designer prefers not to update the previously displayed results until the end user submits the search criteria values on the form. Additionally, when a search form is invoked from a task flow, the search form will appear empty when this option is deselected and populated when enabled. By default, this option is disabled for a view criteria you define in the Edit View Criteria dialog.

Show Operators: Determine how you want the query component to display the operator selection field for the view criteria items to the end user. Note that the operators displayed by the query component when the users changes the mode of the query component from advanced to basic may change depending upon how you have configured **Show Operators**. The query component will only preserve the user’s selection between modes when you set **Show Operators** to **Always** to allow the full list of operators in both modes. However, if you set **Show Operators** to **In Advanced Mode** to only show the full list of operators in advanced mode, then after the user makes their selections and switches to basic mode, the operators will default to the ones defined by the view criteria and for this reason the user’s selections may appear to change. Select **Never** when you want the view criteria to be executed using ONLY the operators it defines (where the full list of operators will not be exposed in either basic or advance modes). Note that although search forms display any bind variables used in the query `WHERE` clause as search criteria, the **Show Operators** setting does not support displaying the operators associated with these bind variable expressions.

Show Match All and Match Any: Select to allow the query component to display the **Match All** and **Match Any** radio selection buttons to the end user. When these buttons are present, the end user can use them to modify the search to return matches for all view criteria items or any one view criteria item. This is equivalent to enforcing `AND` (match all) or `OR` (match any) conjunctions between view criteria items. Deselect when you want the view criteria to be executed using the view criteria item conjunctions it defines. In this case, the query component will not display the radio selection buttons.

Rendered Mode: Select individual view criteria items from the view criteria tree component and choose whether you want the selected item to appear in the search form when the end user toggles the query component between basic mode and advanced mode. The default for every view criteria item is **All**. The default mode permits the query component to render an item in either basic or advanced mode. By changing the **Rendered Mode** setting for individual view criteria items, you can customize the search form's appearance at runtime. For example, you may want basic mode to display a simplified search form to the end user, reserving advanced mode for displaying a search form with the full set of view criteria items. In this case, you would select **Advanced** for the view criteria item that you do not want displayed in the query component's basic mode. In contrast, when you want the selected view criteria item to be rendered only in basic mode, select **Basic**. Set any item that you do not want the search form to render in either basic or advanced mode to **Never**.

> **Note:**
>
> When your view criteria includes an item that should not be exposed to the user, use the **Rendered Mode** setting **Never** to prevent it from appearing in the search form. For example, a view criteria may be created to search for products in the logged-in customer's cart; however, you may want to prevent the user from changing the customer ID to display another customer's cart contents. In this scenario, the view criteria item corresponding to the customer ID would be set to the current customer ID using a named bind variable. Although the bind variable definition might specify the variable as not required and not updatable, with the UI hint property **Display** set to **Hide**, only the **Rendered Mode** setting determines whether or not the search form displays the value.

Support Multiple Value Selection: Select when you want to allow the end user to make multiple selections for an individual criteria item that the query component displays. This option is only enabled when the view object attribute specified by the view criteria item has a List of Values (LOV) defined. Additionally, multiple selections will only be supported by the query component when the end user selects the operator **equal to** or **not equal to**. For example, if the criteria item names an attribute `CountryId` and this attribute derives its values from a list of country IDs accessed by the attribute's associated LOV, then selecting this option would allow the end user to submit the query with multiple country selections. At runtime, the query component will generate the appropriate query clause based on the end user's operator selection.

Display Width: Enter the character width to be used by the control that displays this criteria item in the query component. The value you enter will override the display width **control hint** defined for the criteria item's corresponding view object attribute. For example, in an edit form the attribute control hint may allow text of 1024 length, but in the search form you might want to limit the field for the criteria item to 20 character length.

Show In List: Select to ensure that the view criteria is defined as a developer-seeded query. Deselect when the named view criteria you are defining is not to be used by the query search component to display a search form. Your selection determines whether the named view criteria will appear in the query search component's **Saved Search** dropdown list of available system searches. By default, this option is enabled for a view criteria you define in the Edit View Criteria dialog.

Display Name: Enter the name of the system search that you want to appear in the query component's **Saved Search** dropdown list or click the **...** button (to the right of the edit field) to select a message string from the resource bundle associated with the view object. The display name will be the name by which the end user identifies the system search. When you select a message string from the resource bundle, JDeveloper saves the string's corresponding message key in the view object definition file. At runtime, the UI locates the string to display based on the end user's locale setting and the message key in the localized resource bundle. When you do not specify a display name, the view criteria name displayed in the Edit View Criteria dialog will be used by default.

To create a system search for use by the ADF query search component, you select **Show In List** in the UI Hints page of the Edit View Criteria dialog. You deselect **Show In List** when you do not want the end user to see the view criteria in their search form.

Before you begin:

It may be helpful to have an understanding of view criteria. See Working with Named View Criteria.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. See Additional Functionality for View Objects.

You will need to complete these tasks:

- Create the desired view objects, as described in How to Create an Entity-Based View Object, and How to Create a Custom SQL Mode View Object.
- Create the view criteria, as described in How to Create Named View Criteria Declaratively.

For information abou defining UI hints for view object attributes, see How to Add Attribute-Specific UI Hints.

To customize a named view criteria for the user interface:

1. In the Applications window, double-click the view object that defines the named view criteria you want to use as a system search.
2. In the overview editor, click the **View Criteria** navigation tab and select the named view criteria that you want to allow in system searches and then click the **Edit Selected View Criteria** button.
3. In Edit View Criteria dialog, click the **UI Hints** tab and ensure that **Show In List** is selected.

   This selection determines whether or not the query component will display the system search in its **Saved Search** dropdown list.
4. Enter a user-friendly display name for the system search to be added to the query component's Saved Search dropdown list.

   When left empty, the view criteria name displayed in the Edit View Criteria dialog will be used by the query component.

5. Optionally, enable **Query Automatically** when you want the query component to automatically display the search results whenever the end user selects the system search from the **Saved Search** dropdown list.

   By default, no search results will be displayed.

6. Optionally, apply **Criteria Item UI Hints** to customize whether the query component renders individual criteria items when the end user toggles the search from between basic and advanced mode.

   By default, all view criteria items defined by the system search will be displayed in either mode.

   If a rendered criteria item is of type `Date`, you must also define UI hints for the corresponding view object attribute. Set the view object attribute's Format Type hint to **Simple Date** and set the Format Mask to an appropriate value. This will allow the search form to accept date values.

7. Click **OK**.

How to Test View Criteria Using the Oracle ADF Model Tester

To test the view criteria you added to a view object, use the Oracle ADF Model Tester, which is accessible from the Applications window.

The Oracle ADF Model Tester, for any view object instance that you browse, lets you bring up the View Criteria dialog, as shown in Figure 5-30. The dialog allows you to create a view criteria comprising one or more view criteria groups.

To apply criteria attributes from a single view criteria group, click the **Specify View Criteria** toolbar button in the browser and enter Query-by-Example criteria in the desired fields, then click **Find**.

To test view criteria using the Oracle ADF Model Tester:

1. In the Applications window, expand the project containing the desired application module and view objects.

2. Right-click the application module and choose **Run**.

3. In the Oracle ADF Model Tester, double-click the view instance you want to filter, and then right-click the view instance and choose **Find**.

   Alternatively, after you double-click a view instance, you can click the **Specify View Criteria** toolbar button to test the view criteria.

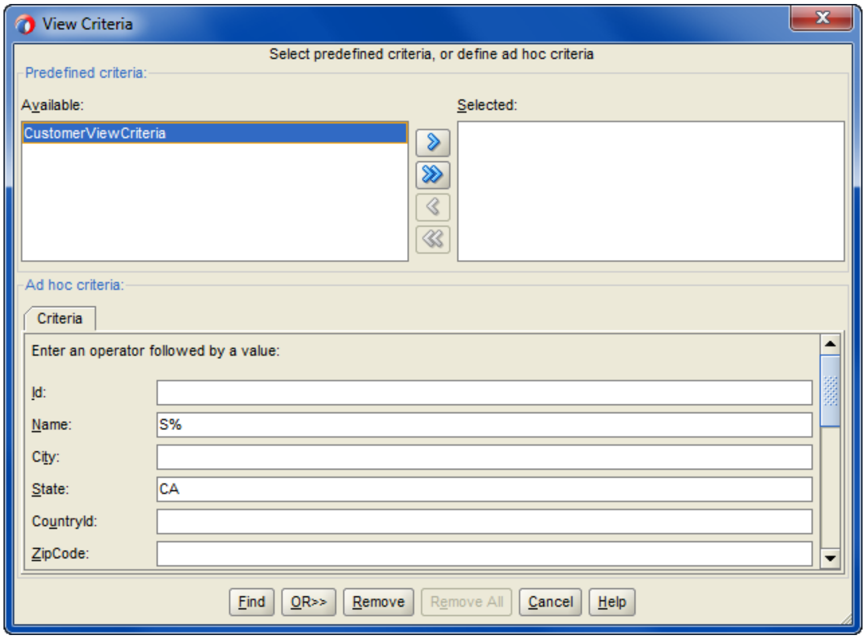
4. In the View Criteria dialog, perform one of the following tasks:

   - To test a view criteria that you added to the view object in your project, select from the list and click **Find**. Any additional criteria that you enter in the ad hoc Criteria panel will be added to the filter.

   - To test ad hoc criteria attributes from a single view criteria group, enter the desired values for the view criteria and click **Find**. For example, Figure 5-30 shows the filter to return all customers whose names begins with "S" and who live in CA.

   - To test additional ad hoc view criteria groups, click the **OR** tab and use the additional tabs that appear to switch between pages, each representing a distinct view criteria group. When you click **Find**, the Oracle ADF Model Tester will create and apply the view criteria to filter the result.

Figure 5-30 View Criteria Dialog in Oracle ADF Model Tester



Select predefined criteria, or define ad hoc criteria

Available: CustomerViewCriteria

Enter an operator followed by a value:

Id:

Name: S%

City:

State: CA

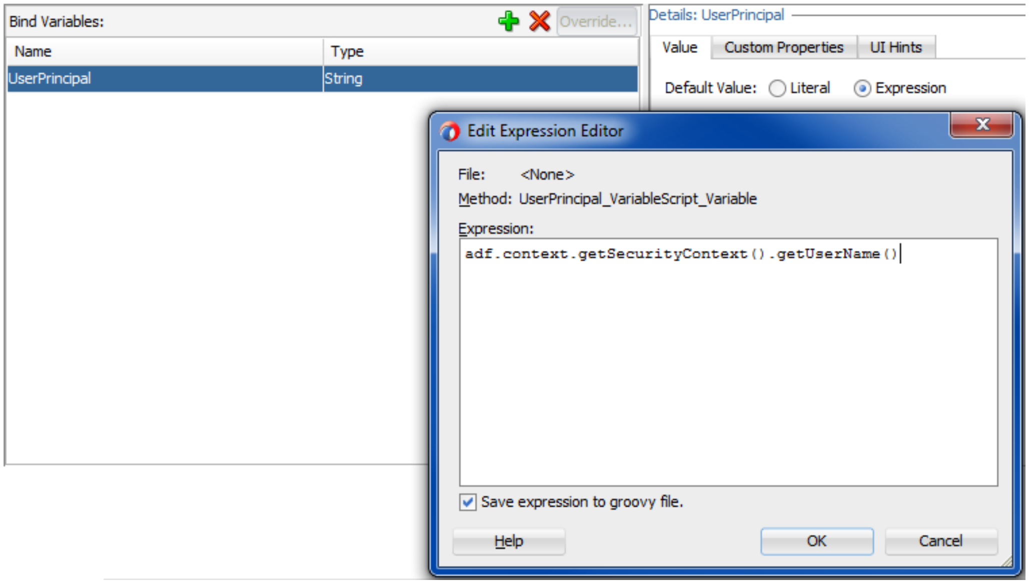
CountryId:

ZipCode:

How to Use View Criteria to Filter a View Object for the Current User

You can use the Groovy expression `adf.context.getSecurityContext().getUserName()` to set the default value for the named bind variable that you use to provide the current user in a view instance filter. Specifically, you can use the bind variable in a named view criteria that you define to filter a view object instance in the data model for the project. For example, the named bind variable `UserPrincipal` is defined in the View Criteria page of the overview editor, as shown in Figure 5-31. Note that you can write Groovy expressions on business components to access the security context using expressions similar to `adf.context.getSecurityContext()`*`methodName()`*. For example, from the security context you can access information about the current user like their user name or whether they belong to a particular role. For more information about the ADF security context, see Getting Information from the ADF Security Context.

Figure 5-31 Groovy Expression Used to Set UserPrincipal Bind Variable



```
adf.context.getSecurityContext().getUserName()
```

The `CustomerVO` view object also defines the `AuthenticatedUserByPrincipalCriteria` view criteria. This view criteria defines a filter for the `PrincipalName` attribute of the `PersonsVO` with the bind variable `userPrincipal` providing the value. In this example, the bind variable `userPrincipal` is defined with **Updatable** enabled. This ensures that the view criteria is able to set the value obtained at runtime from the ADF security context. Since the bind variable is not used in the SQL `WHERE` clause for the `PersonsVO` view object, the **Required** field is unselected. This ensures that the value is optional and that no runtime exception will be thrown if the bind variable is not resolved.

Then in the data model, where the `CustomerVO` specifies the view definition for the usage `AuthenticatedUser`, the view criteria `AuthenticatedUserByPrincipalCriteria` with the named bind variable is defined as the view usage's runtime filter. For details about creating view instances for your project's data model, see Customizing a View Object Instance that You Add to an Application Module.

How to Create View Criteria Programmatically

The following example shows the `main()` method finds the `CustomerList` view object instance to be filtered, creates a view criteria for the attributes of this view object, and applies the view criteria to the view object.

To create a view criteria programmatically, follow these basic steps (as illustrated in the following example from the `oracle.summit.model.viewobjects.ProgrammaticVOCriteria.java` example in the `SummitADF_Examples` workspace):

1. Find the view object instance to be filtered.
2. Create a view criteria row set for the view object.
3. Use the view criteria to create one or more empty view criteria groups
4. Set attribute values to filter on the appropriate view criteria groups.

You use the `ensureCriteriaItem()`, `setOperator()`, and `setSearchValue()` methods on the view criteria groups to set attribute name, comparison operator, and value to filter on individually.

5. Add the view criteria groups to the view criteria row set.Apply the view criteria to the view object.
6. Execute the query.

The last step to execute the query is important, since a newly applied view criteria is applied to the view object's SQL query only at its next execution.

```
public class ProgrammaticVOCriteria {
    public static void main(String[] args) {

      // get the application module
      String amDef = "oracle.summit.model.viewobjects.AppModule";
      String config = "AppModuleLocal";
      ApplicationModule am =
            Configuration.createRootApplicationModule(amDef, config);

      // 1. Find the View Object you want to filter
      ViewObject customerList = am.findViewObject("SCustomerView1");

      // Work with your appmodule and view object here

      // iterate through all the rows first, just to see to full VO results
      customerList.executeQuery();
      while (customerList.hasNext()) {
          Row customer = customerList.next();
          System.out.println("Customer: " + customer.getAttribute("Name") + "
                State: " + customer.getAttribute("State"));

      System.out.println("********* Set View Criteria and requery *********");

      // 2. Create a view criteria row set for this view object
      ViewCriteria vc = customerList.createViewCriteria();

      // 3. Use view criteria row set to create at least one view criteria group
      ViewCriteriaRow vcr1 = vc.createViewCriteriaRow();

      // 4. Set attribute values to filter on
      ViewCriteriaItem vci = vcr1.ensureCriteriaItem("State");
      vci.setOperator("=");
      vci.setSearchValue("TX");

      // 5. Add the view criteria group to the view criteria row set
      vc.add(vcr1);

      // 6. Apply the view criteria to the view object
      customerList.applyViewCriteria(vc, true);

      // 7. Execute the query
      customerList.executeQuery();

      // Iterate throught the rows to see the new results
      while (customerList.hasNext()) {
          Row customer = customerList.next();
          System.out.println("Customer: " + customer.getAttribute("Name") + "
                  State: " + customer.getAttribute("State"));
      }
```

```
        System.out.println("********* Set View Criteria and requery *********");

        // set the attribute to a different value
           vci.setSearchValue("CA")

        // Execute the query
          customerList.executeQuery();

        // iterate throught the rows to see the new results
        while (customerList.hasNext()) {
            Row customer = customerList.next();
            System.out.println("Customer: " + customer.getAttribute("Name") + "
                        State: " + customer.getAttribute("State"));
        }


        Configuration.releaseRootApplicationModule(am, true);
      }
}
```

Running the `ProgrammaticVOCriteria.java` test client produces one set of records each time the query is executed. The first query is with no criteria applied (shows all records), the second query is with one criteria set (shows records for `TX` only), and the third query is with the criteria changed (shows records for `CA` only).

```
Customer: Great Gear   State: TX
Customer: Acme Outfitters   State: TX
Customer: Athena's Closet   State: TX
Customer: Big John's Sports Emporium   State: CA
Customer: Perfect Purchase   State: CA
Customer: Father Gym's   State: CA
...
********* Set View Criteria and requery *********
Customer: Great Gear   State: TX
Customer: Acme Outfitters   State: TX
Customer: Athena's Closet   State: TX
...
********* Set View Criteria and requery *********
Customer: Big John's Sports Emporium   State: CA
Customer: Perfect Purchase   State: CA
Customer: Father Gym's   State: CA
...
```

View criteria can also be created using `buildCriteria()`.

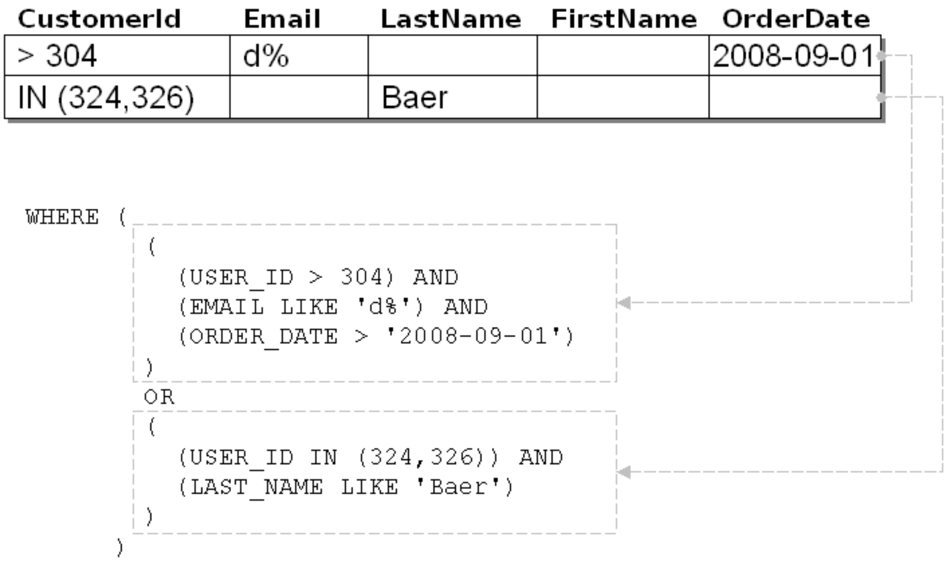
```
 public class ProgrammaticVOCriteria {
    public static void main(String[] args) {
        ProgrammaticVOCriteria pc = new ProgrammaticVOCriteria();
        // get the application module
             String amDef = "model.AppModule";
             String config = "AppModuleLocal";
             ApplicationModule am =
                   Configuration.createRootApplicationModule(amDef, config);
             ViewObjectImpl deptvo = (ViewObjectImpl)
am.findViewObject("DeptView1");
           }
        pc.testExpr("DeptViewCriteria", deptvo, "Deptno in (20, 30, 40)");
        Configuration.releaseRootApplicationModule(am, true);
    }
    public void testExpr(String vcName, ViewObjectImpl vo, String expr){
        System.out.println("Expression = " + expr);
        ViewCriteria vc = vo.buildViewCriteria(vcName, expr);
        System.out.println("VC: " + vc);
        vo.applyViewCriteria(vc);
        vo.executeQuery();
        while(vo.hasNext()){
        Row r = vo.next();
        System.out.println("Dept : " + r.getAttribute("Deptno") + "/" +
r.getAttribute("Dname"));
        }
    }
}
```

What Happens at Runtime: How the View Criteria Is Applied to a View Object

When you apply a view criteria containing one or more view criteria groups to a view object, the next time it is executed it augments its SQL query with an additional WHERE clause predicate corresponding to the Query-by-Example criteria that you've populated in the view criteria groups. As shown in Figure 5-32, when you apply a view criteria containing multiple view criteria groups, the view object augments its design time WHERE clause by adding an additional runtime WHERE clause based on the non-null example criteria attributes in each view criteria group.

A corollary of the view criteria feature is that each time you apply a new view criteria (or remove an existing one), the text of the view object's SQL query is effectively changed. Changing the SQL query causes the database to reparse the statement the next time it is executed. You can eliminate the reparsing and improve the performance of a view criteria, as described in What You May Need to Know About Query-by-Example Criteria.

Figure 5-32 View Object Automatically Translates View Criteria Groups into Additional Runtime WHERE Filter



CustomerId	Email	LastName	FirstName	OrderDate
> 304	d%			2008-09-01
IN (324,326)		Baer		

```
WHERE (
        (
          (USER_ID > 304) AND
          (EMAIL LIKE 'd%') AND
          (ORDER_DATE > '2008-09-01')
        )
        OR
        (
          (USER_ID IN (324,326)) AND
          (LAST_NAME LIKE 'Baer')
        )
      )
```

What You May Need to Know About the View Criteria API

When you need to perform tasks that the Edit View Criteria dialog does not support, review the View Criteria API. For example, programmatically, you can alter compound search conditions using multiple view criteria groups, search for a row whose attribute value is NULL, search case insensitively, and clear view criteria in effect.

Referencing Attribute Names in View Criteria

The setWhereClause() method allows you to add a dynamic WHERE clause to a view object, as described in ViewObject Interface Methods for Working with the View Object's Default RowSet. You can also use setWhereClause() to pass a string that contains literal database column names like this:

```
vo.setWhereClause("LAST_NAME LIKE UPPER(:NameToFind)");
```

In contrast, when you use the view criteria mechanism, described in How to Create View Criteria Programmatically, you must reference the view object attribute name instead, like this:

```
    ViewCriteriaItem vc_item1 = vc_row1.ensureCriteriaItem("UserId");
    vc_item1.setOperator(">");
    vc_item1.setValue("304");
```

Note that method calls like vcr.setAttribute("UserId", "> 304") as documented in a past release must not be used to set the attribute values for the view criteria.

The view criteria groups are then translated by the view object into corresponding WHERE clause predicates that reference the corresponding column names. The programmatically set WHERE clause is AND-ed with the WHERE clauses of any view criteria that have been defined for the view object at design time.

Referencing Bind Variables in View Criteria

When you want to set the value of a view criteria item to a bind variable, use setIsBindVarValue(true), like this:

```
    ViewCriteriaItem vc_item1 = vc_row1.ensureCriteriaItem("UserId");
    vc_item1.setIsBindVarValue(true);
    vc_item1.setValue(":VariableName");
```

Searching for a Row Whose Attribute Is NULL

To search for a row containing NULL in a column, populate a corresponding view criteria group attribute with the value "IS NULL" or use ViewCriteriaItem.setOperator("ISBLANK").

Searching for Rows Using a Date Comparison

When you want to perform a comparison of date values in a view criteria item, use the following predefined operators:

- BEFORE
- AFTER

- ONORBEFORE
- ONORAFTER

For example, to search for rows that contain a DATE type attribute whose value is after a given date, you would use ViewCriteriaItem.setOperator("AFTER").

Do not use operators like <, >, <=, and >= because these operators perform string comparisons that yield inaccurate results for date values.

Searching for Rows Whose Attribute Value Matches a Value in a List

To search for all rows with a value in a column that matches any value in a list of values that you specify, populate a corresponding view criteria group attribute with the comma-separated list of values and use the IN operator. For example, to filter the list of persons by IDs that match 204 and 206, set:

```
ViewCriteriaItem vci = vcr.ensureCriteriaItem("CustomerId");
vci.setOperator("IN");
vci.setValue(0, 204);
vci.setValue(1, 206);
```

Searching Case-Insensitively

To search case-insensitively, call setUpperColumns(true) on the view criteria group to which you want the case-insensitivity to apply. This affects the WHERE clause predicate generated for String-valued attributes in the view object to use UPPER(COLUMN_NAME) instead of COLUMN_NAME in the predicate. Note that the value of the supplied view criteria group attributes for these String-valued attributes must be uppercase or the predicate won't match. In addition to the predicate, it also possible to use UPPER() on the value. For example, you can set UPPER(ename) = UPPER("scott").

Clearing View Criteria in Effect

To clear any view criteria in effect, you can call getViewCriteria() on a view object and then delete all the view criteria groups from it using the remove() method, passing the zero-based index of the criteria row you want to remove. If you don't plan to add back other view criteria groups, you can also clear the entire view criteria in effect by simply calling removeApplyViewCriteriaName("*namedViewCriteria*") on the view object with the name of the view criteria passed in.

Altering Compound Search Conditions Using Multiple View Criteria

When you add multiple view criteria, you can call the setConjunction() method on a view criteria to alter the conjunction used between the predicate corresponding to that view criteria and the one for the previous view criteria. The legal constants to pass as an argument are:

- ViewCriteriaComponent.VC_CONJ_AND
- ViewCriteriaComponent.VC_CONJ_NOT
- ViewCriteriaComponent.VC_CONJ_UNION
- ViewCriteriaComponent.VC_CONJ_OR (*default*)

The NOT value can be combined with AND or OR to create filter criteria like:

```
( PredicateForViewCriteria1 ) AND (NOT ( PredicateForViewCriteria2 ) )
```

or

```
( PredicateForViewCriteria1 ) OR (NOT ( PredicateForViewCriteria2 ) )
```

The syntax to achieve compound search conditions requires using Java's bitwise `OR` operator like this:

```
vc2.setConjunction(ViewCriteriaComponent.VC_CONJ_AND |
 ViewCriteriaComponent.VC_CONJ_NOT);
```

> **Performance Tip:**
>
> Use the `UNION` value instead of an `OR` clause when the `UNION` query can make use of indices. For example, if the view criteria searches for `sal > 2000 or job = 'CLERK'` this query may turn into a full table scan. Whereas if you specify the query as the union of two inner view criteria, and the database table has an index on `sal` and an index on `job`, then the query can take advantage of these indices and the query performance will be significantly better for a large data set.

The limitation for the `UNION` clause is that it must be defined over one view object. This means that the `SELECT` and the `FROM` list will be the same for inner queries of the `UNION` clause. To specify a `UNION` query, call `setConjunction()` on the outer view criteria like this:

```
vc.setConjunction(ViewCriteriaComponent.VC_CONJ_UNION);
```

The outer view criteria should contain inner queries whose results will be the union. For example, suppose you want to specify the union of these two view criteria:

- A view criteria named `MyEmpJob`, which searches for `Job = 'SALESMAN'`.
- A view criteria named `MyEmpSalary`, which searches for `Sal = 1500`.

To create the `UNION` query for these two view criteria, you would make the calls shown in the following example.

```
vcu = voEmp.createViewCriteria();
vcm = voEmp.getViewCriteriaManager();

vcu.setConjunction(ViewCriteria.VC_CONJ_UNION);
vcu.add(vcm.getViewCriteria("MyEmpJob"));
vcu.add(vcm.getViewCriteria("MyEmpSal"));

voEmp.applyViewCriteria(vcu);
```

When this view criteria is applied, it will return rows where `Job` is `SALESMAN` or `Sal` is greater than `1500`.

When you use a `UNION` view criteria, be sure that only one of the applied view criteria has the `UNION` conjunction. Other view criteria that you apply will be applied to each inner query of the `UNION` query.

What You May Need to Know About Query-by-Example Criteria

For performance reasons, you want to avoid setting a bind variable as the value of a view criteria item in these two cases:

- In the specialized case where the value of a view criteria item is defined as selectively required and the value changes from non-`NULL` to `NULL`.

  In this case, the SQL statement for the view criteria will be regenerated each time the value changes from non-`NULL` to `NULL`.
- In the case where the value of the view criteria item is optional and that item references an attribute for an indexed column.

  In the case of optional view criteria items, an additional SQL clause `OR (:Variable IS NULL)` is generated, and the clause does not support using column indices.

In either of these cases, you will get better performance by using a view object whose `WHERE` clause contains the named bind variables, as described in How to Add WHERE Clause Bind Variables to a View Object Definition.

Working with Bind Variables

You can define bind variables to supply runtime attribute values to the ADF view object or view criteria.

Bind variables provide you with the means to supply attribute values at runtime to the view object or view criteria. All bind variables are defined at the level of the view object and used in one of the following ways:

- You can select the bind variable from a selection list to define the attribute value for a view criteria in the Edit View Criteria dialog you open on the view object. In this case, the bind variables allow you to change the values for attributes you will use to filter the view object row set. For more information about filtering view object row sets, see Working with Named View Criteria.

  If the view criteria is to be used in a UI developer-defined search form, you have the option of making the bind variable updatable by the end user. With this updatable option, end users will be expected to enter the value in a search form corresponding to the view object query.
- You can type the bind variable directly into the `WHERE` clause of your view object's query to include values that might change from execution to execution. In this case, bind variables serve as placeholders in the SQL string whose value you can easily change at runtime without altering the text of the SQL string itself. Since the query doesn't change, the database can efficiently reuse the same parsed representation of the query across multiple executions, which leads to higher runtime performance of your application.

In contrast, to query `WHERE` clause bind variables that are needed anytime the view object is executed, bind variables that you define for view criteria are needed only when the view criteria is applied.

> **Note:**
>
> Oracle recommends that your application use view criteria to filter view objects rather than hardcoding the filter in the query `WHERE` clause. When you use view criteria you can change the values of the search criteria without changing the text of the view object's SQL statement each time those values change.

You can define a default value for the bind variable or write scripting expressions for the bind variable that includes dot notation access to attribute property values. Expressions are based on the Groovy scripting language, as described in Using Groovy Scripting Language With Business Components.

How to Add View Criteria Bind Variables to a View Object Definition

Bind variable usages that you specify in a view criteria also require that you add the bind variable to the view object definition. If you add a bind variable usage to the view criteria definition and the view object does not have a bind variable definition, a runtime exception will result when the view criteria is applied. To create the bind variable definition, you use the View Criteria page in the overview editor for the view object. When you define the variable in the editor, the bind variable will receive a value at runtime, which when left undefined will be null. The default value of null may be suitable in some scenarios but not in others. When a non-null value is required, you can specify a default value for the bind variable definition, or you may let the end user supply the value at runtime (for example, through search criteria in the case of query search forms).

To add a view criteria bind variable to a view object, use the View Criteria page of the overview editor for the view object.

Before you begin:

It may be helpful to have an understanding of the support for bind variables at the level of view objects. For more information, see Working with Bind Variables.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for View Objects.

You will need to complete this task:

> Create the desired view objects, as described in How to Create an Entity-Based View Object, and How to Create a Custom SQL Mode View Object.

To define a named bind variable for the view criteria:

1. In the Applications window, double-click the view object.
2. In the overview editor, click the **View Criteria** navigation tab.
3. In the Query page, in the **Bind Variables** section, click the **Create New Bind Variable** button.
4. In the New Variable dialog, enter the name of bind variable and select the data type. Click **OK**.

Because the bind variables share the same namespace as view object attributes, specify names that don't conflict with existing view object attribute names. As with view objects attributes, by convention bind variable names are created with an initial capital letter, but you can rename it as desired.

5. Optionally, in the View Criteria page, in the **Details** section, click the **Value** tab and specify a default value for the bind variable:

   - When you want the value to be determined at runtime using an expression, enter a Groovy scripting language expression, select the **Expression** value type and enter the expression in the **Value** field. For example, you might want to define a bind variable to filter view instances based on the current user, as described in How to Use View Criteria to Filter a View Object for the Current User.

   - When you want to define a default value, select the **Literal** value type and enter the literal value in the **Value** field.

   - If you do not supply a default value for your named bind variable, it defaults to `NULL` at runtime and the view criteria will return no rows.

6. Leave **Updatable** selected when you want the bind variable value to be defined through the user interface.

   The **Updatable** checkbox controls whether the end user will be allowed to change the bind variable value through the user interface. If a bind variable is not updatable (it may not be changed either programmatically or by the end user), deselect **Updatable**.

7. In the View Criteria page, click the **UI Hints** tab and specify hints like **Label**, **Format Type**, **Format** mask, and others.

   The view layer will use bind variable UI hints when you build user interfaces like search forms that allow the user to enter values for the named bind variables. Note that formats are only supported for bind variables defined by the `Date` type or any numeric data type.

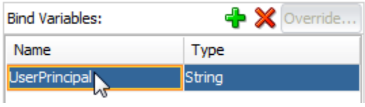
8. Reference the bind variables you defined in the view criteria of the view object.

   You must either reference the bind variable at runtime or use it in the view object query definition. Unreferenced bind variables that you configure in the overview editor will result in a runtime error, as described in Errors Related to the Names of Bind Variables.

9. If you created bind variables that you do not intend to name in your view criteria or reference programmatically, select the bind variable and click the **Delete** button.

   Confirm that your bind variable has been named in the view criteria by moving your cursor over the bind variable name field, as shown in Figure 5-34. JDeveloper identifies unreferenced bind variables by displaying the name field with an orange border.

Figure 5-33 Orange Border Reminds to Reference the Bind Variable



Name	Type
UserPrincipal	String

How to Add WHERE Clause Bind Variables to a View Object Definition

Bind variable usages that you specify in a query WHERE clause also require that you add the bind variable to the view object definition. If you add a bind variable usage to the query statement and the view object does not have a bind variable definition, a runtime exception will result. To create the bind variable definition, you use the Query page in the overview editor for the view object. When you define the variable in the editor, the bind variable will receive a value at runtime, which when left undefined will be null. The default value of null may be suitable in some scenarios but not in others. When a non-null value is required, you can specify a default value for the bind variable definition, or you set the value at runtime, as described in How to Set Existing WHERE Clause Bind Variable Values at Runtime.

To add a WHERE clause bind variable to a view object, use the Query page of the overview editor for the view object. You can define as many bind variables as you need.

> **Caution:**
>
> Unreferenced bind variables that you configure in the overview editor will result in a runtime error, as described in Errors Related to the Names of Bind Variables. Delete all bind variables definitions from the overview editor unless you name it in the view object definition or reference it programmatically.

Before you begin:

It may be helpful to have an understanding of the support for bind variables at the level of view objects. For more information, see Working with Bind Variables.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for View Objects.

You will need to complete this task:

Create the desired view objects, as described in How to Create an Entity-Based View Object, and How to Create a Custom SQL Mode View Object.

To define a named bind variable for the WHERE clause:

1. In the Applications window, double-click the view object.
2. In the overview editor, click the **Query** navigation tab.
3. In the Query page, in the **Bind Variables** section, click the **Create New Bind Variable** button.
4. In the New Variable dialog, enter the name of bind variable and select the data type. Click **OK**.

   Because the bind variables share the same namespace as view object attributes, specify names that don't conflict with existing view object attribute names. As with view objects attributes, by convention bind variable names are created with an initial capital letter, but you can rename it as desired.

5. Optionally, in the Query page, in the **Details** section, click the **Value** tab and specify a default value for the bind variable:

   - When you want the value to be determined at runtime using an expression, enter a Groovy scripting language expression, select the **Expression** value type and enter the expression in the **Value** field.
   - When you want to define a default value, select the **Literal** value type and enter the literal value in the **Value** field.
   - If you do not supply a default value for your named bind variable, it defaults to `NULL` at runtime and the query will return no rows. If this makes sense for your application, you might want to leverage SQL functions to handle this situation, as described in Default Value of NULL for Bind Variables.

6. Leave **Updatable** selected when you want the bind variable value to be defined through the user interface.

   The **Updatable** checkbox controls whether the end user will be allowed to change the bind variable value through the user interface. If a bind variable is not updatable (it may not be changed programmatically or by the end user), deselect **Updatable**.

7. In the Query page, click the **UI Hints** tab and specify hints like **Label**, **Format Type**, **Format** mask, and others.

   The view layer will use bind variable UI hints when you build user interfaces like search pages that allow the user to enter values for the named bind variables. Note that formats are only supported for bind variables defined by the `Date` type or any numeric data type.

8. Reference the bind variables you defined in the SQL statement of the view object.

   You must either reference the bind variable at runtime or use it in the view object query definition. Unreferenced bind variables that you configure in the overview editor will result in a runtime error, as described in Errors Related to the Names of Bind Variables.

   While SQL syntax allows bind variables to appear both in the `SELECT` list and in the `WHERE` clause, you'll typically use them in the latter context, as part of your `WHERE` clause. For example, the following sample shows the bind variables `LowUserId` and `HighUserId` introduced into a SQL statement created using the Query page in the overview editor for the view object.

```
select CUSTOMER_ID, EMAIL, FIRST_NAME, LAST_NAME
from CUSTOMERS
where (upper(FIRST_NAME) like upper(:TheName)||'%'
   or  upper(LAST_NAME)  like upper(:TheName)||'%')
  and CUSTOMER_ID between :LowUserId and :HighUserId
order by EMAIL
```

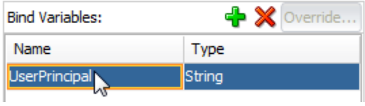
   Notice that you reference the bind variables in the SQL statement by prefixing their name with a colon like `:TheName` or `:LowUserId`. You can reference the bind variables in any order and repeat them as many times as needed within the SQL statement.

9. If you created bind variables that you do not intend to name in your view object SQL query or reference programmatically, select the bind variable and click the **Delete** button.

   Confirm that your bind variable has been named in the view object query by moving your cursor over the bind variable name field, as shown in Figure 5-34.

JDeveloper identifies unreferenced bind variables by displaying the name field with an orange border.

Figure 5-34 Orange Border Reminds to Reference the Bind Variable



Name	Type
UserPrincipal	String

What Happens When You Add Named Bind Variables

Once you've added one or more named bind variables to a view object, you gain the ability to easily see and set the values of these variables at runtime. Information about the name, type, and default value of each bind variable is saved in the view object's XML document file. If you have defined UI hints for the bind variables, this information is saved in the view object's component message bundle file along with other UI hints for the view object.

Note that the overview editor displays an orange warning box around the bind variable name for any bind variable that you define but do not use in the view object query. Be sure to delete bind variables that the query statement or view criteria definition does not reference. A runtime error will result when the view object query is executed and one or more bind variable definitions remain unreferenced, as described in Errors Related to the Names of Bind Variables.

How to Test Named Bind Variables

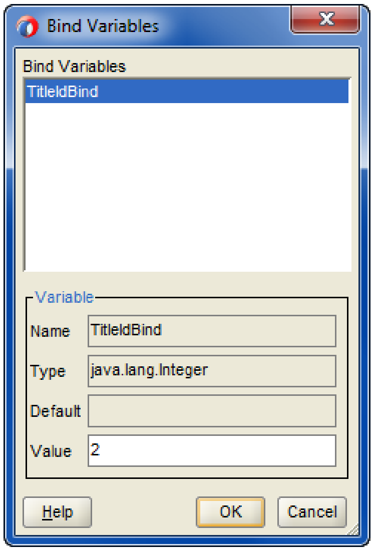
The Oracle ADF Model Tester allows you to interactively inspect and change the values of the named bind variables for any view object, which can really simplify experimenting with your application module's data model when named bind parameters are involved. For more information about editing the data model and running the Oracle ADF Model Tester, see Testing View Object Instances Using the Oracle ADF Model Tester.

The first time you execute a view object in the Oracle ADF Model Tester to display the results in the data view page, a Bind Variables dialog will appear, as shown in Figure 5-35.

The Bind Variables dialog lets you:

- View the name, as well as the default and current values, of the particular bind variable you select from the list
- Change the value of any bind variable by updating its corresponding **Value** field before clicking **OK** to set the bind variable values and execute the query
- Inspect and set the bind variables for the view object in the current data view page, using the **Edit Bind Parameters** button in the toolbar — whose icon looks like "`:id`"
- Verify UI hints are correctly set up by showing the label text hint in the **Bind Variables** list and by formatting the **Value** attribute using the respective format mask

Figure 5-35 Setting Bind Variables in the Oracle ADF Model Tester



TitleIdBind

Name: TitleIdBind
Type: java.lang.Integer
Default:
Value: 2

If you defined the bind variable in the Bind Variables dialog with the **Required** checkbox deselected, you will be able to test view criteria and supply the bind variable with values as needed. Otherwise, if you left the **Required** checkbox selected (the default), then you must supply a value for the bind variable in the Oracle ADF Model Tester. The Oracle ADF Model Tester will throw the same exception seen at runtime for any view object whose SQL statement use bind variables that do not resolve with a supplied value.

How to Add a WHERE Clause with Named Bind Variables at Runtime

Using the view object's `setWhereClause()` method, you can add an additional filtering clause at runtime. This runtime-added `WHERE` clause predicate does *not* replace the design-time generated predicate, but rather further narrows the query result by adding to the existing design time `WHERE` clause. Whenever the dynamically added clause refers to a value that might change during the life of the application, you should use a named bind variable instead of concatenating the literal value into the `WHERE` clause predicate.

For example, assume you want to further filter the `CustomerList` view object at runtime based on the value of the `CUSTOMER_TYPE_CODE` column in the table. Also assume that you plan to search sometimes for rows where `CUSTOMER_TYPE_CODE = 'CUST'` and other times for rows where `CUSTOMER_TYPE_CODE = 'SUPP'`. While it contains slightly fewer lines of code, the following example is not desirable because it changes the `WHERE` clause twice just to query two different *values* of the same `CUSTOMER_TYPE_CODE` column.

```
// Don't use literal strings if you plan to change the value!
vo.setWhereClause("customer_type_code = 'CUST'");
// execute the query and process the results, and then later...
vo.setWhereClause("customer_type_code = 'SUPP'");
```

Instead, you should add a `WHERE` clause predicate that references named bind variables that you define at runtime as shown in the following example.

```
vo.setWhereClause("customer_type_code = :TheCustomerType");
vo.defineNamedWhereClauseParam("TheCustomerType", null, null);
vo.setNamedWhereClauseParam("TheCustomerType","CUST");
// execute the query and process the results, and then later...
vo.setNamedWhereClauseParam("TheCustomerType","SUPP");
```

This allows the text of the SQL statement to stay the same, regardless of the value of CUSTOMER_TYPE_CODE you need to query on. When the query text stays the same across multiple executions, the database will return the results without having to reparse the query.

An updated test client class illustrating these techniques would look like what you see in the following example. In this case, the functionality that loops over the results several times has been refactored into a separate executeAndShowResults() method. The program first adds an additional WHERE clause of customer_id = :TheCustomerId and then later replaces it with a second clause of customer_type_code = :TheCustomerType.

```
package devguide.examples.readonlyvo.client;

import oracle.jbo.ApplicationModule;
import oracle.jbo.Row;
import oracle.jbo.ViewObject;
import oracle.jbo.client.Configuration;

public class TestClientBindVars {
  public static void main(String[] args) {
    String        amDef = "devguide.examples.readonlyvo.CustomerService";
    String        config = "CustomerServiceLocal";
    ApplicationModule am =
     Configuration.createRootApplicationModule(amDef,config);
    ViewObject vo = am.findViewObject("CustomerList");
    // Set the two design time named bind variables
    vo.setNamedWhereClauseParam("TheName","shelli%");
    vo.setNamedWhereClauseParam("HighUserId", 215);
    executeAndShowResults(vo);
    // Add an extra where clause with a new named bind variable
    vo.setWhereClause("customer_type_code = :TheCustomerId");
    vo.defineNamedWhereClauseParam("TheCustomerId", null, null);
    vo.setNamedWhereClauseParam("TheCustomerId",116);
    executeAndShowResults(vo);
    vo.removeNamedWhereClauseParam("TheCustomerId");
    // Add an extra where clause with a new named bind variable
    vo.setWhereClause("customer_type_code = :TheCustomerType");
    vo.defineNamedWhereClauseParam("TheCustomerType", null, null);
    vo.setNamedWhereClauseParam("TheCustomerType","SUPP");
    // Show results when :TheCustomerType = 'SUPP'
    executeAndShowResults(vo);
    vo.setNamedWhereClauseParam("TheCustomerType","CUST");
    // Show results when :TheCustomerType = 'CUST'
    executeAndShowResults(vo);
    Configuration.releaseRootApplicationModule(am,true);
  }
  private static void executeAndShowResults(ViewObject vo) {
    System.out.println("---");
    vo.executeQuery();
    while (vo.hasNext()) {
      Row curUser = vo.next();
      System.out.println(curUser.getAttribute("CustomerId")+" "+
                         curUser.getAttribute("ShortName"));
    }
  }
}
```

However, if you run this test program, you may actually get a runtime error like the one shown in the following example.

```
oracle.jbo.SQLStmtException: JBO-27122: SQL error during statement preparation.
Statement:
SELECT * FROM (select PERSON_ID, EMAIL, FIRST_NAME, LAST_NAME
from PERSONS
where (upper(FIRST_NAME) like upper(:TheName)||'%'
   or  upper(LAST_NAME)  like upper(:TheName)||'%')
  and PERSON_ID between :LowUserId and :HighUserId
order by EMAIL) QRSLT  WHERE (person_type_code = :TheCustomerType)
## Detail 0 ##
java.sql.SQLException: ORA-00904: "PERSON_TYPE": invalid identifier
```

The root cause, which appears after the `## Detail 0 ##` in the stack trace, is a SQL parsing error from the database reporting that `PERSON_TYPE_CODE` column does not exist even though the `PERSONS` table definitely has a `PERSON_TYPE_CODE` column. The problem occurs due to the mechanism that view objects use by default to apply additional runtime `WHERE` clauses on top of read-only queries. What Happens at Runtime: Dynamic Read-Only View Object WHERE Clause, explains a resolution for this issue.

How to Set Existing WHERE Clause Bind Variable Values at Runtime

To set named bind variables at runtime, use the `setNamedWhereClauseParam()` method on the `ViewObject` interface. In JDeveloper, you can choose **Refactor > Duplicate** in the main menu to create a new `TestClientBindVars` class based on the existing `TestClient.java` class as shown in How to Create a Command-Line Java Test Client. In the test client class, you can set the values of the bind variables using a few additional lines of code. For example, the `setNamedWhereClauseParam()` might take as arguments the bind variables `HighUserId` and `TheName` as shown in the following example.

```
// changed lines in TestClient class
ViewObject vo = am.findViewObject("CustomerList");
vo.setNamedWhereClauseParam("TheName","alex%");
vo.setNamedWhereClauseParam("HighUserId", 315);
vo.executeQuery();
// etc.
```

Running the test client class shows that your bind variables are filtering the data. For example, the resulting rows for the `setNamedWhereClauseParam()` method shown in example above may produce only two matches based on the name `alex` as shown in the following results.

```
303 ahunold
315 akhoo
```

Whenever a view object's query is executed, you can view the actual bind variable values in the runtime debug diagnostics, as the following results example illustrates.

```
[256] Bind params for ViewObject: CustomerList
[257] Binding param "LowUserId": 0
```

```
[258] Binding param "HighUserId": 315
[259] Binding param "TheName": alex%
```

This information that can be invaluable when debugging your applications. Notice that since the code did not set the value of the `LowUserId` bind variable, it took on the default value of `0` (zero) specified at design time. Also notice that the use of the `UPPER()` function in the `WHERE` clause and around the bind variable ensured that the match using the bind variable value for `TheName` was performed case-insensitively. The sample code set the bind variable value to "`alex%`" with a lowercase "`a`", and the results show that it matched `Alexander`.

What Happens at Runtime: Dynamic Read-Only View Object WHERE Clause

If you dynamically add an additional `WHERE` clause at runtime to a read-only view object, its query gets nested into an inline view before applying the additional `WHERE` clause.

For example, suppose your query was defined as shown in the following example.

```
select PERSON_ID, EMAIL, FIRST_NAME, LAST_NAME
from PERSONS
where (upper(FIRST_NAME) like upper(:TheName)||'%'
   or  upper(LAST_NAME)  like upper(:TheName)||'%')
  and PERSON_ID between :LowUserId and :HighUserId
order by EMAIL
```

At runtime, when you set an additional `WHERE` clause like `person_type_code = :TheCustomerType` as the test program in How to Add a WHERE Clause with Named Bind Variables at Runtime illustrates, the framework nests the original query into an inline view like the following example.

```
SELECT * FROM(
select PERSON_ID, EMAIL, FIRST_NAME, LAST_NAME
from PERSONS
where (upper(FIRST_NAME) like upper(:TheName)||'%'
   or  upper(LAST_NAME)  like upper(:TheName)||'%')
  and PERSON_ID between :LowUserId and :HighUserId
order by EMAIL) QRSLT
```

Then the framework adds the dynamic `WHERE` clause predicate at the end, so that the final query the database sees is like the following example.

```
SELECT * FROM(
select PERSON_ID, EMAIL, FIRST_NAME, LAST_NAME
from PERSONS
where (upper(FIRST_NAME) like upper(:TheName)||'%'
   or  upper(LAST_NAME)  like upper(:TheName)||'%')
  and PERSON_ID between :LowUserId and :HighUserId
order by EMAIL) QRSLT
WHERE person_type_code = :TheCustomerType
```

This query "wrapping" is necessary in the general case since the original query could be arbitrarily complex, including SQL `UNION`, `INTERSECT`, `MINUS`, or other operators that combine multiple queries into a single result. In those cases, simply "gluing" the additional runtime `WHERE` clause onto the end of the query text could produce unexpected results because, for example, it might apply only to the *last* of several

UNION'ed statements. By nesting the original query verbatim into an inline view, the view object guarantees that your additional WHERE clause is correctly used to filter the results of the original query, regardless of how complex it is. The consequence (that results in an ORA-00904 error) is that the dynamically added WHERE clause can refer only to columns that have been selected in the original query.

The simplest solution is to add the dynamic query column names to the end of the query's SELECT list in the Query page of the overview editor for the view object. Just adding the new column name at the end of the existing SELECT list — of course, preceded by a comma — is enough to prevent the ORA-00904 error: JDeveloper will automatically keep your view object's attribute list synchronized with the query statement. Alternatively, Inline View Wrapping at Runtime explains how to disable this query nesting when you don't require it.

The test client program described in How to Add a WHERE Clause with Named Bind Variables at Runtime now produces the following results.

```
---
116 S. Baida
---
116 S. Baida
---
---
116 S. Baida
```

What You May Need to Know About Named Bind Variables

There are several things you may need to know about named bind variables, including the runtime errors that are displayed when bind variables have mismatched names and the default value for bind variables.

Errors Related to the Names of Bind Variables

You need to ensure that the list of named bind variables that you reference in your SQL statement matches the list of named bind variables that you've defined in the **Bind Variables** section of the Query page for the view object overview editor. Failure to have these two agree correctly can result in one of the following two errors at runtime.

If you use a named bind variable in your SQL statement but have not defined it, you'll receive an error like this:

```
(oracle.jbo.SQLStmtException) JBO-27122: SQL error during statement preparation.
## Detail 0 ##
(java.sql.SQLException) Missing IN or OUT parameter at index:: 1
```

On the other hand, if you have defined a named bind variable, but then forgotten to reference it or mistyped its name in the SQL, then you will see an error like this:

```
oracle.jbo.SQLStmtException: JBO-27122: SQL error during statement preparation.
## Detail 0 ##
java.sql.SQLException: Attempt to set a parameter name that does not occur in the
 SQL: LowUserId
```

To resolve either of these errors, double-check that the list of named bind variables in the SQL matches the list of named bind variables in the **Bind Variables** section of the Query page for the overview editor.

Default Value of NULL for Bind Variables

If you do not supply a default value for your named bind variable, it defaults to `NULL` at runtime. This means that if you have a `WHERE` clause like:

```
PERSON_ID = :TheCustomerId
```

and you do not provide a default value for the `TheCustomerId` bind variable, it will default to `NULL` and cause the query to return no rows. Where it makes sense for your application, you can leverage SQL functions like `NVL()`, `CASE`, `DECODE()`, or others to handle the situation as you require. For example, the following `WHERE` clause fragment allows the view object query to match any name if the value of `:TheName` is `null`.

```
upper(FIRST_NAME) like upper(:TheName)||'%'
```

Working with Row Finders

You use row finders using ADF view criteria to retrieve specific rows within a row set. You can use a bind variable in the view criteria statement, where the bind variable's value is value of the filter parameter.

Row finders are objects that the application may use to locate specific rows within a row set using a view criteria. You can define a row finder when you need to perform row lookup operations on the row set and you do not want the application to use row key attributes to specify the row lookup.

Currently, row finders that you define at design time can participate in these scenarios where non-row key attribute lookup is desired at runtime:

- When you expose the row finder in an ADF Business Components web service, the end user may initiate row updates on the specific rows of the service view instance that match one or more row-finder mapped, non-key attribute values. For information about this use case, see What You May Need to Know About View Criteria and Row Finder Usage.
- Programmatically in the application when you want to obtain a set of rows that match one or more row-finder mapped, non-key attribute values. For information about this use case, see Programmatically Invoking the Row Finder.

You can specify the row finder to locate all row matches or a specified number of row matches. If the row finder locates more rows than the fetch limit you specify, you can enable the row finder to throw an exception.

The view object definition with row finder includes the following elements.

1. A specified view criteria that applies one or more view criteria items to filter the owning view object. View criteria items are defined on the owning view object attributes that determine the row match and must be defined by bind variables to allow the row finder to supply the values at runtime.
2. A mapping between view criteria bind variables and the source of the value: either an attribute of the view object or an attribute value expression that need not necessarily map to attributes on the owning view object.

   Use an expression row finder in use cases where you need to filter by one or more attributes that are not mapped to the owning view object and use an attribute row finder in use cases where you need to lookup a row with an attribute key value from the owning view object.

For example, where EmpVO has an attribute email, an EmpVO row finder may locate a specific employee row using a view criteria defined on the EmpVO and the email address of the employee. The view criteria for the EmpVO might look similar to the one shown in the following example. In this example, the view criteria statement uses a bind variable EmailBindVar to set the value of the email attribute on the owning view object EmpVO.

```
 ( (UPPER(EmpEO.EMAIL) = UPPER(:EmailBindVar) ) )
```

The application may invoke the row finder by applying values to the mapped view criteria bind variables in a variety of ways:

- An ADF Business Components web service that makes uses of a single row in an upsert or merge operation. The application must invoke the row finder on the owning view object by using bind variables that map to locator attributes of the owning view object.
- A search field that filters a set of master rows with a value from the child records. The application must invoke the row finder on the master view object that participates in a master-detail relationship by using bind variables that map to a WHERE clause type expression containing one or more attributes of the child view object.
- An ADF Business Components web service that makes use of an update operation may display a form with input fields that are bound to view object attributes that the row finder maps to view criteria bind variables.

  For details about enabling operations of Business Components web services, see How to Enable the Application Module Service Interface.
- The application may invoke the row finder on the view object and programmatically set the value of the mapped view object attributes to the view criteria bind variables. See Programmatically Invoking the Row Finder.

How to Add Row Finders to a View Object Definition

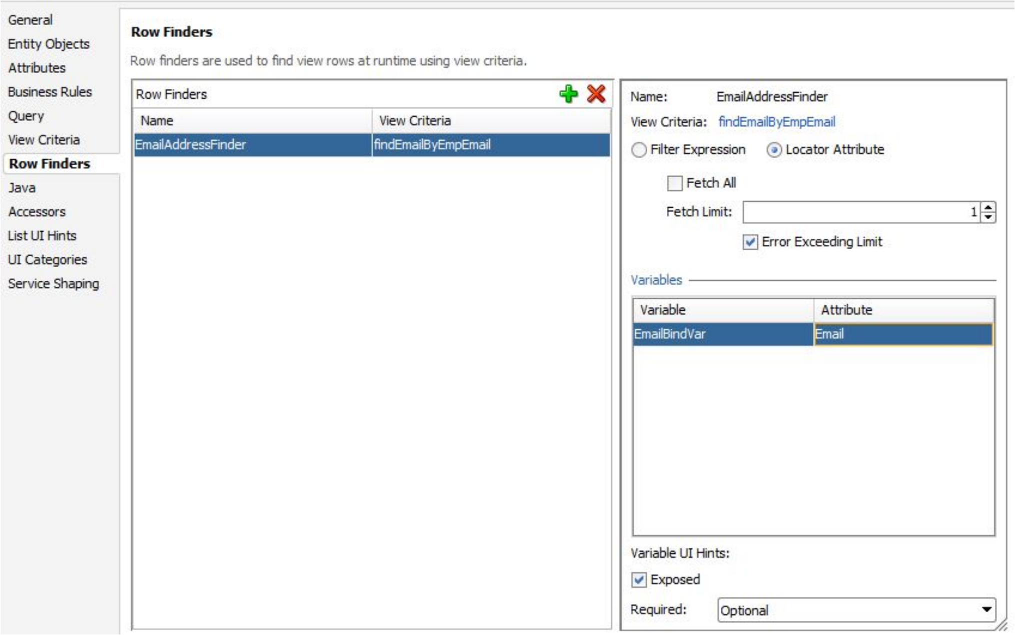
You use the Row Finders page of the view object's overview editor to define the row finder. In the editor, you define a value source for each bind variable of the view criteria that you select. Attributes of the view object or attribute value expressions are both valid sources for bind variables defined by the row finder.

The row finder that you define can map any number of view criteria bind variables to a valid value source. Mapping additional variables in the row finder definition will result in more restrictive row matches. In this regard, row finders are similar to query search operations that do not alter the row set. Using an appropriately defined row finder, the application can locate a single, specific row and the allow the end user to perform row updates on the row.

As a rule of thumb, use an attribute expression row finder in use cases that map to filtering the results with a WHERE clause and use an attribute row finder to lookup a row with a key value.

Figure 5-36 shows the overview editor with a row finder that maps the view criteria bind variable EmailBindVar specified in the findEmpByEmpEmail view criteria to the attribute Email defined by the view object EmpVO as its value source.

Figure 5-36 View Object Overview Editor with Row Finder



Row Finders

Row finders are used to find view rows at runtime using view criteria.

Row Finders

Name	View Criteria
EmailAddressFinder	findEmailByEmpEmail

Name: EmailAddressFinder
View Criteria: findEmailByEmpEmail
○ Filter Expression ● Locator Attribute
□ Fetch All
Fetch Limit: 1
☑ Error Exceeding Limit

Variables

Variable	Attribute
EmailBindVar	Email

Variable UI Hints:
☑ Exposed
Required: Optional

Note:

You can define row finders on view objects in a master-detail hierarchy when you want to filter the master by attributes of the detail view object. In this scenario, use map the bind variable to an expression that contains attributes of the child view object.

Before you begin:

It may be helpful to have an understanding of the row finder. For more information, see Working with Row Finders.

It may be helpful to have an understanding of view criteria. For more information, see Working with Named View Criteria.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for View Objects.

You will need to complete these tasks:

- Create the desired view objects, as described in Populating View Object Rows from a Single Database Table or Working with Multiple Tables in Join Query Results.
- Define a view criteria on the view object, as described in How to Create Named View Criteria Declaratively.
- Optionally, add updatable transient attributes to the view object, as described in How to Add a Transient Attribute. The transient attribute may be set programmatically by the application or exposed in an ADF Business Components

web service payload to solicit the value from an end user. The transient attributes must be defined as updatable to receive the criteria lookup value.

To create a row finder for a view object:

1. In the Applications window, double-click the view object for which you want to create the row finder.
2. In the overview editor, click the **Row Finders** navigation tab and then click the **Add Row Finder** button.
3. In the **Name** field, rename the row finder.

   For example, a row finder that you define to locate an employee by their email address, might be named `EmpByEmailRowFinder`.
4. Select the new row finder in the **Row Finders** list and then, in the **View Criteria** dropdown, select the view criteria that filters the view object row set.

   The desired row finder should appear highlighted, as shown in Figure 5-36. The view criteria list will be empty unless you have created a view criteria for the row finder to use.
5. When you want the end user of an ADF Business Components web service to supply a value, leave **Attribute** selected, select the bind variable from the list, and then, in the **Attribute** dropdown, select the attribute from the view object that supplies the bind variable value at runtime.
6. When you want to filter the results in affect by using a WHERE clause, select **Filter Expression**, select the bind variable from the list, and then, click **New expression method** to enter the attribute value expression and click **OK**.
7. Deselect **Fetch All** when you want to specify the number of rows that the row finder is allowed to match.

   When you enter a number of rows for **Fetch Limit**, you can also select **Error Exceeding Limit** to enable Oracle ADF to throw an exception when more matching rows exist in the database than the number you specify for **Fetch Limit**.
8. Leave **Exposed** selected when you want to associate previously defined UI hints with the current bind variable. If you know that the bind variable will not be utilized by the user interface, you can deselect **Exposed**.
9. You can choose whether you want to make the value of a named bind variable required, optional, or selectively required for the row finder that references the bind variable as a view criteria item.

   Press **F1** to view the JDeveloper Help Center topic for details on the **Required** dropdown options.

What Happens When You Define a Row Finder

When you create a view object row finder, the view object definition contains all the metadata required by the row finder, including the row finder definition itself. As the following example shows, the metadata for a row finder `RowFinder` includes a bind variable `EmailBindVar` in the `<Variable>` element, a transient attribute `TrEmpEmail` in the `<ViewAttribute>` element, a view criteria `findEmpByEmpEmail` and a view criteria row `EmpVOCriteria_Row0` in the `<ViewCriteria>` element, and finally the row finder `EmpByEmailRowFinder` in the `<RowFinders>` element of the view object definition.

The row finder <VarAttributeMapping> subelement maps the view criteria bind variable EmailBindVar to the transient attribute TrEmpEmail of the view object, which allows the end user to supply a value at runtime and allows the application to invoke the row finder with the required criteria attribute. The <ViewCriteriaItem> subelement sets the criteria attribute Email to the value of the bind variable EmailBindVar.

At runtime, when the row finder is invoked on the view object, the transient attribute values passed to one or more criteria attributes identify the matching rows in the row set. In this example, the email address of the employee is used to match the employee record from the EMPLOYEE table. The row finder locates the record for the employee without the need to pass a row key value.

```
<ViewObject
  xmlns="http://xmlns.oracle.com/bc4j"
  Name="EmpVO"
  SelectList="EmpEO.ID,
       EmpEO.LAST_NAME,
       EmpEO.FIRST_NAME"
       EmpEO.USERID"
       EmpEO.DEPT_ID"
       EmpEO.EMAIL"
  FromList="S_EMP EmpEO"
  ...
  <Variable
    Name="EmailBindVar"
    Kind="viewcriteria"
    Type="java.lang.String"/>
  <EntityUsage
    Name="EmpEO"
    Entity="model.entities.EmpEO"/>
  ...
  <ViewAttribute
    Name="TrEmpEmail"
    IsUpdateable="false"
    IsSelected="false"
    IsPersistent="false"
    PrecisionRule="true"
    Type="java.lang.String"
    ColumnType="CHAR"
    AliasName="VIEW_ATTR"
    SQLType="VARCHAR"/>
  ...
  <ViewCriteria
    Name="findEmpByEmpEmail"
    ViewObjectName="model.views.EmpVO"
    Conjunction="AND">
    ...
    <ViewCriteriaRow
      Name="EmpVOCriteria_row_0"
      UpperColumns="1">
      <ViewCriteriaItem
        Name="Email"
        ViewAttribute="Email"
        Operator="="
        Conjunction="AND"
        Value=":EmailBindVar"
        IsBindVarValue="true"
        Required="Optional"/>
    </ViewCriteriaRow>
  </ViewCriteria>
  ...
  <RowFinders>
    <AttrValueRowFinder
      Name="EmpByEmailRowFinder"
      FetchLimit="1"
      ErrorOnExceedingLimit="true">
      <ViewCriteriaUsage
        Name="findEmpByEmpEmail"
        FullName="model.views.EmpVO.findEmpByEmpEmail"/>
      <VarAttributeMap>
        <VariableAttributeMapping
          Variable="EmailBindVar"
          Attribute="TrEmpEmail"/>
      </VarAttributeMap>
    </AttrValueRowFinder>
  </RowFinders>
</ViewObject>
```

What You May Need to Know About View Criteria and Row Finder Usage

Row finders and view criteria can both be used either programmatically or declaratively to filter a view object row set. The row finder differs from the view criteria because it defines a map between transient attributes that you define on the view object and view criteria attribute bind variables. Ultimately, the row finder matches one or more rows on the view object by applying the view criteria to the view object with the bind variable defined by a transient attribute value that is resolved at runtime.

The transient attribute mapped by the row finder allows the row finder to be used in application use cases that would not be suitable for view criteria alone. In particular, row finders are most useful when ADF Business Components web services enable CRUD operations. In this case, the row finder gives the ADF Business Components web services developer the ability to expose CRUD operations without needing obscure row key values.

Fusion web application developers can create a web service data control to expose the view objects of an ADF Business Components web service in the JDeveloper Data Controls panel. The exposed row-finder mapped transient attributes of the view object can then be used to design an input form that allows the end user to make the desired updates. For example, an end user may supply the value of a familiar attribute, such as a person's email address, to match a particular person row and then initiate an update operation on any attribute of that row, such as the one that would allow a name change.

For details about row finder support in ADF web services, see What You May Need to Know About Row Finders and the ADF Web Service Operations. For details about creating data controls based on ADF web services, see Using ADF Data Controls in *Developing Applications with Oracle ADF Data Controls*.

Note that the capability of row finders to match rows using non-key attributes extends to master-detail related view objects, as described in How to Find Rows of a Master View Object Using Row Finders.

Programmatically Invoking the Row Finder

When you define a row finder you invoke the row finder on the view object. If the row finder definition specifies a row limit, the executed row finder will throw a `RowFinderFetchLimitExceededException` exception.

To invoke a row finder that you define on a view object, follow these basic steps (as illustrated in the following example):

1. Find the view object that defines the row finder.
2. Find the row finder from the view object.
3. Create a name value pair for each row finder bind variable mapping.
4. Set each bind variable using a row-finder mapped transient attribute.
5. Invoke the row finder on the desired view object.
6. Throw an exception when the number of matching rows exceeds the row finder fetch limit.

At runtime, when the row finder is invoked on the view object, row-finder mapped transient attribute values populated by `setAttribute()` are set on criteria attributes to identify the matching rows in the row set. In the following example, the email address of the employee is used to match the employee record on the `EmpView` view object. The row finder `EmpByEmailRowFinder` locates the record for the employee using the transient attribute `TrEmpEmail` to specify the criteria attribute without the need to pass a row key value.

```
package model;

import oracle.jbo.*;
import oracle.jbo.client.Configuration;
import oracle.jbo.server.RowFinder;
import oracle.jbo.server.ViewObjectImpl;

public class TestClient
{
   public TestClient()
   {
      super();
   }

   public static void main(String[] args)
   {
      TestClient testClient = new TestClient();
      String amDef = "model.AppModule";
      String config = "AppModuleLocal";
      ApplicationModule am =
                 Configuration.createRootApplicationModule(amDef, config);

      // 1. Find the view object with the row finder
      ViewObjectImpl vo = (ViewObjectImpl)am.findViewObject("EmpView1");
      Row r;
      RowIterator ri;

      // 2. Find the row finder
      RowFinder finder = vo.lookupRowFinder("EmpByEmailRowFinder");
      // 3. Create name value pairs for the row finder
      NameValuePairs nvp = new NameValuePairs();
```

```
        // 4. Set the row-finder mapped transient attribute
        nvp.setAttribute("TrEmpEmail", "cee.mague@company.com");
        // 5. Invoke the row finder
        try
        {
           ri = finder.execute(nvp, vo);
        }
        // 6. Throw an exception when row match exceeds specified limit
        catch(RowFinderFetchLimitExceededException e)
        {
           System.out.println("Warning: more than one row match exists.");
        }

        while (ri.hasNext())
        {
          r = ri.next();
          System.out.println("Find emp row by email finder: " +
             r.getAttribute("FirstName") + "/" + r.getAttribute("LastName"));
        }

        Configuration.releaseRootApplicationModule(am, true);
     }
  }
```

Working with List of Values (LOV) in View Object Attributes

You can define an attribute of a view object in the ADF Model project as a LOV attribute and make it available on the user interface for user input based on selection.

Edit forms displayed in the user interface portion of your application can utilize LOV-enabled attributes that you define in the data model project to predetermine a list of values for individual input fields. For example, the edit form may display a list of countries to populate the corresponding address field in the edit form. To facilitate this common design task, ADF Business Components provides declarative support to specify the LOV usage in the user interface.

Defining an LOV for attributes of a view object in the data model project greatly simplifies the task of working with list controls in the user interface. Because you define the LOV on the individual attributes of the view object, you can customize the LOV usage for an attribute once and expect to see the list component in the form wherever the attribute appears.

> **Note:**
>
> In order for the LOV to appear in the UI, the LOV usage must exist before the user interface designer creates the databound form. Defining an LOV usage for an attribute referenced by an existing form will not change the component that the form displays to an LOV.

You can define an LOV for any view object attribute that you anticipate the user interface will display as a selection list. The characteristics of the attribute's LOV definition depend on the requirements of the user interface. The information you gather from the user interface designer will determine the best solution. For example, you might define LOV attributes in the following cases:

- When you need to display attribute values resulting from a view object query against a business domain object.

  For example, define LOV attributes to display the list of suppliers in a purchase order form.

- When you want to display attribute values resulting from a view object query that you wish to filter using a parameter value from any attribute of the LOV attribute's current row.

  For example, define LOV attributes to display the list of supplier addresses in a purchase order form but limit the addresses list based on the current supplier.

  If you wish, you can enable a second LOV to drive the value of the parameter based on a user selection. For example, you can let the user select the current supplier to drive the supplier addresses list. In this case, the two LOVs are known as a **cascading list**.

Before you can define the LOV attribute, you must create a data source view object in your data model project that queries the eligible rows for the attribute value you want the LOV to display. After this, you work entirely on the base view object to define the LOV. The base view object is the one that contains the primary data for display in the user interface. The LOV usage will define the following additional view object metadata:

- A view accessor to access the data source for the LOV attribute. The view accessor is the ADF Business Components mechanism that lets you obtain the full list of possible values from the row set of the data source view object.
- Optionally, supplemental values that the data source may return to attributes of the base view object other than the data source attribute for which the list is defined.
- User interface hints, including the type of list component to display, attributes to display from the current row when multiple display attributes are desirable, and a few options specific to the choice list component.

> **Note:**
>
> The LOV feature does not support the use of attribute validation to validate the display list. Any validation rules that may have been defined on data source attributes (including supplemental ones) will be suppressed when the list is displayed and will therefore not limit the LOV list. Developers must ensure that the list of values returned from the data source view object contains only desired, valid values.

The general process for defining the LOV-enabled attribute relies on the Create List of Values dialog that you display for the base view object attribute.

To define the LOV-enabled attribute, follow this general process:

1. Open the Create List of Values dialog for the base attribute.
2. Create a new view accessor definition to point to the data source view object or select an existing view accessor that the base view object already defines.

   Always create a new view accessor for each use case that your wish to support. Oracle recommends that you do not reuse a view accessor to define multiple LOV lists that happen to rely on the same data source. Reusing a view accessor can produce unintended results at runtime.

3. Optionally, you can filter the view accessor by creating a view criteria using a bind variable that obtains its value from any attribute of the base view object's current row.

   If you create a view criteria to filter the data source view object, you may also set a prerequisite LOV on the attribute of the base view object that you use to supply the value for the view criteria bind variable. LOV lists that cooperate in this manner, are known as cascading LOV lists. You set cascading LOV lists when you want the user's selection of one attribute to drive the options displayed in a second attribute's list.

4. Select the list attribute from the view accessor's data source view object.

   This maps the attribute you select to the current attribute of the base view object.

5. Optionally, select list return values to map any supplemental values that your list returns to the base view object.

6. Select user interface hints to specify the list's display features.

7. Save the attribute changes.

Once you create the LOV-enabled attribute, the user interface designer can create the list component in the web page by dragging the LOV-enabled attribute's collection from the Data Controls panel. For further information about creating a web page that display the list, see Creating Databound Selection Lists and Shuttles . Specifically, for information about working with LOV-enabled attributes in the web page, see How to Create a Model-Driven List.

How to Define a Single LOV-Enabled View Object Attribute

When an edit form needs to display a list values that is not dependent on another selection in the edit form, you can define a view accessor to point to the list data source. For example, assume that a purchase order form contains a field that requires the user to select the order item's supplier. In this example, you would first create a view accessor that points to the data source view object (`SuppliersView`). You would then set the LOV on the `SupplierDesc` attribute of the base view object (`PurchaseOrdersView`). Finally, you would reference that view accessor from the LOV-enabled attribute (`SupplierDesc`) of the base view object and select the data source attribute (`SupplierDesc`).

You will use the Create List of Values dialog to define an LOV-enabled attribute for the base view object. The dialog lets you select an existing view accessor or create a new one to save with the LOV-attribute definition.

Before you begin:

It may be helpful to have an understanding of LOV-enabled attributes. For more information, see Working with List of Values (LOV) in View Object Attributes.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for View Objects.

You will need to complete this task:

Create the desired view objects, as described in How to Create an Entity-Based View Object, and How to Create a Custom SQL Mode View Object.

To define an LOV that displays values from a view object attribute:

1. In the Applications window, double-click the view object that contains the attribute you wish to enable as an LOV.
2. In the overview editor, click the **Attributes** navigation tab.
3. In the Attributes page, select the attribute that is to display the LOV, and then click the **List of Values** tab and click the **Add list of values** button.

   Use the Create List of Values dialog to create the LOV on the attribute you have currently selected in the attribute list of the overview editor. JDeveloper assigns a unique name to identify the LOV usage. For example, the metadata for the attribute `SupplierDesc` will specify the name `SupplierDescLOV` to indicate that the attribute is LOV enabled.
4. In the Create List of Values dialog, on the Configuration tab, click the **Create new view accessor** button to add a view accessor to the view object you are currently editing.

   Alternatively, you can expand **List Data Source** and select among the existing view accessors. The dropdown list displays all view accessors that you have added to the view object you are editing.
5. In the View Accessors dialog, select the view object definition or shared view instance that defines the data source for the attribute and shuttle it to the view accessors list.

   By default, the view accessor you create will display the same name as the view object. You can edit the accessor name to supply a unique name. For example, assign the name `SuppliersViewAccessor` for the `SuppliersView` view object.

   The view instance is a view object usage that you have defined in the data model of a **shared application module**. For more information about using shared view instances in an LOV, see How to Create an LOV Based on a Lookup Table.
6. Click **OK** to save the view accessor definition for the view object.
7. In the Create List of Values dialog, expand **List Data Source** and select the view accessor you created for the base view object to use as the data source. Then select the same attribute from this view accessor that will provide the list data for the LOV-enabled attribute.

   The editor creates a default mapping between the list data source attribute and the LOV-enabled attribute. For example, the attribute `SuppliersDesc` from the `PurchaseOrdersView` view object would map to the attribute `SuppliersDesc` from the `SuppliersViewAccessor` view accessor.

   The editor does not allow you to remove the default attribute mapping for the attribute for which the list is defined.
8. If you want to specify supplemental values that your list returns to the base view object, click the **Create return attribute map** button in the **List Return Values** section and map the desired base view object attributes with attributes accessed by the view accessor.

   Supplemental attribute return values are useful when you do not require the user to make a list selection for the attributes, yet you want those values, as determined by the current row, to participate in the update. For example, to map the attribute

SupplierAddress from the PurchaseOrdersView view object, you would choose the attribute SupplierAddress from the SuppliersViewAccessor view accessor.

9. Click **OK**.

How to Define Cascading Lists for LOV-Enabled View Object Attributes

When the application user interface requires a list of values in one input field to be dependent on the user's entry in another field, you can create attributes that will display as cascading lists in the user interface. In this case, the list of possible values for the LOV-enabled attributes might be different for each row. As the user changes the current row, the LOV values vary based on the value of one or more controlling attribute values in the LOV-enabled attribute's view row. To apply the controlling attribute to the LOV-enabled attribute, you will create a view accessor to access the data source view object with the additional requirement that the accessor filters the list of possible values based on the current value of the controlling attribute. To filter the LOV-enabled attribute, you can edit the view accessor to add a named view criteria with a bind variable to obtain the user's selection.

For example, assume that a purchase order form contains a field that requires the user to select the supplier's specific site and that the available sites will depend on the order's already specified supplier. To implement this requirement, you would first create a view accessor that points to the data source view object. The data source view object will be specific to the LOV usage, because it must perform a query that filters the available supplier sites based on the user's supplier selection. You might name this data source view object definition SupplierIdsForCurrentSupplierSite to help distinguish it from the SupplierSitesView view object that the data model already contains. The data source view object will use a named view criteria (SupplierCriteria) with a single view criteria item set by a bind variable (TheSupplierId) to obtain the user's selection for the controlling attribute (SupplierId).

You would then set the LOV on the SupplierSiteId attribute of the base view object (PurchaseOrdersView). You can then reference the view accessor that points to the data source view object from the LOV-enabled attribute (PurchaseOrdersView.SupplierSiteId) of the base view object. Finally, you must edit the LOV-enabled attribute's view accessor definition to specify the corresponding attribute (SupplierIdsForCurrentSupplierSite.SupplierSiteId) from the view object as the data source and, importantly, source the value of the bind variable from the view row's result using the attribute SupplierId.

To define cascading lists for LOV-enabled view object attributes:

1. Create a data source view object to control the cascading list.
2. Create a view accessor to filter the cascading list.

Creating a Data Source View Object to Control the Cascading List

The data source view object defines the controlling attribute for the LOV-enabled attribute. To make the controlling attribute accessible to the LOV-enabled attribute of the base view object, you must define a named view criteria to filter the data source attribute based on the value of another attribute. Because the value of the controlling attribute is expected to change at runtime, the view criteria uses a bind variable to set the controlling attribute.

Before you begin:

It may be helpful to have an understanding of cascading LOV-enabled attributes. For more information, see How to Define Cascading Lists for LOV-Enabled View Object Attributes.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for View Objects.

To define the view criteria for the data source to be referenced by the LOV-enabled attribute:

1. In the Applications window, double-click the view object that you created to query the list of all possible values for the controlling attribute.

   For example, if the LOV-enabled attribute `SupplierSiteId` depends on the controlling attribute `SupplierId` value, you might have created the data source view object `SupplierIdsForCurrentSupplierSite` to query the list of all supplier sites.
2. In the overview editor, click the **Query** navigation tab.
3. In the Query page, in the **Bind Variables** section, click the **Create New Bind Variable** button to add a bind variable to the data source view object.

   For example, for a data source view object `SupplierIdsForCurrentSupplierSite` used to query the list of all supplier sites, you would create the bind variable `TheSupplierId`, since it will be the controlling attribute for the LOV-enabled attribute.
4. In the **New Variable** dialog, enter the name and select the type of the bind variable and click **OK**.

   By default, the view accessor you create will display the same name as the view object instance. You can edit the accessor name to supply a unique name. For example, assign the name `CurrencyLookupViewAccessor` for the `CurrencyLookupView` view object instance.
5. In the overview editor, click the **View Criteria** navigation tab and click the **Create New View Criteria** button to add the view criteria to the data source view object you are currently editing.
6. In the Create View Criteria dialog, click **Add Group** and define a single **Criteria Item** for the group as follows:
   - Enter a **Criteria Name** to identify the view criteria. For example, you might enter the name `SupplierCriteria` for the `SupplierIdsForCurrentSupplierSite`.
   - Select the controlling attribute from the **Attributes** list. For example, you would select the `SupplierSiteId` attribute from the `SupplierIdsForCurrentSupplierSite`.
   - Select **equal to** from the view criteria **Operator** list.
   - Select **Bind Variable** from the view criteria **Operand** list.

- Select the name of the previously defined bind variable from the **Parameter** list.
- Select among the following bind variable configuration options to determine whether or not the value is required by the parent LOV:

  Optional from the **Validation** menu and deselect **Ignore Null Values** when you want to configure cascading LOVs where the parent LOV value is required. This combination supports the cascading LOV use case where no selection in the parent LOV returns *no* rows in the child LOV. The `WHERE` clause shown in the Edit View Criteria dialog should look similar to `((SupplierIdsForCurrentSupplierSite.SUPPLIER_ID = :TheSupplierId))`.

  Optional from the **Validation** menu and leave **Ignore Null Values** selected (default) when you want to configure cascading LOVs where the parent LOV value is optional. This combination supports the cascading LOV use case where no selection in the parent LOV returns *all* rows in the child LOV. The `WHERE` clause shown in the Edit View Criteria dialog should look similar to `(((SupplierIdsForCurrentSupplierSite.SUPPLIER_ID = :TheSupplierId) OR ( :TheSupplierId IS NULL )))`.

  For more details about these settings, see What You May Need to Know About Bind Variables in View Criteria. Do not select **Required** for the Validation option for cascading LOVs, because no selection in the parent LOV will cause a validation error.

7. Click **OK**.

Creating a View Accessor to Filter the Cascading List

To populate the cascading LOV-enabled attribute, you must first set up a named view criteria on a data source view object. To make the LOV-enabled attribute of the base view object dependent on the controlling attribute of the data source view object, you then add a view accessor to the LOV-enabled attribute of the base view object and reference the previously defined data source view object's named view criteria.

Before you begin:

It may be helpful to have an understanding of cascading LOV-enabled attributes. For more information, see How to Define Cascading Lists for LOV-Enabled View Object Attributes.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for View Objects.

You will need to complete this task:

> Create the data source view object and named view criteria, as described in Creating a Data Source View Object to Control the Cascading List.

To create a view accessor that filters display values for an LOV-enabled attribute based on the value of another attribute in the same view row:

1. In the Applications window, double-click the base view object that contains the attribute you want to use the filtered view accessor as the list data source.

   For example, the base view object `PurchaseOrdersView` might contain the attribute `SupplierSiteId` that will depend on the value of the controlling attribute `SupplierId`.

2. In the overview editor, click the **Attributes** navigation tab.
3. In the Attributes page, select the attribute that is to filter the cascading LOV, and then click the **List of Values** tab and click the **Add List of Values** button.
4. In the Create List of Values dialog, click the **Create New View Accessor** button to add a view accessor to the view object you are currently editing.

   Alternatively, you can expand **List Data Source** and select among the existing view accessors. The dropdown list displays all view accessors that you have added to the view object you are editing.
5. In the View Accessors dialog, select the view object instance name that you created for the data source view object and shuttle it to the view accessors list.
6. With the new view accessor selected in the dialog, click **Edit**.
7. In the Edit View Accessor dialog, apply the previously defined view criteria to the view accessor and provide a value for the bind variable as follows:
   - Click the data source view object's view criteria in the **Available** list and add it to the Selected list. For example, you would select `SupplierCriteria` from the `SupplierIdsForCurrentSupplierSite` view object definition.
   - Set the value for the bind variable to the name of the controlling attribute. The attribute name must be identical to the base view object's controlling attribute. For example, if the base view object `PurchaseOrdersView` contains the LOV-enabled attribute `SupplierSiteId` that depends on the value of the controlling attribute `SupplierId`, you would enter `SupplierId` for the bind variable value.
   - Select the name of the previously defined bind variable from the **Parameter** list.
   - Select **Required** from the **Usage** dropdown list.
8. Click **OK** to save the view accessor definition for the base view object.
9. In the Attributes page of the overview editor, select the attribute that is to display the LOV, and then click the **List of Values** tab and click the **Add list of values** button.
10. In the Create List of Values dialog, expand **List Data Source** and select the view accessor you created for the data source view object instance to use as the data source. Then select the controlling attribute from this view accessor that will serve to filter the attribute you are currently editing.

    The editor creates a default mapping between the view object attribute and the LOV-enabled attribute. You use separate attributes in order to allow the bind variable (set by the user's controlling attribute selection) to filter the LOV-enabled attribute. For example, the LOV-enabled attribute `SupplierId` from the `PurchaseOrdersView` view object would map to the controlling attribute `SupplierSiteId` for the `SupplierIdsForCurrentSupplierSiteViewAccessor`. The runtime automatically supports these two cascading LOVs where the row set and the base row attribute differ.
11. Click **OK**.

How to Specify Multiple LOVs for a Single LOV-Enabled View Object Attribute

Another way to vary the list of values that your application user interface can display is to define multiple list of values for a single LOV-enabled view object attribute. In contrast to a cascading list, which varies the list contents based on a dependent LOV list selection, an LOV-enabled switcher attribute with multiple LOVs lets you vary the entire LOV itself. The LOV choice to display is controlled at runtime by the value of an attribute that you have defined specifically to resolve to the name of the LOV to apply.

For example, you might want to define one LOV to apply in a create or edit form and another LOV to apply for a search component. In the first case, the LOV-enabled attribute that the form can use is likely to be an entity-based view accessor that is shared across all the view objects that reference the entity. The entity-based view accessor is useful for user interface forms because a single accessor definition can apply to each instance of the same LOV in the forms. However, in the case of the search component, LOV definitions based on view accessors derived from an underlying entity will not work. The LOV definitions for search components must be based on view accessors defined in the view object. Note that when the user initiates a search, the values in the criteria row will be converted into WHERE clause parameters. Unlike a regular view row displayed in create or edit type forms, the criteria row is not backed by an entity. In this scenario, one LOV uses the entity-based accessor as a data source and a second LOV uses the view object-based accessor as a data source.

To address this requirement to define multiple LOV lists that access the same attribute, you add a switcher attribute to the base view object. For example, you might add a ShipperLOVSwitcher attribute for the Orders view object that resolves through an expression to the name of the LOV to display. Such an expression can specify two LOVs that may apply to the ShipperID attribute:

```
(adf.isCriteriaRow) ? "LOV_ShipperID_ForSearch" : "LOV_ShipperID"
```

This expression would appear in the **Value** field of the switcher attribute. At runtime, in the case of the search component, the expression resolves to the value that identifies the view object-based accessor LOV. In the case of the create or edit form, the expression resolves to the value that identifies the entity-based accessor LOV.

You will use the Create List of Values dialog to add multiple LOV lists to an attribute of the base view object. You will also use the List of Values section in the Attributes page of the overview editor for the base view object to define the default LOV to display and the switcher attribute to apply.

Before you begin:

It may be helpful to have an understanding of LOV-enabled attributes. For more information, see Working with List of Values (LOV) in View Object Attributes.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for View Objects.

You will need to complete this task:

- Create the first LOV list for the attribute, as described in How to Define a Single LOV-Enabled View Object Attribute.

Note that the switcher attribute scenario requires that you create unique view accessors. You must not reuse a view accessor to define multiple LOV lists. Reusing a view accessor across various use cases can produce unintended results at runtime.

To specify additional LOV lists for a view object attribute with an existing LOV:

1. In the Applications window, double-click the view object that contains the attribute for which you want to specify multiple LOV lists.
2. In the overview editor, click the **Attributes** navigation tab.
3. In the Attributes page, select the desired attribute, and then click the **List of Values** tab and click the **Add list of values** button.
4. In the Create List of Values dialog, define the first LOV, as described in How to Define a Single LOV-Enabled View Object Attribute.

   When you define the LOV, change the name of the LOV to match the value returned by the attribute that you will use to determine which LOV your application applies to the LOV-enabled attribute.
5. After you define the first LOV, return to the **List of Values** tab of the Attributes page of the overview editor and, with the original attribute selected, click the **Add List of Values** button.

   If you have selected the correct attribute from the Attributes page of the overview editor, the **List of Values** section should display your previously defined LOV.
6. In the Create List of Values dialog, repeat the procedure described in How to Define a Single LOV-Enabled View Object Attribute to define each subsequent LOV.

   The name of each LOV must correspond to a unique value returned by the attribute that determines which LOV to apply to the LOV-enabled attribute.

   You must define the second LOV using a unique view accessor, but you may use any attribute. There are no restrictions on the type of LOV lists that you can add to an attribute with multiple LOV lists specified.

   After you finish defining the second LOV, the **List of Values** section in the view object overview editor changes to display additional features that you will use to control the selection of the LOV.
7. In the Attributes page of the overview editor, click the **List of Values** tab and use the **List of Values Switcher** dropdown list to select the attribute that will return the name of the List of Value to use.

   The dropdown list displays the attributes of the base view object. If you want your application to dynamically apply the LOV from the LOVs you have defined, your view object must define an attribute whose values resolve to the names of the LOVs you defined. If you have not added this attribute to the view object, be sure that the dropdown list displays **<None Specified>**. In this case, at runtime your application will display the LOV-enabled attribute with the default LOV and it will not be possible to apply a different LOV.
8. To change the default LOV to apply at runtime, choose the **Default** radio button corresponding to the desired LOV definition.

   The default LOV selection determines which list of values your application will display when the **List of Values Switcher** dropdown list displays **<None Specified>**. Initially, the first LOV in the overview editor **List of Values** section is the default.

9. To change the component that your application will use to display the various LOV lists, select from desired component from the **List Type UI Hint** dropdown list.

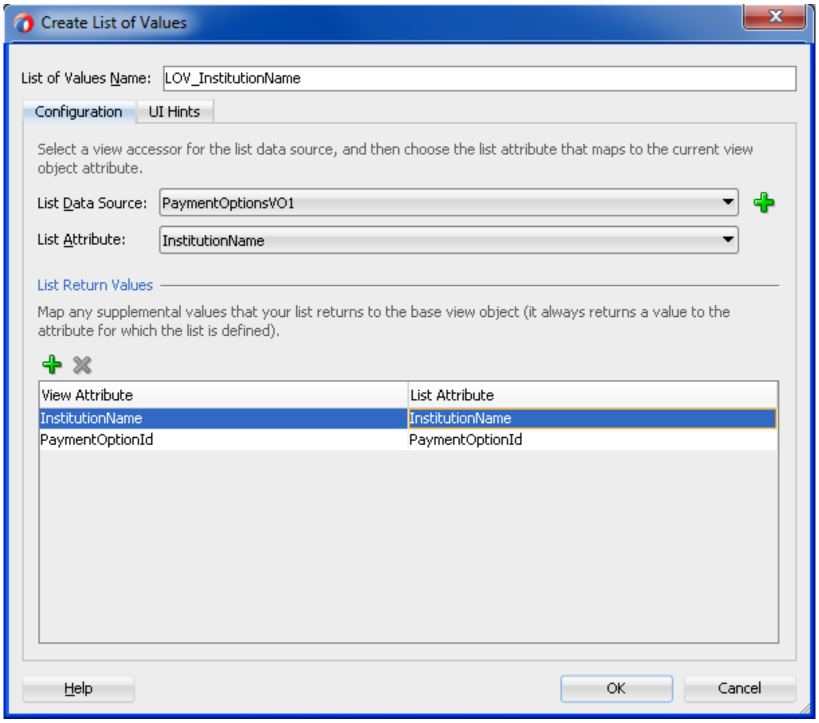
   The component you select will apply to all LOV lists. For a description of the available components, see How to Set User Interface Hints on a View Object LOV-Enabled Attribute.

How to Define an LOV to Display a Reference Attribute

Reference attributes that your view objects define are often desirable attributes to use as the source for LOV lists. Reference attributes belong to secondary entity usages that you have added to the view object to provide meaningful information beyond the entity usage's primary key attribute. For example, when you create an `OrderInfo` view object, the view object may define a secondary entity usage for `PaymentOptionsEO` to include billing information for the order, including the list of credit cards the end user has added to their account.

An LOV that you define for the secondary entity object needs to be able to update the primary attribute value, but to be meaningful to the end user, you typically hide the primary attribute in the list and display one or more reference attributes instead. In this case, your LOV might display the list of credit cards by institution name, but hide the payment option ID value that gets updated by the end user's selection. Figure 5-37 shows the LOV defined on the reference attribute `InstitutionName`. The **List Return Values** section of the Create List of Values dialog lists the reference attribute and then lists the primary key attribute `PaymentOptionId` to provide supplemental values.

Figure 5-37 Create View Criteria Dialog with Reference Attribute Specified



Create List of Values

List of Values Name: LOV_InstitutionName

Configuration | UI Hints

Select a view accessor for the list data source, and then choose the list attribute that maps to the current view object attribute.

List Data Source: PaymentOptionsVO1

List Attribute: InstitutionName

List Return Values

Map any supplemental values that your list returns to the base view object (it always returns a value to the attribute for which the list is defined).

View Attribute	List Attribute
InstitutionName	InstitutionName
PaymentOptionId	PaymentOptionId

Before you begin:

It may be helpful to have an understanding of reference attributes in secondary entity usages. For more information, see Working with List of Values (LOV) in View Object Attributes.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for View Objects.

You will need to complete this task:

> Create the entity-based view object and add a secondary entity usage that defines the desired LOV attributes, as described in How to Define a Single LOV-Enabled View Object Attribute. By default, the secondary entity usage will include the primary key attribute that you want the LOV to update.

To define reference attributes for an LOV:

1. In the Applications window, double-click the view object that contains the secondary entity usage attribute you wish to enable as an LOV.
2. In the overview editor, click the **Attributes** navigation tab.
3. In the Attributes page, select the desired reference attribute that is to display the LOV, click the **List of Values** tab, and then click the **Add List of Values** button.

   Use the Create List of Values dialog to create the LOV on the attribute you have currently selected in the attribute list of the overview editor. JDeveloper assigns a unique name to identify the LOV usage. For example, the metadata for the attribute `SupplierDesc` will specify the name `SupplierDescLOV` to indicate that the attribute is LOV-enabled.
4. In the Create List of Values dialog, click the **Create new view accessor** button to add a view accessor to the view object you are currently editing.

   Alternatively, you can expand **List Data Source** and select among the existing view accessors. The dropdown list displays all the view accessors that you have added to the view object you are editing.
5. In the View Accessors dialog, select the view object definition or shared view instance that defines the data source for the attribute and shuttle it to the view accessors list.

   By default, the view accessor you create will display the same name as the view object. You can edit the accessor name to supply a unique name. For example, assign the name `SuppliersViewAccessor` for the `SuppliersView` view object.
6. Click **OK** to save the view accessor definition for the view object.
7. In the Create List of Values dialog, expand **List Data Source** and select the view accessor you created for the base view object to use as the data source. Then select the same attribute from this view accessor that will provide the list data for the LOV-enabled attribute.

   The editor creates a default mapping between the list data source attribute and the LOV-enabled attribute. For example, the attribute `SuppliersDesc` from the `PurchaseOrdersView` view object would map to the attribute `SuppliersDesc` from the `SuppliersViewAccessor` view accessor.

   The editor does not allow you to remove the default attribute mapping for the attribute for which the list is defined.
8. If you want to specify supplemental values that your list returns to the base view object, click the **Create return attribute map** button in the **List Return Values**

section and map the desired base view object attributes with attributes accessed by the view accessor.

Supplemental attribute return values are useful when you do not require the user to make a list selection for the attributes, yet you want those values, as determined by the current row, to participate in the update. For example, to map the attribute `SupplierAddress` from the `PurchaseOrdersView` view object, you would choose the attribute `SupplierAddress` from the `SuppliersViewAccessor` view accessor.

9. Click **OK**.

How to Set User Interface Hints on a View Object LOV-Enabled Attribute

When you know how the view object attribute that you define as an LOV should appear in the user interface, you can specify additional properties of the LOV to determine its display characteristics. These properties, or **UI hints**, augment the attribute hint properties that ADF Business Components lets you set on any view object attribute. Among the LOV UI hints for the LOV-enabled attribute is the type of component the user interface will use to display the list. For a description of the available components, see Table 5-2. (Not all ADF Faces components support the default list types, as noted in the Table 5-2.)

Table 5-2   List Component Types for List Type UI Hint

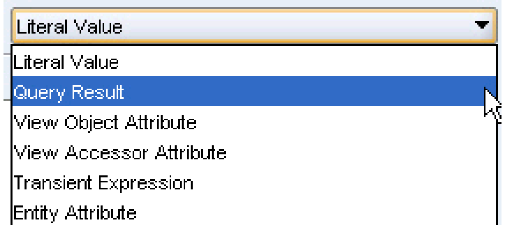
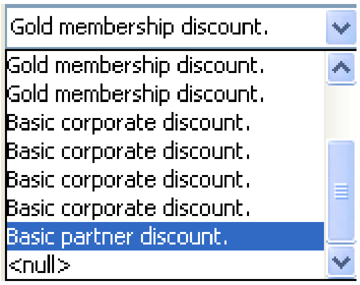
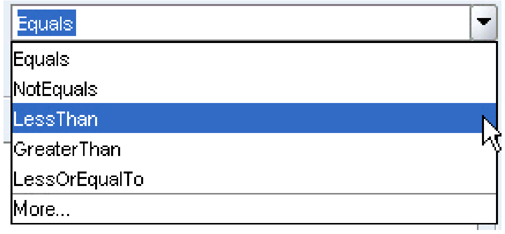
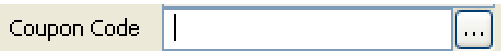
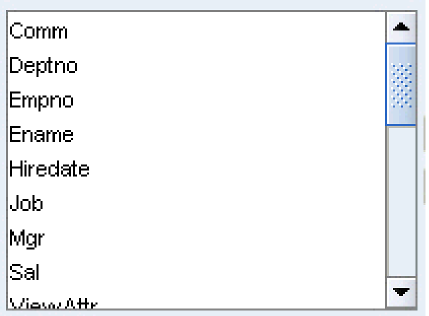
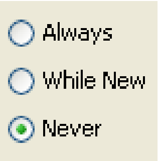
LOV List Component Type	Usage
Choice List  Literal Value Literal Value Query Result View Object Attribute View Accessor Attribute Transient Expression Entity Attribute	This component does not allow the user to type in text, only select from the dropdown list.
Combo Box  Gold membership discount. Gold membership discount. Gold membership discount. Basic corporate discount. Basic corporate discount. Basic corporate discount. Basic corporate discount. Basic partner discount. <null>	This component allows the user to type text or select from the dropdown list. This component sometimes supports auto-complete as the user types. This component is *not* supported for ADF Faces.

Table 5-2 (Cont.) List Component Types for List Type UI Hint

LOV List Component Type	Usage
Combo Box with List of Values  Equals Equals NotEquals LessThan GreaterThan LessOrEqualTo More...	This component is the same the as the combo box, except that the last entry (More...) opens a List of Values lookup dialog that supports query with filtering when enabled for the LOV attribute in its UI hints. The default UI hint enables queries on all attributes. Note that when the LOV attribute appears in a table component, the list type changes to an **Input Text with List of Values** component.
Input Text with List of Values 	This component displays an input text field with an LOV button next to it. The List of Values lookup dialog opens when the user clicks the button or enters an invalid value into the text field. The List of Values lookup dialog for this component supports query with filtering when enabled in the UI hints for the LOV attribute. The default UI hint enables queries on all attributes. This component may also support auto-complete when a unique match exists.
List Box  Comm Deptno Empno Ename Hiredate Job Mgr Sal ViewAttr	This component takes up a fixed amount of real estate on the screen and is scrollable (as opposed to the choice list, which takes up a single line until the user clicks on it).
Radio Group 	This component displays a radio button group with the selection choices determined by the LOV attribute values. This component is most useful for very short, fixed lists.

Before you begin:

It may be helpful to have an understanding of LOV-enabled attributes. See Working with List of Values (LOV) in View Object Attributes.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. See Additional Functionality for View Objects.

You will need to complete this task:

Create the LOV list for the attribute, as described in How to Define a Single LOV-Enabled View Object Attribute.

To set view object attribute UI hints for an LOV-enabled attribute:

1. In the Applications window, double-click the view object that contains the attribute that you want to customize.
2. In the overview editor, click the **Attributes** navigation tab.
3. In the Attributes page, select the desired attribute and then click the **List of Values** tab.
4. In the List of Values page, select the LOV list that you want to customize and click the **Edit list of values** button.
5. In the Edit List of Values dialog, click the **UI Hints** tab.
6. In the UI Hints page, select a default list type as the type of component to display the list.

   For a description of the available components, see Table 5-2.

   The list component displayed by the web page and the view object's default list type must match at runtime or a method-not-found runtime exception results. To avoid this error, confirm the desired list component with the user interface designer. You can also edit the default list type to match, so that, should the user interface designer subsequently change the component used in the web page, the two stay in sync.
7. Optionally, select additional display attributes to add values to the display.

   The list of additional attributes is derived from the LOV-enabled attribute's view row. The additional attribute values can help the end user select an item from the list.
8. If you selected the **Combo Box with List of Values** type component, by default, the dropdown list for the component will display the first 10 records from the data source. This limit also serves to keep the view object fetch size small. To change the number of records the dropdown list of a Combo Box with List of Values component can display, enter the number of records for **Query Limit**.

   When you want the user to be able to view the full set of records, do not use **Query Limit** for this purpose. Because **Query Limit** also controls the number of rows the view object will fetch (it sets the view object definition `ListRangeSize` property), the value should be keep small for optimal performance. The end user who needs access to the full set of records should click on the component's lookup icon to open an LOV lookup dialog and view the records there.

   Query Limit is disabled for all other component types and those components place no restriction on the number of rows that the LOV will access.

   For details about the `ListRangeSize` property, see What Happens at Runtime: How an LOV Queries the List Data Source.

9. If you selected a component type that allows the user to open a List of Values lookup dialog to select a list value (this includes either the **Combo Box with List of Values** type component or **Input Text with List of Values** type component), by default, the lookup dialog will display a search form that will allow the user to search on all queryable attributes of the data source view object (the one defined by the LOV-enabled attribute's view accessor). Decide how you want to customize these components.

   a. When you select the **Combo Box with List of Values** type component and you have added a large number of attributes to the **Selected** list, use **Show in Combo Box** to improve the readability of the dropdown list portion of the component. To limit the attribute columns to display in the dropdown list that the Combo Box with List of Values component displays, choose **First** from **Show in Combo Box** and enter a number corresponding to the number of attributes from the top of the **Selected** list that you want the dropdown list to display (this combination means you are specifying the "first" *x* number of attributes to display from the Create List of Values dialog's **Selected** list). Limiting the number of attribute columns to display in the dropdown list ensures that the user does not have to horizontally scroll to view the full list, but it does not limit the number of attribute columns to display in the List of Values lookup dialog. This option is disabled for all list component types except Combo Box with List of Values.

   b. You can limit the attributes to display in the List of Values lookup dialog by selecting a view criteria from the **Include Search Region** dropdown list. To appear in the dropdown list, the view criteria must already have been defined on the data source view object (the one that the LOV-enabled attribute's view accessor defines). Click the **Edit View Criteria** button to set search form properties for the selected view criteria. For more information about customizing view criteria for search forms, see How to Set User Interface Hints on View Criteria to Support Search Forms.

   c. You can prepopulate the results table of the List of Values lookup dialog by selecting **Query List Automatically**. The List of Values lookup dialog will display the results of the query when the user opens the dialog. If you leave this option deselected, no results will be displayed until the user submits the search form.

10. Alternatively, if you prefer not to display a search region in the List of Values lookup dialog, select **<No Search>** from the **Include Search Region** dropdown list. In this case, the List of Values lookup dialog will display only attributes you add to the **Display Attributes** list.

11. If you selected a choice type component to display the list, you can specify a **Most Recently Used Count** as an alternative to displaying all possible values.

    For example, your form might display a choice list of `SupplierId` values to drive a purchase order form. In this case, you can allow the user to select from a list of their most recently viewed suppliers, where the number of supplier choices is determined by the count you enter. The default count `0` (zero) for the choice list displays all values for the attribute.

12. If you selected a **Combo Box with List of Values** type component to display the list, you can select a view criteria from the **Filter Combo Box Using** dropdown list to limit the list of valid values the LOV will display.

    When you enable **Filter Combo Box Using**, the dropdown list displays the existing view criteria from the view object definition related to the LOV's view accessor. If the dropdown list displays no view criteria, then the data source view object defines no view criteria. When you do not enable this feature, the Combo Box with List of Values component derives its values from the full row set returned by the view accessor. The filtered Combo Box with List of Values is a useful feature when you want to support the use of an LOV with popup search dialog or LOV with a dropdown list that has a limited set of valid choices. For details about using the Combo Box with List of Values component in user interfaces, see List of Values (LOV) Input Fields.

13. Decide how you want the list component to handle a null value choice to display in the list component. This option is not enabled for every list component type that you can select.

    If you enable **Include "No Selection" Item**, you can also determine how the null value selection should appear in the list by making a selection from the dropdown list. For example, when you select **Labeled Item**, you can enter the desired label in the edit field to the right of the dropdown list or you can click the **...** button (to the right of the edit field) to select a message string from the resource bundle associated with the view object. When you select a message string from the resource bundle, JDeveloper saves the string's corresponding message key in the view object definition file. At runtime, the UI locates the string to display based on the current user's locale setting and the message key in the localized resource bundle.

14. Click **OK**.

How to Handle Date Conversion for List Type UI Components

When the LOV-enabled attribute of the view object is bound to date information (such as the attribute `OrderShippedDate`), by default Oracle ADF assumes a format for the field like `yyyy-MM-dd hh:mm:ss`, which combines date and time. This combined date-time format is specified by the ADF Business Components Date domain class (`jbo.domain.Date`) and creates a conversion issue for the ADF Faces component when the user selects a date supplied by the LOV-enable attribute. When the ADF Faces component is unable to convert the domain type to the Date type, the user interface invalidates the input field and displays the message `Error: The date is not in the correct format`.

To avoid this potential conversion error, configure a UI hint setting for the date value attribute of the view object that you want to enable for an LOV. The UI hint you specify will define a date-only mask, such as `yyyy-MM-dd`. Subsequently, any ADF Faces component that references the attribute will perform the conversion based on a pattern specified by its EL value-binding expression (such as `#{bindings.Hiredate.format}`) and will reference the UI hint format instead of the ADF Business Components domain date-time. The conversion error happens when the EL expression evaluates to null because no format mask has been specified.

Before you begin:

It may be helpful to have an understanding of LOV-enabled attributes. For more information, see Working with List of Values (LOV) in View Object Attributes.

You may also find it helpful to understand support for UI hints at the level of view objects. For more information about UI hints, see Defining UI Hints for View Objects.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for View Objects.

You will need to complete this task:

Create the LOV list for the attribute, as described in How to Define a Single LOV-Enabled View Object Attribute.

To set a UI hint to match the date format for the LOV-enable attribute:

1. In the Applications window, double-click the view object that contains the LOV-enabled attribute.
2. In the overview editor, click the **Attributes** navigation tab.
3. In the Attributes page, select the date-value attribute that you want to customize with UI hints, and then click the **UI Hints** tab.
4. In the UI Hints page, select **Simple Date** for the **Format Type** and choose the format with the date-only mask.

   Mapping of the ADF Business Components domain type to its available formatters is provided in the `formatinfo.xml` file in the BC4J subdirectory of the JDeveloper system directory (for example, `C:\Documents and Settings\<username>\Application Data\JDeveloper\system<version#>\o.BC4J.\formatinfo.xml`).

How to Automatically Refresh the View Object of the View Accessor

If you need to ensure that your view accessor always queries the latest data from the database table, you can set the **Auto Refresh** property on the data source view object. This property allows the view object instance to refresh itself after a change in the database. You can enable this feature for any read-only view instance that your application modules define. Once you enable this property on a view object, it ensures that the changes a user commits to a database table will become available to any other user working with the same database table. A typical use case is to enable auto refresh for the data source view object when you define a view accessor for an LOV-enabled view object attribute.

Because the auto-refresh feature relies on Oracle Database change notification feature, observe these restrictions when enabling auto-refresh for your view object:

- Ensure the view objects query only read-only tables, and as few tables as possible. This will ensure the best performance and prevent the database invalidation queue from becoming too large.
- Configure the application module that contains updatable, auto-refresh view instances to be shared and to lock rows during updates.
- Ensure the database user has database notification privileges. For example, to accomplish this with a SQL*Plus command use `grant change notification to <user name>`.
- Test the query for compatibility with query result change notification before you enable auto refresh on the view object. Not all query definitions satisfy the requirements for query result change notification.

When you enable auto refresh for the view object and observe these restrictions, the refresh is accomplished through the Oracle database change notification feature. At runtime, prior to executing the view object query, the framework will use the JDBC API to register the view object query to receive database change notifications for underlying data changes. When the view object receives a notification (because its underlying data has changed), the row sets of the view object are marked as dirty and the framework will refresh the row set on the next server trip from the client to the middle tier. At that point, the dirty collections will be discarded and the request for the updated data will trigger a new query execution by the view object. Because the application module waits until the next checkout, the row set currency of the current transaction is maintained and the end user is not hampered by the update.

For example, assume that a user can create or edit a calendar entry but cannot edit calendar entries added by other users. When the user creates and commits a new entry, then in the same server trip the calendar entries that other users modified or entered will be updated. But when another user creates a calendar entry, the view object receives a notification and waits for the next server trip before it refreshes itself; the delay to perform the update prevents contention among various users to read the same data.

In another example, say an LOV displays a list of postal codes that is managed in read-only fashion by a database administrator. After the administrator adds a new postal code as a row to the database, the application module detects a time when there are no outstanding requests and determines that a pending notification exists for the view instance that accesses the list of postal codes; at that point, the view object refreshes the data and all future requests will see the new postal code.

> **Note:**
>
> Use optimistic row locking for web applications. Optimistic locking, the default configuration setting, assumes that multiple transactions can complete without affecting each other. Optimistic locking therefore allows auto-refresh to proceed without locking the rows being refreshed. Pessimistic row locking prevents the row set refresh and causes the framework to throw an exception anytime the row set has a transaction pending (for example, a user may be in the process of adding a new row). To ensure that the application module configuration uses optimistic row locking, open the Business Components page of the `adf-config.xml` overview editor and confirm that **Locking Mode** (corresponding to the `jbo.locking.mode` property) is set to `optimistic`, as described in How to Confirm That Fusion Web Applications Use Optimistic Locking.

Before you begin:

It may be helpful to have an understanding of LOV-enabled attributes. See Working with List of Values (LOV) in View Object Attributes.

It may be helpful to understand the query definition requirements for registering a query for query result change notification. See Using Continuous Query Notification in the *Oracle Database Advanced Application Developer's Guide*.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for View Objects.

You will also need to complete this task:

- Create the view accessor, as described in How to Create a View Accessor for an Entity Object or View Object.

For information about queries see Using Continuous Query Notification in the *Oracle Database Advanced Application Developer's Guide*.

To register a view object to receive data change notifications:

1. In the Applications window, double-click the view object that you want to receive database change notifications.
2. In the overview editor, click the **Query** navigation tab.
3. In the Query page, click **Test and Explain**.
4. In the Test Query dialog, click the **Change Notification** tab to validate the query is compatible with Oracle Database change notification.

   If the Test Query dialog shows the query does not pass change notification compatibility, you can not enable auto refresh on the view object. Not all query definitions satisfy the requirements for query result change notification.
5. If the query is compatible with change notification, in the overview editor, click the **General** navigation tab.
6. In the Properties window, expand the **Tuning** section, and select **True** from the **Auto Refresh** dropdown list.

   If the **Tuning** section is not displayed in the Properties window, click the **General** navigation tab in the overview editor to set the proper focus.

How to Test LOV-Enabled Attributes Using the Oracle ADF Model Tester

To test the LOV you created for a view object attribute, use the Oracle ADF Model Tester, which is accessible from the Applications window.

The Oracle ADF Model Tester, for any view object instance that you browse, will display any LOV-enabled attributes using one of two component types you can select in the UI Hints page of the List of Values dialog. Currently, only a Choice List component type and Input Text with List of Values component type are supported. Otherwise, the Oracle ADF Model Tester uses the default choice list type to display the LOV-enabled attribute.

Before you begin:

It may be helpful to have an understanding of LOV-enabled attributes. For more information, see Working with List of Values (LOV) in View Object Attributes.

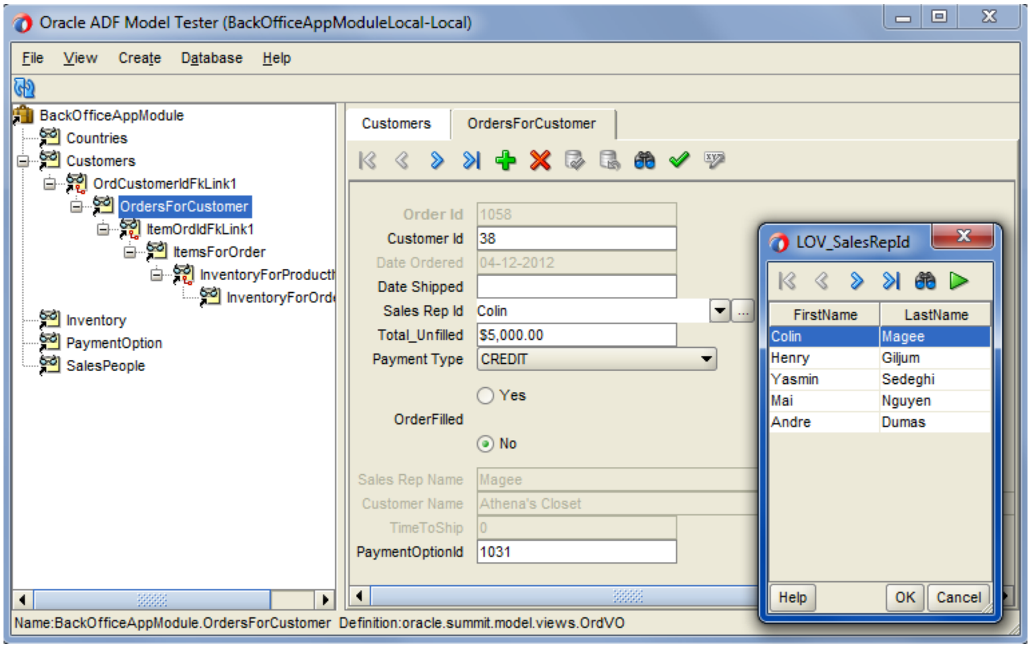
You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for View Objects.

To test an LOV using the Oracle ADF Model Tester:

1. In the Applications window, expand the project containing the desired application module and view objects.
2. Right-click the application module and choose **Run**.
3. In the Oracle ADF Model Tester, select the desired view object from the section on the left. The Oracle ADF Model Tester displays the LOV-enabled attribute values in a dropdown list unless you specified the component type as an Input Text with List of Value, UI hint.

Figure 5-38 shows an LOV-enabled attribute, `SalesRepId` for the `CustomerVO`, that specifies an input text field and List of Values dialog as the UI hint list type. The Input Text with List of Values component is useful when you want to display the choices in a separate LOV dialog. Other list types are not supported by the Oracle ADF Model Tester.

Figure 5-38 Displaying LOV-Enabled Attributes in the Oracle ADF Model Tester



Order Id	1058
Customer Id	38
Date Ordered	04-12-2012
Date Shipped	
Sales Rep Id	Colin
Total_Unfilled	$5,000.00
Payment Type	CREDIT
OrderFilled	○ Yes ● No
Sales Rep Name	Magee
Customer Name	Athena's Closet
TimeToShip	0
PaymentOptionId	1031

FirstName	LastName
Colin	Magee
Henry	Giljum
Yasmin	Sedeghi
Mai	Nguyen
Andre	Dumas

Name:BackOfficeAppModule.OrdersForCustomer Definition:oracle.summit.model.views.OrdVO

What Happens When You Define an LOV for a View Object Attribute

When you define an LOV for a view object attribute, the view object metadata defines the following additional information, as shown in the following example for the `CustomerVO.SalesRepId` attribute.

- The `<ViewAttribute>` element names the attribute, points to the list binding element that defines the LOV behavior, and specifies the component type to display in the web page. For example, the LOV-enabled attribute `SalesRepId` points to the list binding named `LOV_SalesRepId` and defines the `CONTROLTYPE` input text field with list (`input_text_lov`) to display the LOV data.

  When the user interface designer creates the web page using the Data Controls panel, the `<CONTROLTYPE Value="`*`namedType`*`"/>` definition determines the component that JDeveloper will add to the web page. When the component type definition in the data model project does not match the component type displayed in the web page, a runtime exception will result. For more information, see What Happens at Runtime: How an LOV Queries the List Data Source.

- The `<ListBinding>` element defines the behavior of the LOV. It also identifies a view accessor to access the data source for the LOV-enabled attribute. The view accessor is the ADF Business Components mechanism that lets you obtain the full list of possible values from the row set of the data source view object. For example, `ListVOName="EmpVO1"` points to the `EmpVO1` view accessor, which accesses the view object `EmpVO`.

- The <ListBinding> element maps the list data source attribute to the LOV-enabled attribute. For example, the ListAttrNames item Id is mapped to the LOV-enabled attribute SalesRepId.
- The <ListBinding> element also identifies one or more attributes to display from the current row and provides a few options that are specific to the choice list type component. For example, the ListDisplayAttrNames item FirstName and LastName are extra attributes displayed by the LOV-enabled attribute SalesRepId. In this example, the value none for NullValueFlag means the user cannot select a blank item from the list.

```
<ViewAttribute
    Name="SalesRepId"
    LOVName="LOV_SalesRepId"
    PrecisionRule="true"
    EntityAttrName="SalesRepId"
    EntityUsage="CustomerEO"
    AliasName="SALES_REP_ID">
    <Properties>
      <SchemaBasedProperties>
        <LABEL
          ResId="oracle.summit.model.views.CustomerVO.SalesRepId_LABEL"/>
        <CONTROLTYPE
          Value="input_text_lov"/>
      </SchemaBasedProperties>
    </Properties>
</ViewAttribute>
. . .
<ListBinding
    Name="LOV_SalesRepId"
    ListVOName="EmpVO1"
    ListRangeSize="10"
    NullValueFlag="none"
    MRUCount="0">
    <AttrArray Name="AttrNames">
      <Item Value="SalesRepId"/>
    </AttrArray>
    <AttrArray Name="ListAttrNames">
      <Item Value="Id"/>
    </AttrArray>
    <AttrArray Name="ListDisplayAttrNames">
      <Item Value="Id"/>
      <Item Value="FirstName"/>
      <Item Value="LastName"/>
    </AttrArray>
    <DisplayCriteria
      Hint="hide"/>
</ListBinding>
. . .
<ViewAccessor
    Name="EmpVO1"
    ViewObjectName="oracle.summit.model.views.EmpVO"
    OrderBy="EmpEO.LAST_NAME">
    <ViewCriteriaUsage
      Name="FilterByTitleIdVC"
      FullName="oracle.summit.model.views.EmpVO.FilterByTitleIdVC"/>
    <ParameterMap>
      <PIMap Variable="TitleIdBind">
        <TransientExpression><![CDATA[2]]></TransientExpression>
      </PIMap>
    </ParameterMap>
  </ViewAccessor>
```

What Happens at Runtime: How an LOV Queries the List Data Source

The ADF Business Components runtime adds view accessors in the attribute setters of the view row and entity object to facilitate the LOV-enabled attribute behavior. In order to display the LOV-enabled attribute values in the user interface, the LOV facility fetches the data source, and finds the relevant row attributes and mapped target attributes. The databound list component's default `AutoSubmit` property setting of `false` ensures the browser makes a server roundtrip only when the end user selects a value. As a result of the roundtrip, all list bindings in the **ADF Model layer** that are derived from the same LOV-enabled attribute get updated and the end-user selection is reflected in the corresponding list components of the user interface.

The number of data objects that the LOV facility fetches is determined in part by the `ListRangeSize` setting in the LOV-enabled attribute's list binding definition, which is specified in the Edit List of Values dialog that you display on the attribute from the view object overview editor. By default, the number of fetched records for LOV queries is not truncated for all types of list components (including LOV and non-LOV type list components). The default value `-1` specified by `ListRangeSize` means that the user will be able to view all the data objects from the data source. However, if the number of records fetched is very large and you want to truncate the values available to the list component used to display the records, then on `ListRangeSize` you can specify a discrete value for the number of records to fetch. The `ListRangeSize` value has no effect on the records that the end user can search on in the lookup dialog displayed for the two LOV type components. For more information about how each list component displays values, see How to Set User Interface Hints on a View Object LOV-Enabled Attribute.

Note that although you can alter the `ListRangeSize` value in the metadata definition for the `<ListBinding>` element, setting the value to a discrete number of records (for example, `ListRangeSize="5"`) most likely will not provide the user with the desired selection choices. Instead, if the value is `-1` (default for list components), then no restrictions are made to the number of records the list component will display, and the user will have access to the full set of values.

Performance Tip:

To limit the set of values an LOV displays, use a view accessor to filter the LOV binding, as described in How to Define a Single LOV-Enabled View Object Attribute. Additionally, in the case of component types that display a choice list, you can change the **Most Recently Used Count** setting to limit the list to display the user's previous selections, as described in How to Set User Interface Hints on a View Object LOV-Enabled Attribute.

Note, a runtime exception will occur when a web page displays a UI component for an LOV-enabled attribute that does not match the view object's `CONTROLTYPE` definition. When the user interface designer creates the page in JDeveloper using the Data Controls panel, JDeveloper automatically inserts the list component identified by the **Default List Type** selection you made for the view object's LOV-enabled attribute in the List UI Hint dialog. However, if the user interface designer changes the list type subsequent to creating the web page, you will need to edit the selection in the List UI Hint dialog to match.

What You May Need to Know About Lists

There are several things you may need to know about LOVs that you define for attributes of view objects, including how to propagate LOV-enabled attributes from parent view objects to child view objects (by extending an existing view object) and when to use validators instead of an LOV to manage a list of values.

Inheritance of AttributeDef Properties from Parent View Object Attributes

When a view object extends another view object, you can create the LOV-enabled attribute on the base object. Then when you define the child view object in the overview editor, the LOV definition will be visible on the corresponding view object attribute. This inheritance mechanism allows you to define an LOV-enabled attribute once and later apply it across multiple view objects instances for the same attribute.

You can also use the overview editor to extend the inherited LOV definition. For example, you may add extra attributes already defined by the base view object's query to display in selection list. Alternatively, you can define a view object that uses a custom `WHERE` clause to query the supplemental attributes not already queried by the based view object. For information about customizing entity-based view objects, see Working with Bind Variables.

Using Validators to Validate Attribute Values

If you have created an LOV-enabled attribute for a view object, there is no need to validate the attribute using a List Validator. You only use an attribute validator when you do not want the list to display in the user interface, but still need to restrict the list of valid values. List validation may be a simple static data or it may be a list of possible values obtained through a view accessor you define. Alternatively, you might prefer to use Key Exists validation when the attribute displayed in the UI is one that references a key value (such as a primary, foreign, or alternate key). For information about declarative validation in ADF Business Components, see Defining Validation and Business Rules Declaratively.

Defining UI Hints for View Objects

You define UI Hints on an ADF view object attributes to display context-sensitive UI information to the end user.

One of the built-in features of ADF Business Components is the ability to define UI hints on view objects and attributes of view objects. UI hints are settings that the view layer can use to automatically display the queried information to the user in a consistent, locale-sensitive way. For example, in web pages, a UI developer may access UI hint values by entering EL expressions utility methods defined on the `bindings` name space and specified for **ADF binding** instance names.

JDeveloper stores the hints in resource bundle files that you can easily localize for multilingual applications.

How to Add Attribute-Specific UI Hints

To create UI hints for attributes of a view object, use the overview editor for the view object, which is accessible from the Applications window. You can also display and edit UI hints using the Properties window that you display for an attribute.

Before you begin:

It may be helpful to have an understanding of attribute UI hints. For more information, see Defining UI Hints for View Objects.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for View Objects.

You will need to complete this task:

> Create the desired view objects, as described in How to Create an Entity-Based View Object, and How to Create a Custom SQL Mode View Object.

To customize view object attribute with UI hints:

1. In the Applications window, double-click the view object that you want to customize with UI hints.
2. In the overview editor, click the **Attributes** navigation tab.
3. In the Attributes page, select the attribute that you want to customize with UI hints, and then click the **UI Hints** tab and define the desired hints.

   For example, for an attribute `UserId`, you might enter a value for its **Label Text** hint like "`Id`" or set the **Format Type** to Number, and enter a **Format** mask of `00000`.

> **Note:**
>
> Java defines a standard set of format masks for numbers and dates that are different from those used by the Oracle database's SQL and PL/SQL languages. For reference, see the Javadoc for the `java.text.DecimalFormat` and `java.text.SimpleDateFormat` classes.

How to Add View Object UI Hints

To create UI hints for a view object, use the overview editor for the view object, which is accessible from the Applications window. You can also display and edit several additional UI hints using the Properties window that you display for the view object.

Before you begin:

It may be helpful to have an understanding of UI hints. For more information, see Defining UI Hints for View Objects.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for View Objects.

You will need to complete this task:

> Create the desired view objects, as described in How to Create an Entity-Based View Object, and How to Create a Custom SQL Mode View Object.

To customize view objects with UI hints:

1. In the Applications window, double-click the view object that you want to customize with UI hints.
2. In the overview editor, click the **General** navigation tab.
3. In the General page, enter a **Display Name** to define an EL accessible hint for the view object name.

   For example, for a view object `OrdersVO`, you might enter a value for its **Display Name** hint like "`Order`".
4. With the General page displayed in the overview editor, open the Properties window for the view object and expand the **UI Hints** section, and then enter additional hints as needed.

   For example, for a view object `OrdersVO`, you might enter a value for the **Display Name (Plural)** hint like "`Orders`" and, for the **Description** hint, you might enter a value like "`customer orders`".

How to Access UI Hints Using EL Expressions

A UI developer can access UI hints using EL expressions and display the hint values as data in a web page. The UI developer may access UI hints through the ADF binding instances that they create after dropping databound components into their web pages.

In the case of the view object hints, the UI developer accesses the view object hints through the collection binding defined for the view object. For example, assume that you have configured the view object UI hints as follows.

- `OrdersVO` view object **Display Name** hint = `Order`
- `OrdersVO` view object **Display Name (Plural)** hint = `Orders`
- `OrdersVO` view object **Description** hint = `customer orders`

The UI developer might display a header that makes use of these hints like this:

`Showing` *`customer orders`* `number 10 of 51` *`Orders`*`.`

The following example shows that the EL expression that produces the above text. In this EL expression the collection binding `OrdersVO1` provides access to the view object hints. The names of the EL expression utility methods match the property names defined in the view object XML definition file for the UI hints. For example, the view object property name `labelPlural`, which defines the **Display Name (Plural)** hint, corresponds to the utility method name used in the expression `bindings.OrdersVO1.hints.labelPlural`.

```
<af:panelHeader id="ph1"
           text="Showing #{bindings.OrdersVO1.hints.description} number
                         #{bindings.OrdersVO1.Orderno.inputValue} of
                         #{bindings.OrdersVO1.estimatedRowCount}
                         #{bindings.OrdersVO1.hints.labelPlural}.">
```

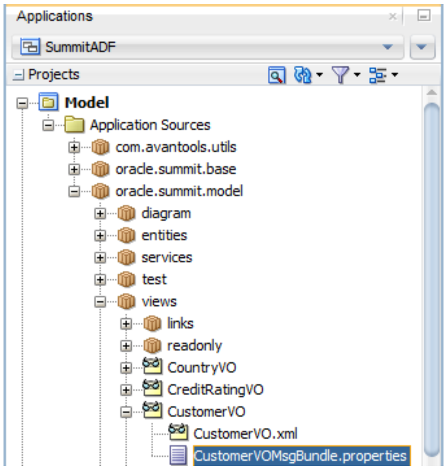
What Happens When You Add UI Hints

When you define attribute UI hints for a view object or view object attributes, by default JDeveloper stores them in a resource bundle file. This allows the hints that you define to be localized for the generated forms and tables of the user interface.

The type of resource bundle file that JDeveloper uses and the granularity of the file are determined by settings on the Resource Bundle page of the Project Properties dialog. By default, JDeveloper sets the option to **Properties Bundle** and generates one `.properties` file for the entire data model project. For example, when you define UI hints for a view object in the `Model` project, JDeveloper creates the bundle file named `ModelBundle.properties` for the package.

Alternatively, if you select the option in the Project Properties dialog to generate one resource bundle per file, you can inspect the message bundle file for any view object by selecting the object in the Applications window and looking under the corresponding **Sources** node in the Structure window. The Structure window shows the implementation files for the component you select in the Applications window. You can inspect the resource bundle file for the view object by expanding the parent package of the view object in the Applications window, as shown in Figure 5-39.

Figure 5-39 Resource Bundle File in Applications Window



For more information on the resource bundle options you can select, see How to Set Message Bundle Options.

The following example shows a sample message bundle file where the UI hint information appears. The first entry in each `String` array is a message key; the second entry is the locale-specific `String` value corresponding to that key.

```
oracle.summit.model.views.CustomerVO.Id_FMT_FORMATTER=
                                       oracle.jbo.format.DefaultNumberFormatter
oracle.summit.model.views.CustomerVO.Id_FMT_FORMAT=00000
oracle.summit.model.views.CustomerVO.Id_LABEL=Id
oracle.summit.model.views.CustomerVO.Email_LABEL=Email Address
oracle.summit.model.views.CustomerVO.LastName_LABEL=Surname
oracle.summit.model.views.CustomerVO.FirstName_LABEL=Given Name
```

How to Define UI Category Hints

UI categories provide the means to group attributes that a view object defines. The category names that you create are identifiers to be used by the dynamic rendering user interface to group attributes for display. The user interface will render the attribute with other attributes of the same category. You can use the category hint to aid the user interface to separate a large list of view object attributes into smaller groups related by categories.

Additionally, you can specify the field order hint to reorder how the user interface will render the attribute values within its category. For example, if a view object defines four attributes `attributeA`, `attributeB`, `attributeC`, and `attributeD` and you specify the field order 4, 3, 2, and 1 respectively for each attribute, then wherever the user interface renders the category, the attributes of that category will appear in the order attributeD, attributeC, attributeB, and attributeA.

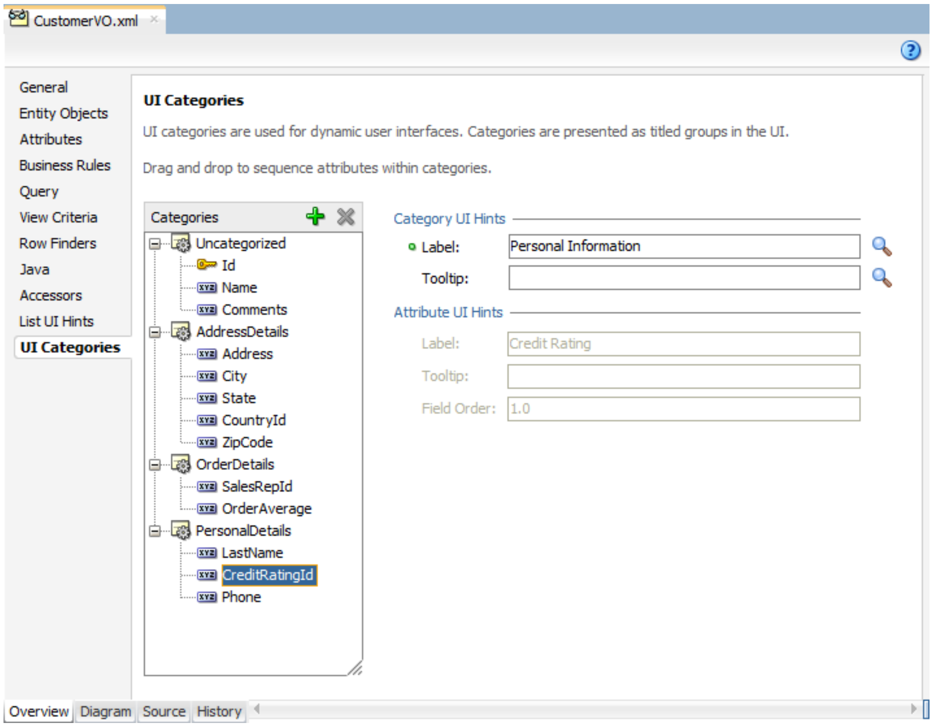
> **Note:**
>
> Use the UI Categories page in the overview editor for the view object to change the order of the attributes listed within a category you've created. JDeveloper automatically assigns and maintains the field order values of the attributes based on their order in the list, and you do not need to edit numeric values to define the field order hint.

The category and field order hints will be utilized by any dynamic rendering user interface that displays the attribute, including dynamic forms and search forms:

- In the case of dynamic forms, the attributes from each category will appear in a separate tab.
- In the case of search forms, the order of the form's individual view criteria is determined by the field order and category assigned to the attribute upon which the view criteria items are based.

To create UI categories for attributes of a view object, use the overview editor for the view object, which is accessible from the Applications window. You can create and edit categories for the entire view object using the UI Categories page, as shown in Figure 5-40.

Figure 5-40 Attribute UI Categories in View Object Overview Editor



UI Categories

UI categories are used for dynamic user interfaces. Categories are presented as titled groups in the UI.

Drag and drop to sequence attributes within categories.

Category UI Hints

Label: Personal Information

Tooltip:

Attribute UI Hints

Label: Credit Rating

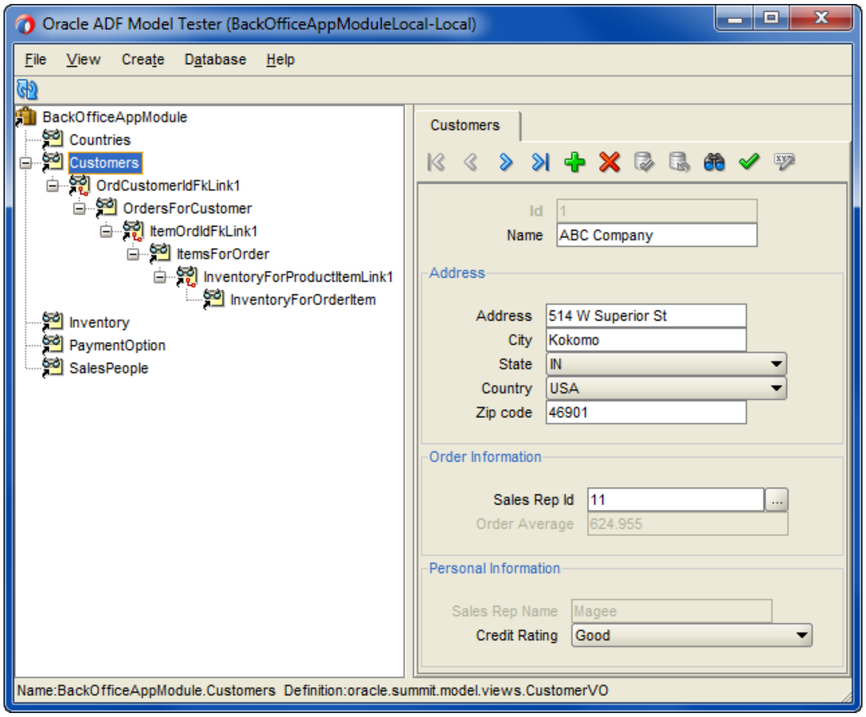
Tooltip:

Field Order: 1.0

When you assign a view object attribute to a category that you create in the UI Categories page, the order of the attributes displayed in the category list determines its numeric field order. The UI Categories page lets you change the field order of the attributes you've assigned to a category by dragging and dropping attributes within the list. When you drag and drop the attributes into the category lists, JDeveloper automatically maintains the correct sequence of the field order hints within the category and displays the field order value of individual attributes in the **Attribute UI Hints** list of the UI Categories page. Other attribute-level UI hints displayed in the UI Categories page are synchronized with settings in the **UI Hints** tab of the Attributes page of the view object editor.

Each category can have a label and tooltip text string resource to be utilized by the user interface when the category is rendered. You can localize these resource strings in the resource bundle file that you select to store the entries. For example, Figure 5-41 shows the attributes `LastName` and `CreditRatingId` with labels **Sales Rep Name** and **Credit Rating** in a category with the label **Personal Information**.

Figure 5-41 UI Categories Displayed in Oracle ADF Model Tester



Id 1
Name ABC Company
Address 514 W Superior St
City Kokomo
State IN
Country USA
Zip code 46901
Sales Rep Id 11
Order Average 624.955
Sales Rep Name Magee
Credit Rating Good

Name:BackOfficeAppModule.Customers Definition:oracle.summit.model.views.CustomerVO

Before you begin:

It may be helpful to have an understanding of UI hints. For more information, see Defining UI Hints for View Objects.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for View Objects.

You will need to complete these tasks:

- Create the desired view objects, as described in How to Create an Entity-Based View Object, and How to Create a Custom SQL Mode View Object.
- Optionally, specify attribute UI hints, as described in How to Add Attribute-Specific UI Hints. Attribute UI hints are displayed read-only in the UI Categories page of the view object editor.

To create user interface categories and reorder the attributes for display:

1. In the Applications window, double-click the view object that you want to customize with user interface categories.
2. In the overview editor, click the **UI Categories** navigation tab.
3. In the UI Categories page, click the **Create New Category** button to create the category and then right-click the new category and choose **Rename**.
4. In the Rename dialog, enter the name of the category.
5. In the overview editor, in the **UI Hints** section, enter the user interface **Label Text** and **Tooltip Text** for the category.

The text that you enter will be added to the default resource bundle file for the project, as described in What Happens When You Add UI Hints. To select a different resource bundle file to store the label and tooltip strings, click the browse (**. . .**) button beside the text field.

6. In the **Categories** list, expand the **Uncategorized** list and scroll the list of attributes to locate the attribute you want to add to a category.

   The **Uncategorized** list displays all the attributes that the view object you are editing defines. This list is displayed by the overview editor as a selection list for the creation of UI Category hints. Attributes that you leave in the **Uncategorized** list will appear by default above the attribute categories displayed in the user interface.

7. Select the desired attribute and drag it into the new category you created. If the category contains more than one attribute, the position of the attribute that you drop into the list will determine its field order value.

   Developer automatically assigns a numeric value to the field order hint based on the sequence of the attributes that appear within a UI Category list. The attribute you place at the top of a category list will be rendered by the user interface first, the attribute second in the list will be rendered second, and so on. The field order hint appears in the **Attribute UI Hints** list of the UI Categories page and is not editable. The hint value is also visible in the source for the view object definition and should not be edited in the source.

What Happens When You Assign Attributes to UI Categories

When you define attribute UI categories for a view object, JDeveloper updates the view object's XML document file. JDeveloper adds the `CATEGORY` and the `FIELDORDER` UI hints in the `<SchemaBasedProperties>` element of the `<ViewAttribute>` element. The definition of the categories appears in a new `<Category>` element.

The metadata in the following example shows that the `CustomerId` attribute's `CATEGORY` hint refers to the `AccountInformation` category and the `FirstName` attribute's `CATEGORY` hint refers to the `UserInformation` category. The definition for both categories appears in `<Category>` elements. The `FIELDORDER` hint for each attribute specifies a numeric value, which JDeveloper assigns and maintains based on the order of the attributes in the UI categories lists you create in the overview editor. As shown in the following example, the `FIELDORDER` hint is a decimal value. A decimal value is used by JDeveloper to allow you to insert new attributes into a category without requiring JDeveloper to change all the existing attribute values and still be able to maintain the correct order.

Note that the default field order value for the first attribute in a category is assigned by JDeveloper as `0.0`. The field order value can be changed to any number to sort the category list, and it is not an index. The field order numeric values do not need to be contiguous.

```
<Category
    Name="AccountInformation">
    <Properties>
      <SchemaBasedProperties>
        <LABEL
          ResId="CUSTOMER_DETAILS"/>
        <TOOLTIP
          ResId="AccountInformation_TOOLTIP"/>
      </SchemaBasedProperties>
    </Properties>
</Category>
<Category
    Name="UserInformation">
    <Properties>
      <SchemaBasedProperties>
        <TOOLTIP
          ResId="UserInformation_TOOLTIP"/>
        <LABEL
          ResId="CUSTOMER_DETAILS"/>
      </SchemaBasedProperties>
    </Properties>
</Category>
...
<ViewAttribute
    Name="CustomerId"
    IsNotNull="true"
    PrecisionRule="true"
    EntityAttrName="CustomerId"
    EntityUsage="CustomerEO"
    AliasName="CUSTOMER_ID">
    <Data>
      <Property
        Name="OWNER_SCOPE"
        Value="INSTANCE"/>
      ...
    </Data>
    <Properties>
      <SchemaBasedProperties>
        <CATEGORY
          Value="AccountInformation"/>
        <FIELDORDER
          Value="0.0"/>
      </SchemaBasedProperties>
    </Properties>
  </ViewAttribute>
...
<ViewAttribute
    Name="FirstName"
    PrecisionRule="true"
    EntityAttrName="FirstName"
    EntityUsage="CustomerEO"
    AliasName="FIRST_NAME">
    <Data>
      <Property
        Name="OWNER_SCOPE"
        Value="INSTANCE"/>
      ...
    </Data>
    <Properties>
      <SchemaBasedProperties>
        <CATEGORY
          Value="UserInformation"/>
        <FIELDORDER
          Value="0.0"/>
      </SchemaBasedProperties>
    </Properties>
</ViewAttribute>
```

What You May Need to Know About Resource Bundles

Internationalizing the model layer of an application built using ADF Business Components entails producing translated versions of each component's resource bundle file. For example, the Italian version of the `QueryDataWithViewObjectsBundle.properties` file would be a file named `QueryDataWithViewObjectsBundle_it.properties`, and a more specific *Swiss* Italian version would have the name `QueryDataWithViewObjectsBundle_it_ch.properties`.

Resource bundle files contain entries for the message keys that need to be localized, together with their localized translation. For example, assuming you didn't want to translate the number format mask for the Italian locale, the Italian version of the `QueryDataWithViewoObjects` view object message keys would look like what you see in the following example. At runtime, the resource bundles are used automatically, based on the current user's locale settings.

```
oracle.summit.model.views.CustomerVO.Id_FMT_FORMATTER=

oracle.jbo.format.DefaultNumberFormatter
oracle.summit.model.views.CustomerVO.Id_FMT_FORMAT=00000
oracle.summit.model.views.CustomerVO.Id_LABEL=Codice Utente
oracle.summit.model.views.CustomerVO.Email_LABEL=Indirizzo Email
oracle.summit.model.views.CustomerVO.LastName_LABEL=Cognome
oracle.summit.model.views.CustomerVO.FirstName_LABEL=Nome
```

The resource bundles of the model project contribute one aspect of internationalizing the Fusion web application. Internationalization for specific locales also requires localizing the ADF Faces pages of the user interface. For more information about how to localize ADF Faces pages, see the "Internationalizing and Localizing Pages" chapter in *Developing Web User Interfaces with Oracle ADF Faces*.

6

Defining Master-Detail Related View Objects

This chapter describes how to link ADF view objects to other view objects to form master-detail hierarchies of any complexity in an Oracle ADF application.
This chapter includes the following sections:

About Master-Detail View Objects

You can create a link between an ADF view object to a single view object or multiple view objects to create master-detail hierarchies.

A **master-detail relationship** is established when a view link is created to associate two view object instances. A **view link** represents the relationship between two view objects, which is usually, but not necessarily, based on a foreign-key relationship between the underlying data tables. The view link associates a row of one view object instance (the master object) with one or more rows of another view object instance (the detail object).

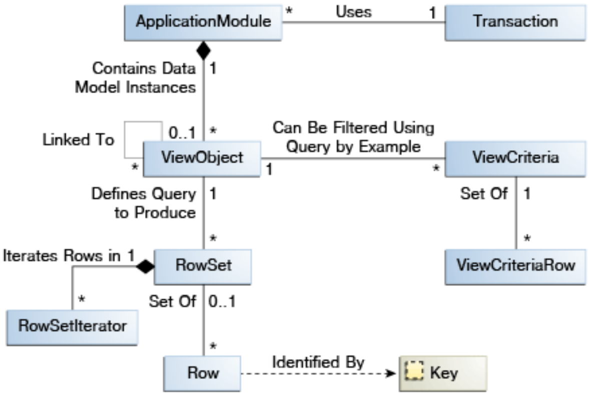
Master-Detail View Object Use Cases and Examples

This chapter helps you understand these **view object** concepts as illustrated in Figure 6-1:

- You can link a view object to one or more others to create master-detail hierarchies of any complexity.
- You can arrange parent-child view object instances in the context of an **application module** such that at runtime the detail view instance is actively coordinated with the current row of the master view instance.
- At design time, you can define a **view link accessor** that specifies an accessor attribute that the master collection uses at runtime to return the detail collection row set.
- You can programmatically access the detail collection row set using the view link accessor attribute that you specify for the master view object in the master-detail relationship.
- You can tune master-detail collection queries, by caching the row set of the detail view instance and support UI components, like the **ADF Faces** tree component, where data for each master node in a tree needs to retain its distinct set of detail rows.

- You can create row finder conditions using **view criteria** to locate specific rows within the master view object.
- You can define a self-referential master and detail hierarchy such that the source and destination view link attributes derive from a single table.
- You can define an inline view criteria to filter the row set of the master view object using attribute values specified by attributes of the detail view object.

Figure 6-1 A View Object Defines a Query and Produces a Row Set of Rows



Additional Functionality for View Objects

You may find it helpful to understand other **Oracle ADF** features before you start working with view objects. Following are links to other functionality that may be of interest.

- For details about using the interactive Oracle ADF Model Tester to validate view object query results, see Testing View Instance Queries.
- For details about creating a data model consisting of view object instances, see Implementing Business Services with Application Modules.
- For a quick reference to the most common code that you will typically write, use, and override in your custom view object classes, see Most Commonly Used ADF Business Components Methods.
- For API documentation related to the `oracle.jbo` package, see the following Javadoc reference document:
  - *Java API Reference for Oracle ADF Model*

Working with Multiple Tables in a Master-Detail Hierarchy

JDeveloper allows you to create source and target view objects in a master-detail hierarchy when there are many queries involving multiple tables related by foreign keys.

Many queries you will work with will involve multiple tables that are related by foreign keys. In this scenario, you can create separate view objects that query the related information and link a "source" view object to one or more "target" view objects to form a master-detail hierarchy.

There are two ways you might handle this situation. You can either:

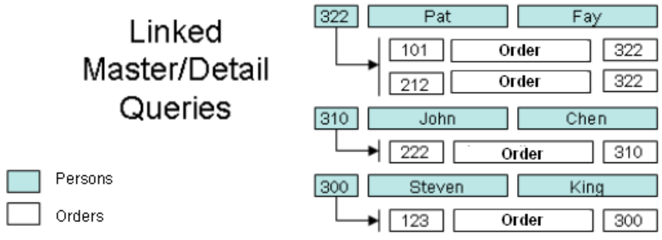
- Create a view link based on an association between **entity objects** when the source and target view objects are based on the underlying entity objects' association.
- Create a view link that defines how the source and target view objects relate.

Whether or not an association exists is determined when entity objects are created. By default, the entity object associations model the hierarchical relationships of the data source. For example, entity objects based on database tables related by foreign keys will capture these relationships in **entity associations**. If you do base the view link on an existing entity association, there is no performance penalty over defining the view link on the view objects alone. In either case, you use the Create View Link wizard to define the master-detail relationship.

> **Note:**
>
> A view link defines a basic master-detail relationship between two view objects. However, by creating more view links you can achieve master-detail hierarchies of any complexity, including:
>
> - Multilevel master-detail-detail
> - Master with multiple (peer) details
> - Detail with multiple masters
>
> The steps to define these more complex hierarchies are the same whether you create the relationships based on view objects alone or view objects with entity associations. In either case, you just need to create each additional hierarchy, one view link at time.

Figure 6-2 illustrates the multilevel result that master-detail linked queries produce.

Figure 6-2 Linked Master-Detail Queries



How to Create a Master-Detail Hierarchy Based on Entity Associations

When you want to show the user a set of master rows, and for each master row a set of coordinated detail rows, then you can create view links to define how you want the master and detail view objects to relate. For example, you could link the `DeptVO` view object to the `EmpVO` view object to create a master-detail hierarchy of departments and the set of employees they define. When these view objects are backed by entity objects related by an existing association, you can select the association to define the master-detail view link.

To create an association-based view link, you use the Create View Link wizard.

Before you begin:

It may be helpful to have an understanding of the ways to create a master-detail hierarchy. For more information, see Working with Multiple Tables in a Master-Detail Hierarchy.

You may also find it helpful to understand how entity associations are created. For more information, see What Happens When You Create Entity Objects and Associations from Existing Tables.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for View Objects.

You will need to complete this task:

> Create the desired entity-based view objects, as described in How to Create an Entity-Based View Object.

To create an association-based view link

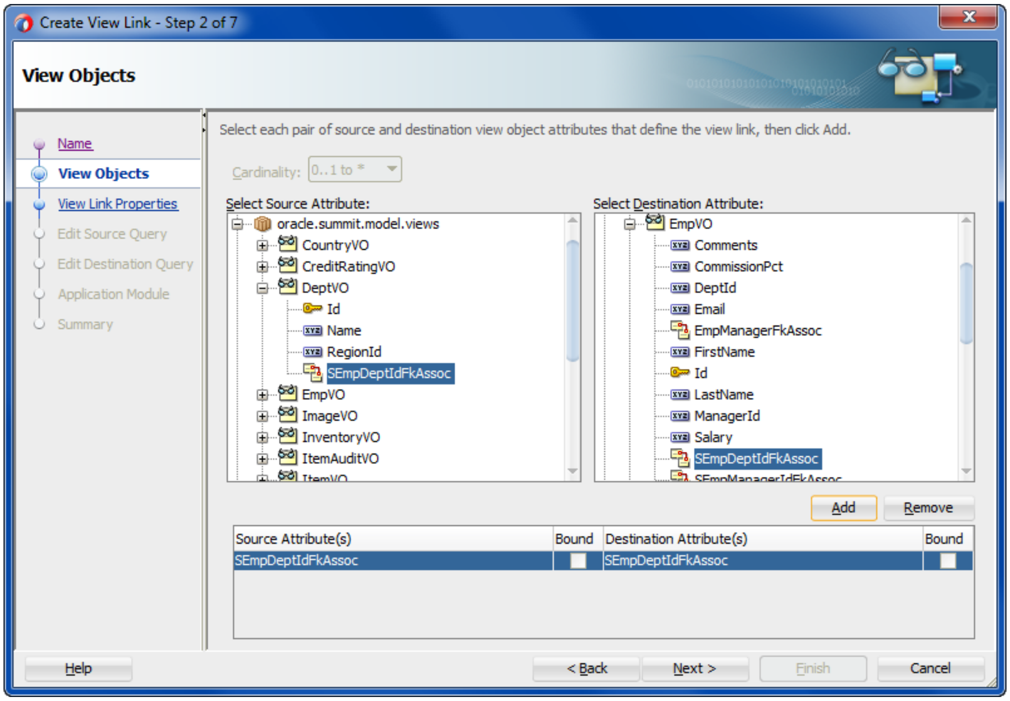
1. In the Applications window, right-click the package node of the project in which you want to create the view object and choose **New** and then **View Link**.
2. In the Create View Link wizard, on the Name page, supply a package and a component name.

   For example, given the purpose of the view link, a name like `DeptToEmpFkLink` is a valid name to identify the link between a department list and its employees when the link is based on a foreign key.
3. On the View Objects page, in the **Select Source Attribute** tree expand the source view object in the desired package. In the **Select Destination Attribute** tree, expand the destination view object.

   For entity-based view objects, notice that in addition to the view object attributes, relevant associations also appear in the list.
4. Select the same association in both **Source** and **Destination** trees. Then click **Add** to add the association to the table below.

   For example, Figure 6-3 shows the same `SEmpDeptIdFkAssoc` association in both **Source** and **Destination** trees selected.

Figure 6-3 Master and Detail Related by an Association Selection



Create View Link - Step 2 of 7

View Objects

Select each pair of source and destination view object attributes that define the view link, then click Add.

Cardinality: 0..1 to *

Source Attribute(s)	Bound	Destination Attribute(s)	Bound
SEmpDeptIdFkAssoc	□	SEmpDeptIdFkAssoc	□

5. Click **Finish**.

How to Create a Master-Detail Hierarchy Based on View Objects Alone

Just as with association-based view links, you can directly link view objects to other view objects to form master-detail hierarchies of any complexity. The only difference in the creation steps, involves the case when both the master and detail view objects are not related by an existing association. In this situation, because there is no association to capture the set of source and destination attribute pairs that relate them, you create the view link by indicating which view object attributes it should be based on.

To create the view link, use the Create View Link wizard.

Before you begin:

It may be helpful to have an understanding of the ways to create a master-detail hierarchy. For more information, see Working with Multiple Tables in a Master-Detail Hierarchy.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for View Objects.

You will need to complete this task:

Create the desired read-only view objects, as described in How to Create a Custom SQL Mode View Object.

To create a view link between view objects:

1. In the Applications window, right-click the package node of the project in which you want to create the view object and choose **New** and then **View Link**.
2. In the Create View Link wizard, on the Name page, enter a package name and a view link name. Click **Next**.

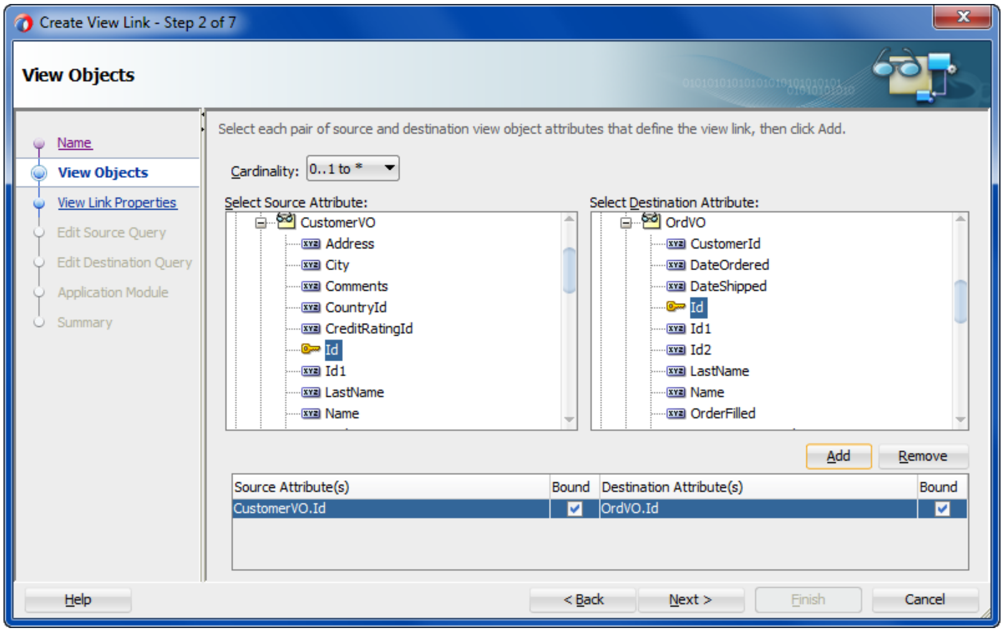
   For example, given the purpose of the view link, a name like `OrdersPlacedBy` is a valid name to identify the link between a customer list and their orders.
3. On the View Objects page, select a "source" attribute from the view object that will act as the master.

   For example, Figure 6-4 shows the `Id` attribute selected from the `CustomerVO` view object to perform this role. Click **Next**.
4. On the View Objects page, select a corresponding destination attribute from the view object that will act as the detail.

   For example, if you want the detail query to show orders that were placed by the currently selected customer, select the `Id` attribute in the `OrdersVO` to perform this role.
5. Click **Add** to add the matching attribute pair to the table of source and destination attribute pairs below. When you are finished defining master and detail link, click **Next**.

   Figure 6-4 shows just one (`CustomerVO.Id`, `OrderVO.Id`) pair. However, if you require multiple attribute pairs to define the link between master and detail, repeat the steps for the View Objects page to add additional source-target attribute pairs.

Figure 6-4 Defining Source/Target Attribute Pairs While Creating a View Link



Create View Link - Step 2 of 7

View Objects

Select each pair of source and destination view object attributes that define the view link, then click Add.

Cardinality: 0..1 to *

Source Attribute(s)	Bound	Destination Attribute(s)	Bound
CustomerVO.Id	☑	OrdVO.Id	☑

6. On the View Link Properties page, you can use the **Accessor Name** field to change the default name of the accessor that lets you programmatically access the destination view object.

   By default, the accessor name will match the name of the destination view object. For example, you might change the default accessor name `OrdersVO1` to

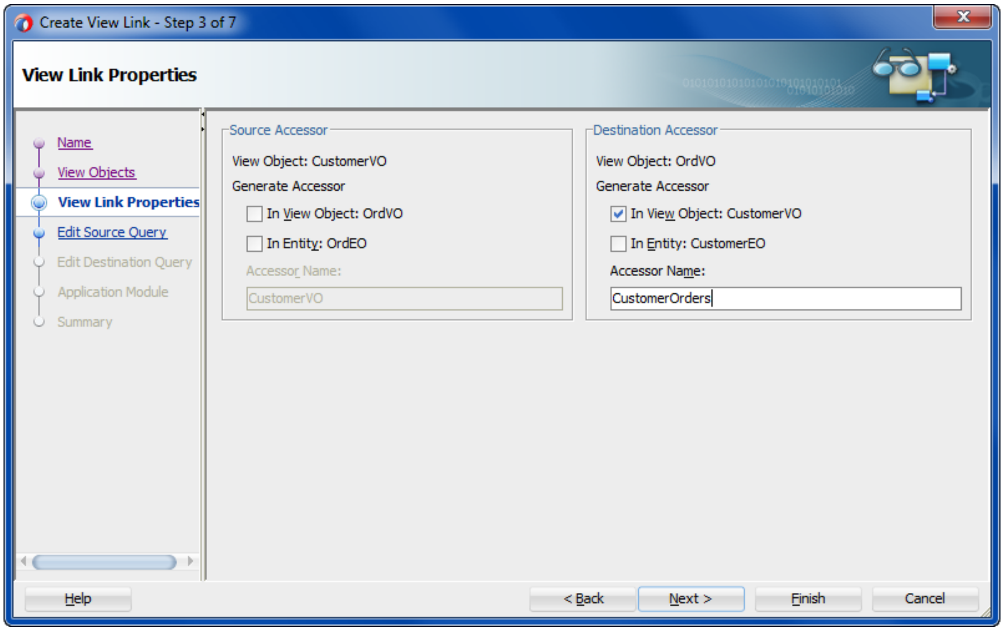
`CustomerOrders` to better describe the master-detail relationship that the accessor defines.

7. Also on the View Link Properties page, you control whether the view link represents a one-way relationship or a bidirectional one.

   By default, a view link is a one-way relationship that allows the current row of the source (master) to access a set of related rows in the destination (detail) view object. Leave the default settings unchanged for **Generate Accessor** when you want to enable support for hierarchical UI components, such as databound trees which require a view link accessor with accessor generated in the master view object.

   For example, in Figure 6-5, the checkbox settings indicate that you'll be able to access a detail collection of rows from `OrdersVO` for the current row in `CustomerVO`, but not vice versa. In this case, this behavior is specified by the checkbox setting in the **Destination Accessor** group box for `OrdersVO` (the **Generate Accessor In View Object: CustomerVO** box is selected) and checkbox setting in the **Source Accessor** group box for `CustomerVO` (the **Generate Accessor In View Object: OrdersVO** box is not selected).

Figure 6-5 View Link Properties Control Name and Direction of Accessors



Source Accessor
View Object: CustomerVO
Generate Accessor
□ In View Object: OrdVO
□ In Entity: OrdEO
Accessor Name:
CustomerVO

Destination Accessor
View Object: OrdVO
Generate Accessor
☑ In View Object: CustomerVO
□ In Entity: CustomerEO
Accessor Name:
CustomerOrders

8. On the Edit Source Query page, preview the view link SQL predicate that will be used at runtime to access the master row in the source view object and click **Next**.

9. On the Edit Destination Query page, preview the view link SQL predicate that will be used at runtime to access the correlated detail rows from the destination view object for the current row in the source view object and click **Next**.

10. On the Application Module page, add the view link to the data model for the desired application module and click **Finish**.

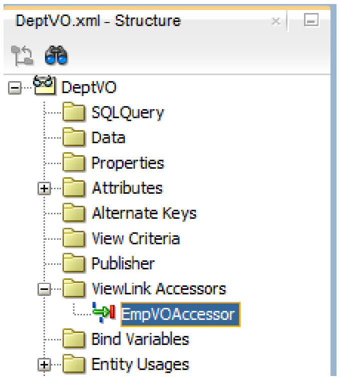
    By default the view link will not be added to the application module's data model. Later you can add the view link to the data model using the overview editor for the application module.

What Happens When You Create Master-Detail Hierarchies Using View Links

When you create a view link or an association-based view link, JDeveloper creates the XML document file that represents its declarative settings and saves it in the directory that corresponds to the name of its package. For example, if the view link is named `DeptToEmpFkLink` and it appears in the `views.links` package, then the XML file created will be `./views/links/DeptToEmpFkLink.xml` under the project's source path. This XML file contains the declarative information about the source and target attribute pairs you've specified and, in the case of an association-based view link, contains the declarative information about the association that relates the source and target view objects you've specified.

In addition to saving the view link component definition itself, JDeveloper also updates the XML definition of the *source* view object in the view link relationship to add information about the view link accessors that you may have defined. As a confirmation of this, you can select the source view object in the Applications window and inspect its details in the Structure window. As shown in Figure 6-6, you can see the default destination accessor `EmpVOAccessor` under the **ViewLink Accessors** node for the `DeptVO` source view object that accesses the destination `EmpVO` view object. This allows the source view object, using the view link accessor, to traverse the destination side of the view link. In addition to the view link accessor definition for the source view object, you can directly customize the view link to define a view link accessor on the destination view object, which provides access to row set of the source view object.

Figure 6-6 View Object with View Link Accessor in the Structure Window



How to Enable Active Master-Detail Coordination in the Data Model

When you enable programmatic navigation to a row set of correlated details by defining a view link, as described in How to Create a Master-Detail Hierarchy Based on Entity Associations, the view link plays a *passive* role, simply defining the information necessary to retrieve the coordinated detail row set when your code requests it. The view link accessor attribute is present and programmatically accessible in any result rows from any instance of the view link's source view object. In other words, programmatic access does not require modifying the application module's data model.

However, since master-detail user interfaces are such a frequent occurrence in enterprise applications, the view link can be also used in a more *active* fashion so you can avoid needing to coordinate master-detail screen programmatically. You opt to have this active master-detail coordination performed by *explicitly* adding an instance of a view-linked view object to your application module's data model.

To enable active master-detail coordination, open the application module in the overview editor and select the Data Model page.

Before you begin:

It may be helpful to have an understanding of the ways to create a master-detail hierarchy. For more information, see Working with Multiple Tables in a Master-Detail Hierarchy.

You may also find it helpful to understand the role of the application module. For more information, see About Application Modules.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for View Objects.

You will need to complete these tasks:

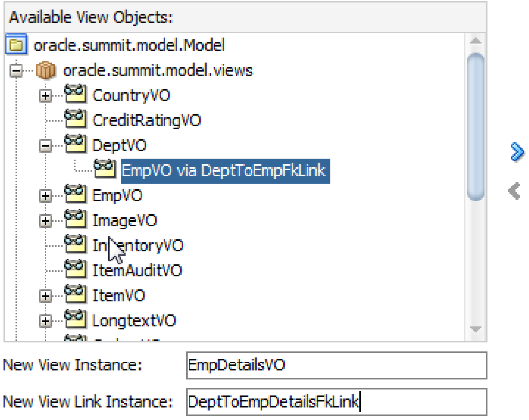
- Create the desired view objects, as described in How to Create an Entity-Based View Object and How to Create a Custom SQL Mode View Object.
- Create an application module, as described in How to Create an Application Module.

To add a detail instance of a view object:

1. In the Applications window, double-click the application module for which you define a new view instance.
2. In the overview editor, click the **Data Model** navigation tab.
3. In the Data Model page, expand the **View Object Instances** section and, in the **Available View Objects** list, select the detail view object node that is indented beneath the master view object.

   Note that the list shows the detail view object *twice*: once on its own, and once as a detail view object via the view link. For example, in Figure 6-7 you would select the detail view object `EmpVO via DeptToEmpFkLink` instead of the view object labeled as `EmpVO` (which, in this case, appears beneath the highlighted view object).

Figure 6-7 Detail View Object Selection from Available View Objects



New View Instance: EmpDetailsVO

New View Link Instance: DeptToEmpDetailsFkLink

4. In the **Name View Instance** field below the **Available View Objects** list, enter a name for the detail instance you're about to create.

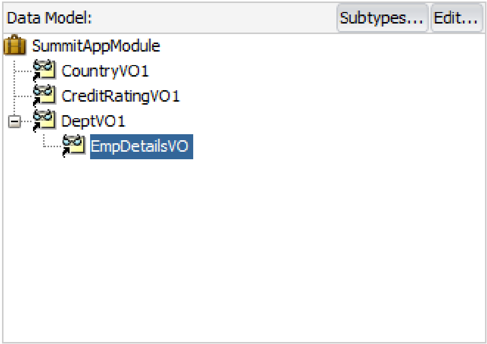
   For example, Figure 6-7 shows the name `EmpDetailsVO` for the instance of the `EmpVO` view object that is a detail view.

5. In the **Data Model** list, select the instance of the view object that you want to be the actively coordinating master.

6. Click **Add Instance** to add the detail instance to the currently selected master instance in the data model, with the name you've chosen.

   For example, in Figure 6-8, the **Data Model** list shows a master-detail hierarchy of view object instances with `EmpDetailsVO` as the detail view object.

Figure 6-8 Data Model with View Linked View Object



How to Test Master-Detail Coordination

To test active master-detail coordination, launch the Oracle ADF Model Tester on the application module by choosing **Run** from its context menu in the Applications window. The Oracle ADF Model Tester data model tree shows the view link instance that is actively coordinating the detail view object instance with the master view object instance. You can double-click the view link instance node in the tree to open a master-detail data view page in the Oracle ADF Model Tester. Then, when you use the toolbar buttons to navigate in the master view object — changing the view object's current row as a result — the coordinated set of details is automatically refreshed and the user interface stays in sync.

If you double-click another view object that is not defined as a master and detail, a second tab will open to show its data; in that case, since it is not actively coordinated by a view link, its query is not constrained by the current row in the master view object.

For information about editing the data model and running the Oracle ADF Model Tester, see Testing View Object Instances Using the Oracle ADF Model Tester.

How to Access the Detail Collection Using the View Link Accessor

To work with view links effectively, you should also understand that view link accessor attributes return a RowSet object and that you can access a detail collection using the view link accessor programmatically.

Accessing Attributes of Row by Name

At runtime, the getAttribute() method on a Row object allows you to access the value of any attribute of that row in the view object's result set by name. The view link accessor behaves like an additional attribute in the current row of the source view object, so you can use the same getAttribute() method to retrieve its value. The only practical difference between a regular view attribute and a view link accessor attribute is its data type. Whereas a regular view attribute typically has a scalar data type with a value like 303 or ngreenbe, the value of a view link accessor attribute is a row set of zero or more correlated detail rows. Assuming that curUser is a Row object from some instance of the Orders view object, you can write a line of code to retrieve the detail row set of order items:

```
RowSet items = (RowSet)curUser.getAttribute("OrderItems");
```

Note:

If you generate the custom Java class for your view row, the type of the view link accessor will be RowIterator. Since at runtime the return value will always be a RowSet object, it is safe to cast the view link attribute value to RowSet.

Programmatically Accessing a Detail Collection Using the View Link Accessor

Once you have retrieved the RowSet object of detail rows using a view link accessor, you can loop over the rows it contains just as you would loop over a view object's row set of results, as shown in the following example.

```
while (items.hasNext()) {
  Row curItem = items.next();
  System.out.println("--> (" + curItem.getAttribute("LineItemId") + ") " +
                     curItem.getAttribute("LineItemTotal"));
}
```

For information about creating a test client, see How to Access a Detail Collection Using the View Link Accessor.

Optimizing View Link Accessor Access to Display Master-Detail Data

You can enable caching of the view link accessor row set when you do not want the application to incur the small amount of overhead associated with reexecuting queries to create new detail row sets. For example, because **view accessor** row sets remain stable as long as the master row view accessor attribute remains unchanged, it would not be necessary to re-create a new row set for UI components, like the ADF Faces tree component, where data for each master node in a tree needs to retain its distinct set of detail rows.

You can enable retention of the view link accessor row set using the overview editor for the view object that is the source for the view link accessor. Select the **Retain Row Set** flag in the View Accessors page of the overview editor for the source view object.

> **Performance Tip:**
>
> When you expect user interface developers will use the view link to create databound master-detail components where the detail collections are stable, such as an ADF Faces tree component, enable the **Retain Row Set** flag. At runtime, this setting ensures the accessors for each detail collection will be executed once. The compromise for this improvement in performance is that accessed collections may occupy more memory space as compared to accessing the detail collections without the flag enabled.

To enable retention of the view link accessor row set:

1. In the Applications window, double-click the source view object that defines the view link.
2. In the overview editor, click the **View Accessors** navigation tab.
3. In the View Accessors page, expand the **View Link Accessors** section and select the **Retain Row Set** checkbox.

What You May Need to Know About View Link Accessors Versus Data Model View Link Instances

View objects support two different styles of master-detail coordination:

- View link *instances* for data model master-detail coordination, as described in Enabling a Dynamic Detail Row Set with Active Master-Detail Coordination.
- View link *accessor* attributes for programmatically accessing detail row sets on demand, as described in Programmatically Accessing a Detail Row Set Using View Link Accessor Attributes.
- You can combine both styles, as described in What Happens At Runtime: When You Combine the Programmatic Access Using View LInk Accessors.

Enabling a Dynamic Detail Row Set with Active Master-Detail Coordination

When you add a view link instance to your application module's data model, you connect two specific view object instances. The use of the view link instance indicates that you want active master-detail coordination between the two. At runtime the view link instance in the data model facilitates the eventing that enables this coordination. Whenever the current row is changed on the master view object instance, an event causes the detail view object to be refreshed by automatically invoking `executeQuery()` with a new set of bind parameters for the new current row in the master view object.

A key feature of this data model master-detail is that the master and detail view object instances are stable objects to which client user interfaces can establish bindings. When the current row changes in the master — instead of producing a *new* detail view object instance — the existing detail view object instance updates its default row set to contain the set of rows related to the new current master row. In addition, the user interface binding objects receive events that allow the display to update to show the detail view object's refreshed row set.

Another key feature that is exclusive to data model hierarchy is that a detail view object instance can have multiple master view object instances. For example, an `PaymentOptions` view object instance may be a detail of *both* a `Customers` and a `Orders` view object instance. Whenever the current row in *either* the `Customers` or `Orders` view object instance changes, the default row set of the detail `PaymentOptions` view object instance is refreshed to include the row of payment information for the current customer and the current order. See How to Create a Master-Detail Hierarchy With Multiple Masters for details on setting up a detail view object instance with multiple-masters.

Programmatically Accessing a Detail Row Set Using View Link Accessor Attributes

When you need to programmatically access the detail row set related to a view object row by virtue of a view link, you can use the **view link accessor** attribute. You specify the finder name of the view link accessor attribute from the overview editor for the view link. Click the **Edit** icon in the Accessors section of the Relationship page and in the Edit View Link Properties dialog, edit the name of the view link accessor attribute.

The following example shows the XML for the view link that defines the `_findername` value of the `<Attr>` element.

```
<ViewLinkDefEnd
    Name="Orders"
    Cardinality="1"
    Owner="devguide.advanced.multiplemasters.Orders"
    Source="true">
    <AttrArray Name="Attributes">
      <Item Value="devguide.advanced.multiplemasters.Orders.PaymentOptionId"/>
    </AttrArray>
    <DesignTime>
      <Attr Name="_minCardinality" Value="1"/>
      <Attr Name="_isUpdateable" Value="true"/>
      <Attr Name="_finderName" Value="Orders"/>
    </DesignTime>
</ViewLinkDefEnd>
```

Assuming you've named your accessor attribute *`AccessorAttrName`*, you can access the detail row set using the generic `getAttribute()` API like:

```
RowSet detail = (RowSet)currentRow.getAttribute("AccessorAttrName");
```

If you've generated a custom view row class for the master view object and exposed the getter method for the view link accessor attribute on the client view row interface, you can write strongly typed code to access the detail row set like this:

```
RowSet detail = (RowSet)currentRow.getAccessorAttrName();
```

Unlike the data model master-detail, programmatic access of view link accessor attributes does not require a detail view object instance in the application module's data model. Each time you invoke the view link accessor attribute, it returns a `RowSet` containing the set of detail rows related to the master row on which you invoke it.

Using the view link accessor attribute, the detail data rows are stable. As long as the attribute value(s) involved in the view link definition in the master row remain unchanged, the detail data rows will not change. Changing of the current row in the master does not affect the detail row set which is "attached" to a given master row. For this reason, in addition to being useful for general programmatic access of detail rows, view link accessor attributes are appropriate for UI objects like the tree control, where data for each master node in a tree needs to retain its distinct set of detail rows.

What Happens At Runtime: When You Combine the Programmatic Access Using View LInk Accessors

Oracle ADF needs to distinguish between the data model master-detail and view link accessor row sets. When you *combine* the use of data model master-detail with programmatic access of detail row sets using view link accessor, it is important to understand that they are distinct mechanisms. For example, imagine that you:

- Define `PersonsVO` and `OrdersVO` view objects
- Define a view link between them, naming the view link accessor `PersonsToOrders`
- Add instances of them to an application module's data model named `master` (of type `PersonsVO`) and `detail` (of type `OrdersVO`) coordinated actively by a view link instance.

If you find a person in the `master` view object instance, the `detail` view object instance updates as expected to show the corresponding orders. At this point, if you invoke a custom method that programmatically accesses the `PersonsToOrders` view link accessor attribute of the current `PersonsVO` row, you get a `RowSet` containing the set of `OrdersVO` rows. You might reasonably expect this programmatically accessed `RowSet` to have come from the `detail` view object instance in the data model, but this is not the case.

The `RowSet` returned by a view link accessor always originates from an *internally created* view object instance, not one you that added to the data model. This internal view object instance is created as needed and added with a system-defined name to the **root application module**.

The principal reason a distinct, internally created view object instance is used is to guarantee that it remains unaffected by developer-related changes to their own view objects instances in the data model. For example, if the view row were to use the detail view object in the data model for view link accessor `RowSet`, the resulting row set could be inadvertently affected when the developer dynamically:

1. Adds a `WHERE` clause with new named bind parameters

   If such a view object instance were used for the view link accessor result, unexpected results or an error could ensue because the dynamically added `WHERE` clause bind parameter values have not been supplied for the view link accessor's `RowSet`: they were only supplied for the default row set of the detail view object instance in the data model.

2. Adds an additional master view object instance for the detail view object instance in the data model.

   In this scenario, the semantics of the accessor would be changed. Instead of the accessor returning `OrdersVO` rows for the current `PersonsVO` row, it could all of a sudden start returning *only* the `OrdersVO` rows for the current `PersonsVO` that were created by a current logged in customer, for example.

3. Removes the detail view object instance or its containing **application module instance**.

   In this scenario, all rows in the programmatically accessed detail `RowSet` would become invalid.

Furthermore, Oracle ADF needs to distinguish between the data model master-detail and view link accessor row sets for certain operations. For example, when you create a new row in a detail view object, the framework automatically populates the attributes involved in the view link with corresponding values of the master. In the data model master-detail case, it gets these values from the current row(s) of the possibly multiple master view object instances in the data model. In the case of creating a new row in a `RowSet` returned by a view link accessor, it populates these values from the master row on which the accessor was called.

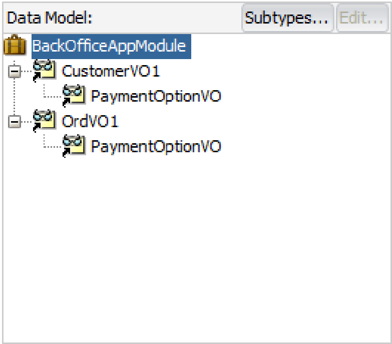
How to Create a Master-Detail Hierarchy With Multiple Masters

When useful, you can set up your data model to have multiple master view object instances for the same detail view object instance. Consider view objects named `Customers`, `Orders`, and `PaymentOptions` with view links defined between:

- `Customers` and `PaymentOptions`
- `Orders` and `PaymentOptions`

Figure 6-9 shows what the data model panel looks like when you've configured both `Customers` and `Orders` view object instances to be masters of the same `PaymentOptions` view object instance.

Figure 6-9 Multiple Master View Object Instances for the Same Detail



To set up the data model as shown in Figure 6-9 you use the overview editor for the application module and use the Data Model page.

Before you begin:

It may be helpful to have an understanding of view objects. For more information, see About View Objects.

To create a view instance with multiple master view objects:

1. Add an instance of the first master view object to the data model.

   Assume you name it `Customers`.
2. Add an instance of the second master view object to the data model.

   Assume you name it `Orders`.
3. Select the first master view object instance in the **Data Model** list
4. In the **Available View Objects** list, select the detail view object indented beneath the first master view object and enter the view object instance name in the **New Instance Name** field. Click **>** to shuttle it into data model as a detail of the existing `Customers` view object instance.
5. Select the second master view object instance in the **Data Model** list
6. In the **Available View Objects** list, select the detail view object indented beneath the second master view object and the **New Instance Name** field, enter the same view object instance name as the previously entered detail view object. Click **>** to shuttle it into data model as a detail of the existing `Orders` view object instance.

   An alert will appear: **An instance of a View Object with the name PaymentOptionsVO has already been used in the data model. Would you like to use the same instance?**
7. Click **Yes** to confirm you want the detail view object instance to *also* be the detail of the second view object instance.

How to Create a Master-Detail Hierarchy for Entity Objects Consisting of Transient-Only Attributes

When you link entity-based view objects to form master-detail hierarchies, the view objects and their respective entity usages are typically related by an association. At runtime, the association constructs an internal association view object that enables a query to be executed to enable the master-detail coordination. An exception to this scenario occurs when the entity object that participates in an association consists exclusively of nonpersistent attributes. This may occur when you define an entity object with transient attributes that you wish to populate programmatically at runtime. In this case, you can use the association overview editor that links the nonpersistent entity object to select an association view object to perform the query.

Before you begin:

It may be helpful to have an understanding of associations. For more information, see What Happens When You Create Entity Objects and Associations from Existing Tables.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for View Objects.

You will need to complete these tasks:

- Create the entity-based view object for the nonpersistent entity object, as described in How to Create an Entity-Based View Object.
- Use the Create Association wizard to create an association between the nonpersistent entity object and a database-derived entity object. For details about creating an association, see Creating and Configuring Associations.

To customize an association for nonpersistent entity objects:

1. In the Applications window, double-click the association that you want to customize.
2. In the overview editor, click the **Tuning** navigation tab and then click the **Edit accessors** icon for the **Custom View Objects** section.
3. In the Custom Views dialog, select **Use Custom View Object** for the nonpersistent entity object.
4. In the **Select View Object** list, select the view object that you created based on the nonpersistent entity-object.
5. Click **OK**.

How to Find Rows of a Master View Object Using Row Finders

Row finders are objects that the application may use to locate specific rows within a row set using a view criteria. The row finder may be invoked on the master row set to match one or more criteria attributes supplied by the detail row set.

Currently, row finders that you define at design time can participate in these master-detail scenarios where non-row key attribute lookup is desired at runtime:

- When you expose the row finder in an ADF Business Components web service, the end user may initiate row updates on the specific rows of the master service view instance that match one or more row-finder mapped attribute values on the detail service view instance. For more information about this use case, see What You May Need to Know About View Criteria and Row Finder Usage.
- Programmatically, in the application, when you want to obtain a set of master view rows that match one or more row-finder mapped, non-key attribute values in the detail view instance. For more information about this use case, see Programmatically Invoking the Row Finder.

> **Note:**
>
> Row finders that you define on view objects are not currently supported by the ADF Business Components **data control** for use in a view project. Therefore row-finder mapped attributes of a view object that the application exposes as a collection in the **Data Controls panel** will not participate in row finder lookup operations.

The view criteria attributes defined by the view criteria can be any attribute that the detail view object defines and specifically need not be a row key attribute. For example, in a master-detail relationship, where `PersonVO` is the master view object and `AddressVO` is the detail view object, the attribute `EmailAddress` will locate the person row that matches the email address of the person in the `AddressVO` row set.

To create a row finder to locate rows of the master view object using criteria attributes supplied by the detail view object, perform the following steps:

1. Define an inline view criteria on the master view object to reference the detail view object and specify the criteria attributes as a view criteria items defined by bind variables.
2. Define an updatable transient attribute on the master view object to receive the value of each criteria attribute. At runtime, the transient attribute may be set programmatically or exposed in a web service payload to allow an end user to supply the criteria attribute value (for example, an email address).
3. Define the row finder on the master view object by selecting the view criteria and setting each criteria attribute's bind variable to a corresponding transient attribute from the master view object.

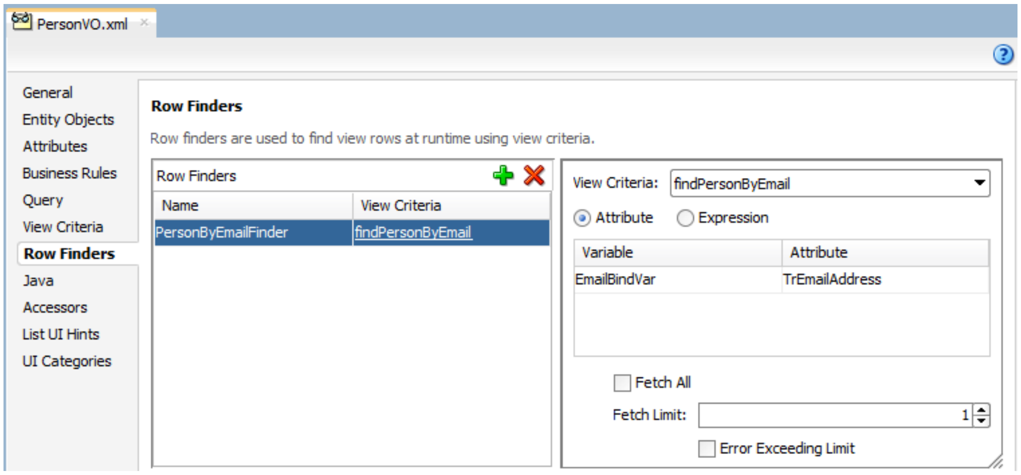
Defining a Row Finder on the Master View Object

Row finders that you define to filter the row set of a master view object are defined entirely on the master view object. The row finder matches the criteria attributes that you specified in the inline view criteria with values obtained at runtime. In order to receive the criteria attribute values, the master view object may define transient attributes that the application can set programmatically using row finder methods on the view object or that a web service payload may expose to receive end user-supplied values.

You use the Row Finders page of the master view object's overview editor to define the row finder. In the editor, you define a value source for each bind variable of the view criteria that you select. Attributes of the view object or attribute-value expressions are both valid sources for bind variables defined by the row finder.

Figure 6-10 shows the overview editor with a row finder that maps the view criteria bind variable `EmailBindVar` specified in the `findPersonByEmail` view criteria to the transient attribute `TrEmailAddress` defined by the master view object `PersonVO` as its value source.

Figure 6-10 View Object Overview Editor with Row Finder



Row Finders

Row finders are used to find view rows at runtime using view criteria.

Row Finders

Name	View Criteria
PersonByEmailFinder	findPersonByEmail

View Criteria: findPersonByEmail

● Attribute ○ Expression

Variable	Attribute
EmailBindVar	TrEmailAddress

□ Fetch All

Fetch Limit: 1

□ Error Exceeding Limit

> **Note:**
>
> You can also define row finders on view objects that do not participate in a master-detail hierarchy, as described in Working with Row Finders.

Before you begin:

It may be helpful to have an understanding of the row finder for master-detail view objects. For more information, see How to Find Rows of a Master View Object Using Row Finders.

It may be helpful to have an understanding of master-detail view objects. For more information, see About Master-Detail View Objects.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for View Objects.

You will need to complete these tasks:

- Create the desired master-detail view objects, as described in How to Create a Master-Detail Hierarchy Based on Entity Associations or How to Create a Master-Detail Hierarchy Based on View Objects Alone.
- Define an inline view criteria on the master view object, as described in Working with Master-Detail Related View Objects in View Criteria. The inline view criteria references the detail view object and specifies one or more detail attributes as view criteria items.
- Optionally, add updatable transient attributes to the master view object, as described in Adding Calculated and Transient Attributes to a View Object. The transient attribute may be set programmatically by the application or exposed in an ADF Business Components web service payload to solicit the value from an end user. The transient attributes must be defined as updatable to receive the criteria lookup value.

To create a row finder for a master view object:

1. In the Applications window, double-click the master view object that you want to customize with a new row finder.
2. In the overview editor, click the **Row Finders** navigation tab and then click the **Add Row Finder** button.
3. In the **Name** field, rename the row finder.

   For example, a row finder that you define to locate a person record by their email address, might be named `PersonByEmailFinder`.
4. Select the new row finder in the **Row Finders** list and then, in the **View Criteria** dropdown, select the view criteria that filters the view object row set.

   The desired row finder should appear highlighted, as shown in Figure 6-10. The view criteria list will be empty unless you have created a view criteria for the row finder to use.
5. When you want the end user of an ADF Business Components web service to supply a value, leave **Attribute** selected, select the bind variable from the list, and

then, in the **Attribute** dropdown, select the transient attribute from the master view object that supplies the bind variable value at runtime.

The transient attribute that you select will be used by the row finder to supply the criteria attribute value. For example, a PersonVO master view object that matches rows based on the email address in the AddressVO, might include a transient attribute that you define named TrEmailAddress.

Note: When you create the transient attribute in the overview editor for the view object, be sure to define the transient attribute as **Updatable** so it can receive the criteria lookup value.

6. Deselect **FetchAll** when you want to specify the number of rows that the row finder is allowed to match.

   When you enter a number of rows for **Fetch Limit**, you can also select **Error Exceeding Limit** to enable Oracle ADF to throw an exception when more matching rows exist in the database than the number you specify for **Fetch Limit**.

What Happens When You Create a Row Finder for a Master View Object

When you create a master view object row finder, the view object definition contains all the metadata required by the row finder, including the row finder definition itself. As the following example shows, the metadata for a row finder PersonByEmailFinder includes a bind variable EmailBindVar in the <Variable> element, a transient attribute TrEmailAddress in the <ViewAttribute> element, a view criteria findPersonByEmail and an inline view criteria AddressVONestedCriteria in the <ViewCriteria> element, and finally the row finder PersonByEmailFinder in the <RowFinders> element of the view object definition.

The row finder <VarAttributeMapping> subelement maps the view criteria bind variable EmailBindVar to the transient attribute TrEmailAddress of the master view object, which allows the end user to supply a value at runtime and allows the application to invoke the row finder with the required criteria attribute. The inline <ViewCriteriaItem> subelement sets the criteria attribute EmailAddress on the detail view object to the value of the bind variable.

At runtime, when the row finder is invoked on the master view object, one or more detail criteria attributes identify the matching rows in the master row set. In this example, the email address is used to match the person to which the email address belongs. The row finder locates the person record without the need to pass a row key value.

```
<ViewObject
  xmlns="http://xmlns.oracle.com/bc4j"
  Name="PersonVO"
  SelectList="PersonEO.ID,
       PersonEO.FIRST_NAME
       PersonEO.LAST_NAME"
  FromList="PERSON PersonEO"
  ...
  <Variable
    Name="EmailBindVar"
    Kind="viewcriteria"
    Type="java.lang.String"/>
  <EntityUsage
    Name="PersonEO"
    Entity="model.entities.PersonEO"/>
    ...
  <ViewAttribute
    Name="TrEmailAddress"
    IsUpdateable="false"
    IsSelected="false"
    IsPersistent="false"
    PrecisionRule="true"
    Type="java.lang.String"
    ColumnType="CHAR"
    AliasName="VIEW_ATTR"
    SQLType="VARCHAR"/>
  <ViewCriteria
    Name="findPersonByEmail"
    ViewObjectName="model.views.PersonVO"
    Conjunction="AND">
    ...
    <ViewCriteriaRow
      Name="PersonVOCriteria_row_0"
      UpperColumns="1">
      <ViewCriteriaItem
        Name="PersonVOCriteria_PersonVOCriteria_row_0_AddressVO"
        ViewAttribute="AddressVO"
        Operator="EXISTS"
        Conjunction="AND"
        IsNestedCriteria="true"
        Required="Optional">
        <ViewCriteria
          Name="AddressVONestedCriteria"
          ViewObjectName="model.views.AddressVO"
          Conjunction="AND">
          <ViewCriteriaRow
            Name="AddressVONestedCriteria_row_0"
            UpperColumns="1">
            <ViewCriteriaItem
              Name="AddressVONestedCriteria_row_0_EmailAddress"
              ViewAttribute="EmailAddress"
              Operator="="
              Conjunction="AND"
              Value=":EmailBindVar"
              IsBindVarValue="true"
              Required="Optional"/>
          </ViewCriteriaRow>
        </ViewCriteria>
      </ViewCriteriaItem>
    </ViewCriteriaRow>
  </ViewCriteria>
  ...
   <RowFinders>
    <AttrValueRowFinder
      Name="PersonByEmailFinder"
      FetchLimit="1">
      <ViewCriteriaUsage
        Name="findPersonByEmail"
        FullName="model.views.PersonVO.findPersonByEmail"/>
      <VarAttributeMap>
        <VariableAttributeMapping
          Variable="EmailBindVar"
          Attribute="TrEmailAddress"/>
      </VarAttributeMap>
    </AttrValueRowFinder>
  </RowFinders>
</ViewObject>
```

Working with a Single Table in a Recursive Master-Detail Hierarchy

JDeveloper allows you to create a master-detail hierarchy based on a single view object when the source and target attribute exist in one table.

A recursive data model is one that utilizes a query that names source and destination attributes in a master-detail relationship based on a single table. In a typical master-detail relationship, the source attribute is supplied by the primary key attribute of the master view object and the destination attribute is supplied by foreign key attribute in the detail view object. For example, a typical master-detail relationship might relate the `DepartmentId` attribute on the `DEPARTMENT` table and the corresponding `DepartmentId` attribute on the `EMPLOYEE` table. However, in a recursive data model, the source attribute `EmployeeId` and the target attribute `ManagerId` both exist in the `EMPLOYEE` table. The query for this relationship therefore involves only a single view object. In this scenario, you create the view object for a single base entity object that specifies both attributes and then you define a self-referential view link to configure this view object as both the "source" and the "target" view object to form a master-detail hierarchy.

After you create the view link, there are two ways you can handle the recursive master-detail hierarchy in the data model project. You can either:

- Create a data model that exposes two instances of the same view object, one playing the role as master and the other playing the role as detail, actively coordinated by a view link instance. This can be useful when you anticipate needing to show a single level of master rows and detail rows at a time in two separate tables.
- Create a data model that exposes only a single instance of the view object, and use the view link accessor attribute in each row to access a row set of details. This is the more typical use case of the two because it allows you to display (or programmatically work with) the recursive master-detail hierarchy to any number of levels that exist in the data. For example, to show the recursive hierarchy in a `tree` or `treeTable` component, you would use this approach, as described in How to Display Master-Detail Objects in Trees.

How to Create a Recursive Master-Detail Hierarchy for an Entity-Based View Object

In a recursive master-detail hierarchy, the attributes of the view object that you select for the source and destination in the view link will typically be the same pair of attributes that define the self-referential association between the underlying entity object, if this association exists. While this underlying association is not required to create the view link, it does simplify the creation of the view link, so you will first create a foreign key association for the base entity object of the view object.

To create the recursive master-detail hierarchy:

1. Create an self-referential view link using the base entity object of the view object.
2. Expose the view object with a view criteria that will filter the view instance's results to include only those rows you want to see at the "root" of the hierarchy.

Creating an Association-Based, Self-Referential View Link

To create an association, you use the Create Association wizard. Then the association will appear as a selection choice when you use the Create View Link wizard. The view link will be self-referential because the association you select for the source and the destination view object names the *same* entity object, which is derived from a single database table.

Before you begin:

It may be helpful to have an understanding of the recursive data model. For more information, see Working with a Single Table in a Recursive Master-Detail Hierarchy.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for View Objects.

You will need to complete this task:

- When you create the view link JDeveloper won't be able to infer the association between the source and destination attributes of the entity object. To support the recursive hierarchy, you can use the Create Association wizard to create an association between the source attribute and the destination attribute. On the Entity Objects page, select the same entity object to specify the source and destination attributes and leave all other default selections unchanged in the wizard. For details about creating an association, see Creating and Configuring Associations.

  For example, assume the recursive master-detail hierarchy displays a list of employees based on their management hierarchy. In this scenario, you would create the association based on the `Employees` entity object. On the Entity Objects page of the Create Association wizard, you would select `Employees.EmployeeId` as the source attribute and `Employee.ManagerId` as the destination attribute. The entity object `Employees` supplies both attributes to ensure the association is self-referential.

To create an association-based, self-referential view link:

1. In the Applications window, right-click the project in which you want to create the view object and choose **New**.

   To avoid having to type in the package name in the Create View Link wizard, you can choose **New** and then **View Link** on the context menu of the **links** package node in the Applications window.

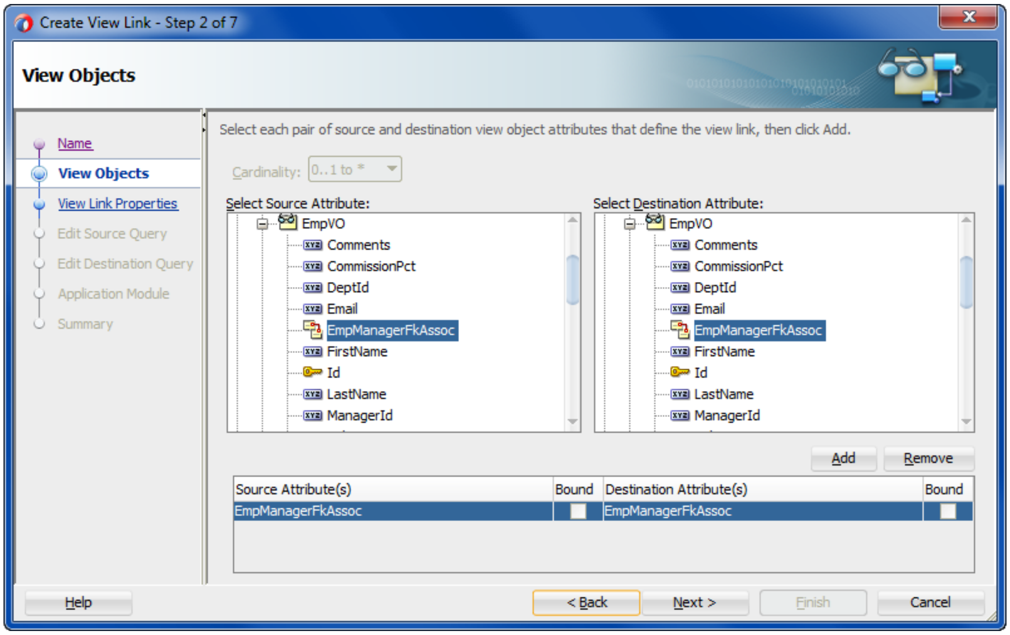
2. In the New Gallery, expand **Business Tier**, select **ADF Business Components** and then **View Link**, and click **OK**.
3. In the Create View Link wizard, on the Name page, supply a package and a component name.
4. On the View Objects page, in the **Select Source Attribute** tree expand the source view object in the desired package. In the **Select Destination Attribute** tree expand the destination view object.

   For entity-based view objects, notice that in addition to the view object attributes, relevant associations also appear in the list.

5. Select the same association in both **Source** and **Destination** trees. Then click **Add** to add the association to the table below.

For example, Figure 6-11 shows the same `EmpManagersFkAssoc` association in both **Source** and **Destination** trees selected. The view link is self-referential because the definition of the association names the source and destination attribute on the *same* entity object (in this case, `Employees`).

Figure 6-11 Master and Detail Related by a Self-Referential Association Selection


Create View Link - Step 2 of 7

View Objects

Select each pair of source and destination view object attributes that define the view link, then click Add.

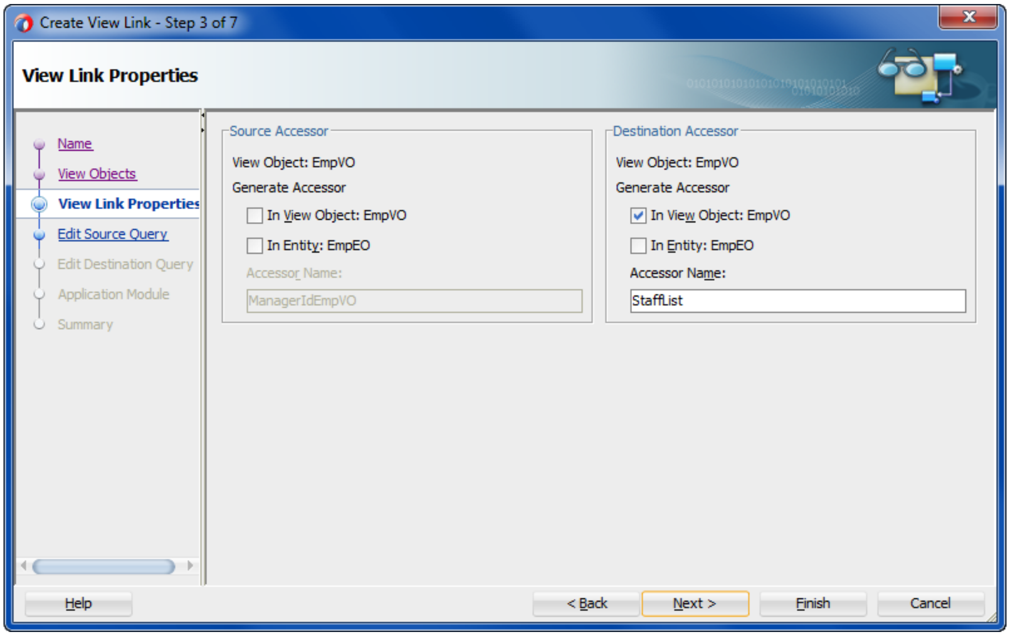
Cardinality: 0..1 to *

Source Attribute(s)	Bound	Destination Attribute(s)	Bound
EmpManagerFkAssoc	□	EmpManagerFkAssoc	□

6. On the View Link Properties page, leave the default selections unchanged, but edit the accessor name of the destination accessor to provide a meaningful name.

   For example, Figure 6-12 shows the destination accessor has been renamed from `EmployeesView` to `StaffList`. This name will be exposed in the binding editor when the user interface developer populates the ADF Faces tree component by selecting this accessor. The name you provide will make clear to the UI developer the purpose of the accessor; in this case, to generate a list of employees associated with each manager.

Figure 6-12 Renamed Destination Accessor in View LInk



Source Accessor

View Object: EmpVO

Generate Accessor

□ In View Object: EmpVO

□ In Entity: EmpEO

Accessor Name: ManagerIdEmpVO

Destination Accessor

View Object: EmpVO

Generate Accessor

☑ In View Object: EmpVO

□ In Entity: EmpEO

Accessor Name: StaffList

7. Click **Finish**.

Exposing the View Instance and Filter with a View Criteria

When you are ready to expose the view object in your project's data model, you will configure the view instance in the data model to use a view criteria to filter the initial value in the root of the tree. For example, assume the recursive master-detail hierarchy displays a list of employees based on their management hierarchy. In this scenario, you'll configure the view criteria's bind variable to specify the employee ID of the employee that you want to be the root value of the entire hierarchy. In this case, the root value of the recursive hierarchy of managers and employees would be the employee ID of the highest level manager in the organization.

Before you begin:

It may be helpful to have an understanding of the recursive data model. For more information, see Working with a Single Table in a Recursive Master-Detail Hierarchy.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for View Objects.

You will need to complete this task:

- Create the entity-based view object and create a view criteria that will filter the view instance's results to include only those rows you want to see at the "root" of the hierarchy. To create a view criteria that uses a bind variable to filter the view object, see Working with Named View Criteria.

  For example, in a recursive hierarchy of managers and employees, you would create the entity-based view object `EmployeesView`. After you create the view object in the Create View Object wizard, you can use the Query page of the overview editor to create a bind variable and view criteria which allow you to identify the employee or employees that will be seen at the top of the hierarchy.

If only a single employee should appear at the root of the hierarchy, then the view criteria in this scenario will filter the employees using a bind variable for the employee ID (corresponding to the source attribute) and the WHERE clause shown in the Create View Criteria dialog would look like `( (Employees.EMPLOYEE_ID = :TheEmployeeId ) )`, where `TheEmployeeId` is the bind variable name. For more information on creating a view criteria that uses a bind variable to filter the view object, see Creating a Data Source View Object to Control the Cascading List.

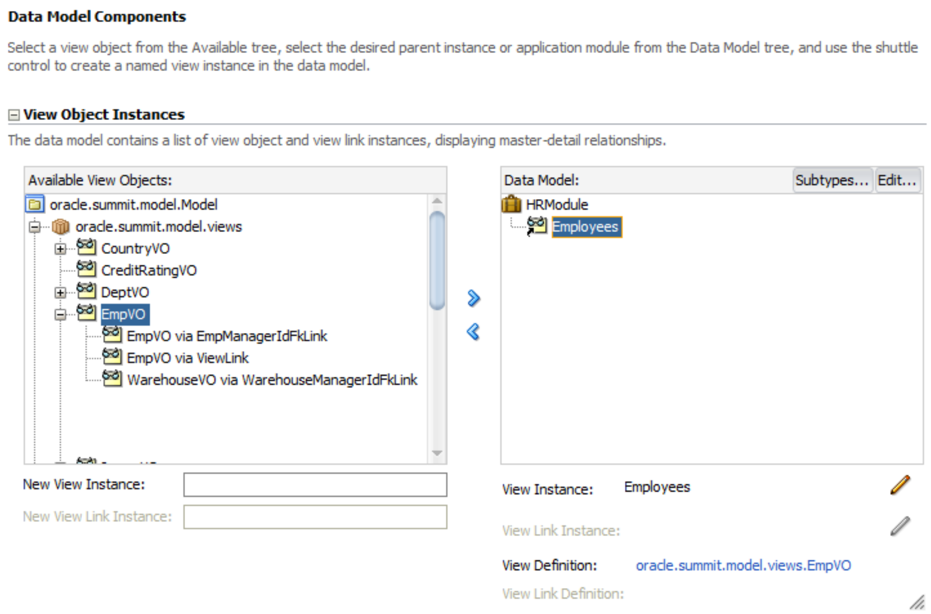
To define the view object instance in an existing application module:

1. In the Applications window, double-click the application module for which you want to define a new view instance.
2. In the overview editor, click the **Data Model** navigation tab.
3. In the Data Model page, expand the **View Object Instances** section and, in the **Available View Objects** list, select the view object definition that you defined the view criteria to filter.

   The **New View Instance** field below the list shows the name that will be used to identify the next instance of that view object that you add to the data model.
4. To change the name before adding it, enter a different name in the **New View Instance** field.
5. With the desired view object selected, shuttle the view object to the **Data Model** list.

   Figure 6-13 shows the view object `EmployeesView` has been renamed to `Employees` before it was shuttled to the **Data Model** list.

Figure 6-13 Data Model Displays Added View Object Instance



Data Model Components

Select a view object from the Available tree, select the desired parent instance or application module from the Data Model tree, and use the shuttle control to create a named view instance in the data model.

View Object Instances

The data model contains a list of view object and view link instances, displaying master-detail relationships.

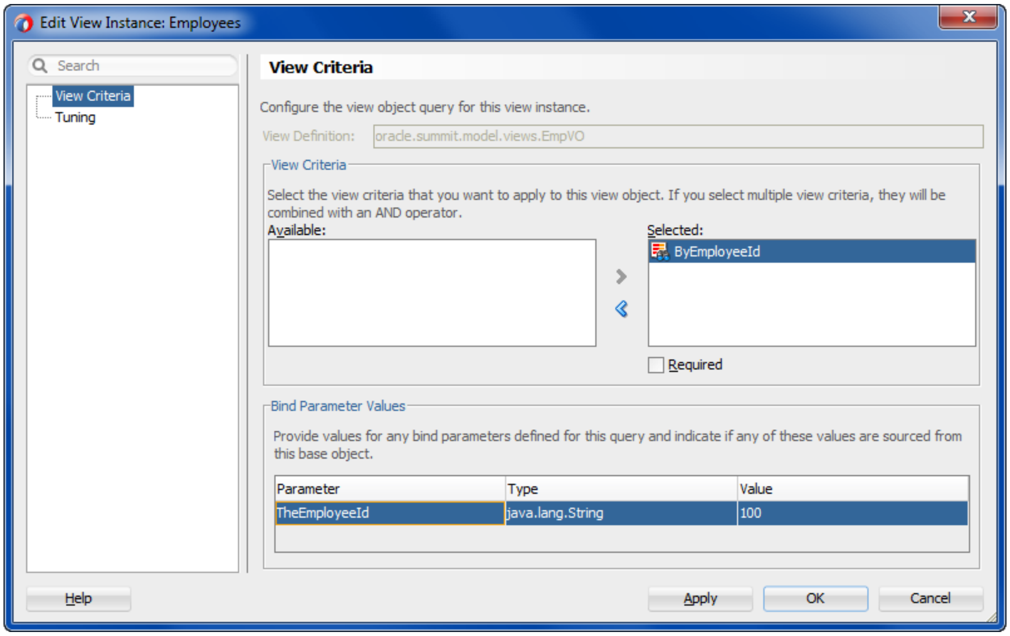
View Instance: Employees

View Definition: oracle.summit.model.views.EmpVO

6. To filter the view object instance so that you specify the root value of the hierarchy, select the view object instance you added and click **Edit**.
7. In the Edit View Instance dialog, shuttle the view criteria you created to the **Selected** list and enter the bind parameter value that corresponds to the root of the hierarchy.

Figure 6-14 shows the view object `ByEmployeeId` view criteria with the bind parameter `TheEmployeeId` set to the value `100` corresponding to the employee that is at the highest level of the hierarchy.

Figure 6-14 View Criteria Filters View Instance



View Criteria

Configure the view object query for this view instance.

View Definition: oracle.summit.model.views.EmpVO

View Criteria

Select the view criteria that you want to apply to this view object. If you select multiple view criteria, they will be combined with an AND operator.

Available:

Selected: ByEmployeeId

□ Required

Bind Parameter Values

Provide values for any bind parameters defined for this query and indicate if any of these values are sourced from this base object.

Parameter	Type	Value
TheEmployeeId	java.lang.String	100

8. Click **OK**.

What Happens When You Create a Recursive Master-Detail Hierarchy

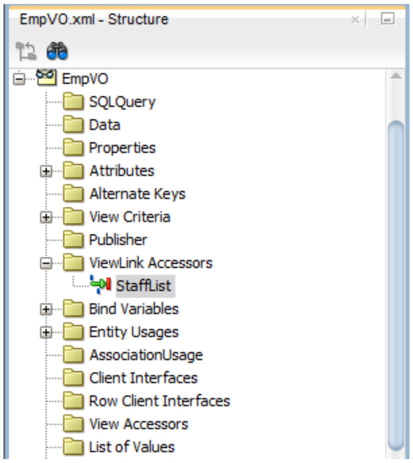
When you create an self-referential view link, JDeveloper creates the XML document file that represents its declarative settings and saves it in the directory that corresponds to the name of its package. This XML file contains the declarative information about the source and target attribute pairs that the association you selected specifies and contains the declarative information about the association that relates the source and target view object you selected.

The following example shows the `EmpManagerFkLink` defines the same view object `EmployeesView` for the source and destination in its XML document file.

```
<ViewLink
  xmlns="http://xmlns.oracle.com/bc4j"
  Name="EmpManagerFkLink"
  EntityAssociation="test.model.EmpManagerFkAssoc">
  <ViewLinkDefEnd
    Name="EmployeesView1"
    Cardinality="1"
    Owner="test.model.EmployeesView"
    Source="true">
    <DesignTime>
      <Attr Name="_finderName" Value="ManagerIdEmployeesView"/>
      <Attr Name="_isUpdateable" Value="true"/>
    </DesignTime>
    <AttrArray Name="Attributes">
      <Item Value="test.model.EmployeesView.EmployeeId"/>
    </AttrArray>
  </ViewLinkDefEnd>
  <ViewLinkDefEnd
    Name="EmployeesView2"
    Cardinality="-1"
    Owner="test.model.EmployeesView">
    <DesignTime>
      <Attr Name="_finderName" Value="StaffList"/>
      <Attr Name="_isUpdateable" Value="true"/>
    </DesignTime>
    <AttrArray Name="Attributes">
      <Item Value="test.model.EmployeesView.ManagerId"/>
    </AttrArray>
  </ViewLinkDefEnd>
</ViewLink>
```

In addition to saving the view link component definition itself, JDeveloper also updates the XML definition of the view object to add information about the view link accessor you've defined. As a confirmation of this, you can select the view object in the Applications window and inspect its details in the Structure window. As shown in Figure 6-15, you can see the defined accessor under the **ViewLink Accessors** node for the `EmployeesView` view object of the `EmpManagerFkLink` view link.

Figure 6-15 View Object with View Link Accessor in the Structure Window



Working with Master-Detail Related View Objects in View Criteria

The ADF Business Components framework allows you to apply view criteria filters on master view objects using attributes of detail view objects.

View criteria provide a declarative way to define view object query filters that apply to the view object's own query. In certain cases you may need to filter the master view object using attributes of the detail view object. View criteria that involve master and detail view objects rely on the `EXISTS` operator and an inline view criteria to define the query filter. The `EXISTS` operator allows the inline view criteria to reference a detail view object and apply attributes that you select as criteria items. For example, a view criteria used in combination with a row finder can filter the master row set and locate specific rows in the row set using view criteria items from the detail view object.

You use the View Criteria page of the view object overview editor to define the view criteria on the master view object. For example, a view criteria `findPersonByEmail` that filters the rows in the master `PersonVO` using the `EmailAddress` attribute of the detail `AddressVO` might look similar to the one shown in the following example. In this example, the view criteria statement uses a bind variable `EmailBindVar` to obtain the value of the email attribute on the detail view object `AddressVO` at runtime.

```
( (EXISTS(SELECT 1 FROM ADDRESS AddressEONQ1 WHERE
    (UPPER(AddressEONQ1.EMAIL_ADDRESS LIKE UPPER (:EmailBindVar || '%') ) )
         AND
      (PersonEO.PERSON_ID = AddressEONQ1.PERSON_ID) ) ) )
```

Before you begin:

It may be helpful to have an understanding of the row finder that operates on a master view object. See How to Find Rows of a Master View Object Using Row Finders.

It may be helpful to have an understanding of view criteria. See Working with Named View Criteria.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. See Additional Functionality for View Objects.

You will need to complete this task:

> Create the desired master-detail view objects, as described in How to Create a Master-Detail Hierarchy Based on Entity Associations or How to Create a Master-Detail Hierarchy Based on View Objects Alone.

To create an inline view criteria:

1. In the Applications window, double-click the master view object for which you want to customize with a new inline view criteria.

   The view object that you create the inline view criteria on must be a master view object in a master-detail hierarchy.

2. In the overview editor, click the **View Criteria** navigation tab and click the **Create New View Criteria** button.

3. In the Create View Criteria dialog, enter the name of the view criteria to identify its usage in your application.

   For example, to filter the person row set by an employee's email, you might specify a view criteria name like `findPersonByEmail`.

4. In the **Query Execution Mode** dropdown list, leave the default **Database** selected.

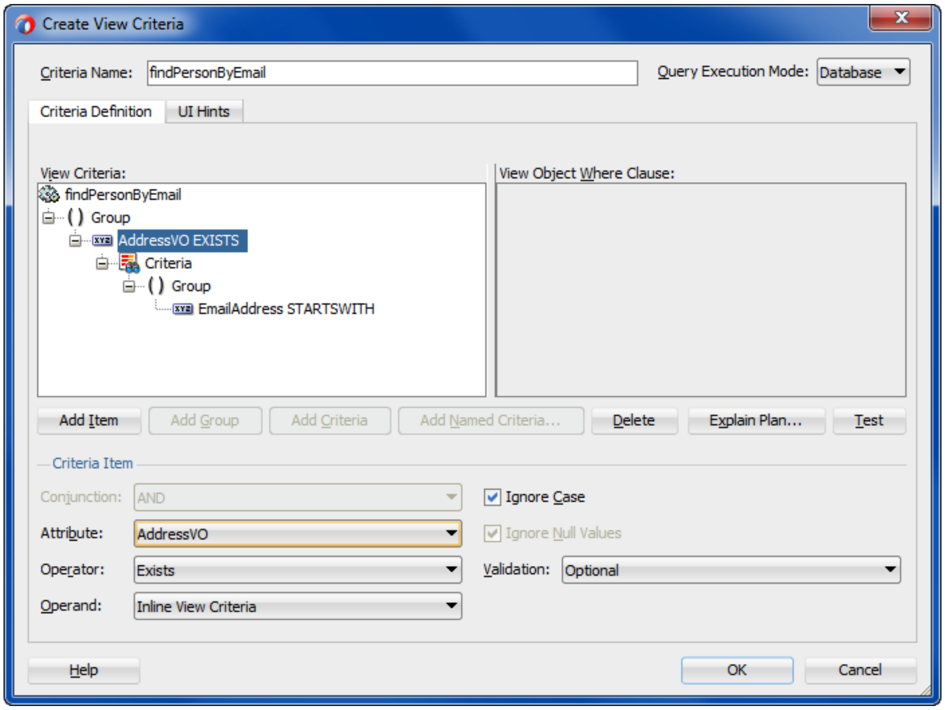
   The default mode allows the view criteria to filter the database table specified by the view object query.

5. In the Criteria Definition tab, click **Add Item** to define the view criteria.

6. In the view criteria hierarchy, select the unassigned view criteria item node beneath **Group** and, in the **Criteria Item** section, select the **Attribute** dropdown and select the detail view object from the list.

   When a view link exists for the current view object you are editing, you may select a detail view object to create a view criteria with a nested view criteria inline. The detail view object name will appear in the **Attribute** dropdown list. For example, `PersonVO` is linked to the `AddressVO` and a view criteria definition for findPersonByEmail filters the master view object PersonVO by attributes of the detail view object AddressVO, as shown in Figure 6-16.

Figure 6-16 Edit View Criteria Dialog with Inline View Criteria Specified



Criteria Name: findPersonByEmail

Query Execution Mode: Database

View Criteria:

```
findPersonByEmail
( ) Group
    AddressVO EXISTS
        Criteria
            ( ) Group
                EmailAddress STARTSWITH
```

Criteria Item

Conjunction: AND

☑ Ignore Case

Attribute: AddressVO

☑ Ignore Null Values

Operator: Exists

Validation: Optional

Operand: Inline View Criteria

7. Leave the default **Operator** and **Operand** selections unchanged.

   When you create an inline view criteria with a view object criteria item, the editor specifies the **Exists** operator and **Inline View Criteria** operand as the default choices. At runtime, the EXISTS operator tests for the presence of the attribute value in the detail view object and filter the rows in the master view object, as described in Working with Master-Detail Related View Objects in View Criteria.

8. In the view criteria hierarchy, select the new view criteria item node beneath **Group** and, in the **Criteria Item** section, enter the following:

   - In the **Attribute** dropdown, select the attribute from the detail view object that you want to filter by.

     The detail attribute is the attribute that will be used to filter the master view object. The value of this attribute can be supplied by the end user when a bind variable is defined for the criteria item.

   - In the **Operator** dropdown, select the desired filter operator.

   - In the **Operand** dropdown, select **Bind Variable** to specify the name of the bind variable for the detail attribute. If the variable was already defined for the view criteria, select it from the **Parameter** dropdown list. Otherwise, click **New** to display the New Variable dialog that lets you name a new bind variable on the view criteria.

     Note: When you select an existing bind variable from the dropdown list for use by the view criteria, do not select a bind variable that has been defined for the view object query (these will appear on the Query page of the overview editor). Select only bind variables that you define in the View Criteria page of the overview editor.

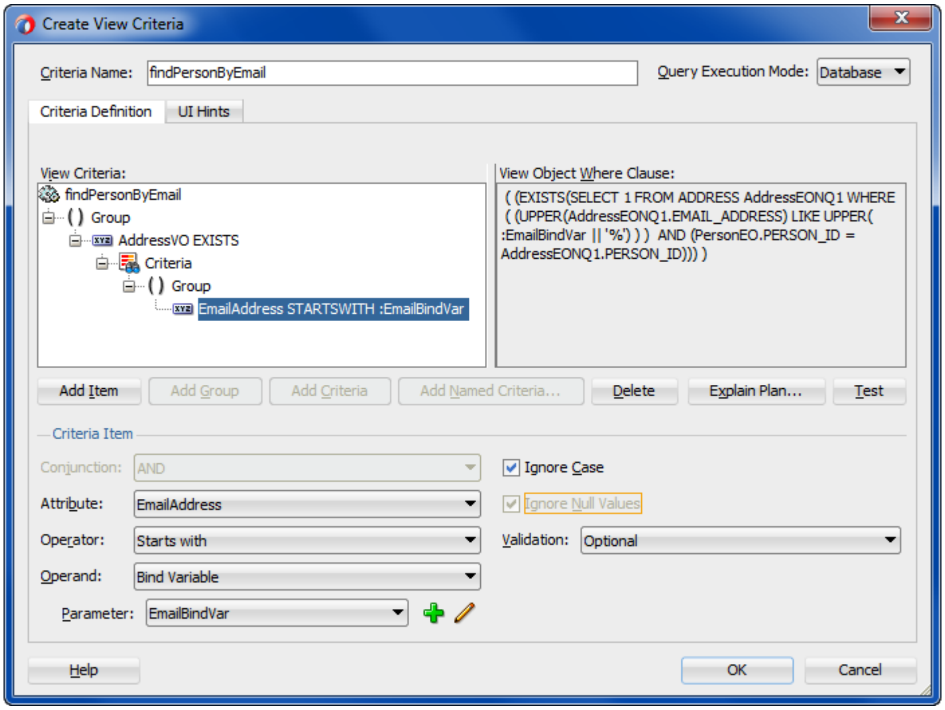
9. To disable case-sensitive filtering of the attribute based on the case of the runtime-supplied value, leave **Ignore Case** selected.

   This option is supported for attributes of type String only. The default disables case sensitive searches.

10. In the **Validation** dropdown list, leave the default **Optional** selected.

    Optional means the view criteria (or search field) is added to the `WHERE` clause only if the value is non-`NULL`. The default **Optional** for each new view criteria item means no exception will be generated for null values. For example, a `findPersonByEmail` view criteria definition filters the master view object `PersonVO` by the `Email` attribute of the detail view object `AddressVO`, and the value may be null, as shown in Figure 6-17.

Figure 6-17 Edit View Criteria Dialog with Bind Variable Criteria Item Specified



Criteria Name: findPersonByEmail

Query Execution Mode: Database

View Criteria:

findPersonByEmail
() Group
AddressVO EXISTS
Criteria
() Group
EmailAddress STARTSWITH :EmailBindVar

View Object Where Clause:

```
( (EXISTS(SELECT 1 FROM ADDRESS AddressEONQ1 WHERE
( (UPPER(AddressEONQ1.EMAIL_ADDRESS) LIKE UPPER(
:EmailBindVar || '%') ) )  AND (PersonEO.PERSON_ID =
AddressEONQ1.PERSON_ID))) )
```

Conjunction: AND

Attribute: EmailAddress

Operator: Starts with

Operand: Bind Variable

Parameter: EmailBindVar

☑ Ignore Case

☑ Ignore Null Values

Validation: Optional

11. Click **OK**.

12. In the View Criteria page of the overview editor, select the bind variable and in the **Details** section, leave the default settings unchanged:

    - Leave **Literal** selected but do not supply a default value. The value will be obtained from the end user at runtime.
    - Leave **Updatable** selected to allow the bind variable value to be defined through the user interface.

13. In the View Criteria page, click the **UI Hints** tab and specify hints like **Label**, **Format Type**, **Format** mask, and others.

    The view layer will use bind variable UI hints when you build user interfaces like search pages that allow the user to enter values for the named bind variables.

Note that formats are only supported for bind variables defined by the `Date` type or any numeric data type.

7
Defining Polymorphic View Objects

This chapter describes how to use inheritance in ADF Business Components to create an ADF view object that exposes multiple view row types in an Oracle ADF application. As an alternative to working with entity object inheritance, this chapter also describes how to define a view object inheritance hierarchy to generate heterogeneous view rows.
This chapter includes the following sections:

About Polymorphic View Objects

ADF Business Components allows you to work with polymorphic view objects that are based on entity objects with inherited attributes from an inheritance hierarchy. A polymorphic view object has a heterogeneous row set.

The **view object** lets you create a table represented by its base **entity object**. Typically, when you create the entity-based view object, you create it to work with entity rows of a single type like `Domestics`, which perhaps include specific attributes that pertain to domestic customers. At other times you may want to query and update rows based on an entity object inheritance hierarchy in the same row set. For example, in the same row set, you might work with attributes that are common to the inheritance hierarchy of `Customers`, `Domestics`, *and* `Internationals` entity objects, where `Domestics` and `Internationals` are subtype entity objects of the parent `Customers` entity object. Such a view object is called a **polymorphic view object**. For a client to properly use the polymorphic view object, the view object must know which entity object to delegate to for each row in its heterogeneous row set. To identify the row type, the polymorphic view object relies on a discriminator attribute to identify each row's corresponding table.

ADF Business Component supports these types of polymorphism:

- The entity object defines polymorphic subtypes, but the view object does not.

  This is useful when the backend business logic needs to be aware of the different types but the presentation layer needs to treat view instances as a single type. In this case, the view instance will contain the common set of attributes defined by the entity subtypes.

- Both the entity object and the view object usages define polymorphic subtypes.

  In this case, both the backend business logic and the presentation layer need to distinguish between the various subtypes.

- The entity object does not define polymorphic subtypes, but the view object usages do define polymorphic subtypes.

  In this case, the presentation layer needs to distinguish between the view subtypes, but the backend business logic is not unique.

When you create polymorphic view object based on inheritance of entity subtypes, the various entity subtypes are not exposed to the client. As far as the client is concerned, every row in the view object's result set has the same set of attributes and the same row type. This is accomplished because ADF Business Components ensures that the correct entity object row subtype is created based on the value of the discriminator attribute. In other words, with a polymorphic entity usage, only the entity row parts of a view row can change type, while the view row type is constant.

Whereas, when you create polymorphic view object based on the inheritance defined by a view object hierarchy (not entity subtypes), you configure the application module to allow one or more view row subtypes in the result. Only polymorphic view rows produce different subtypes of client-visible view row types. And, unlike view objects derived from polymorphic entity usages, only polymorphic view rows allow you to expose view row subtypes to the client with additional attributes.

> **Note:**
>
> To experiment with the examples that demonstrate the concept in this chapter, use the `SummitADF_Examples` workspace, as described in Running the Standalone Samples from the SummitADF_Examples Workspace. For information about how to obtain and install the Summit ADF standalone sample applications, see Setting Up the Summit Sample Applications for Oracle ADF.

Polymorphic Entity Usages and Polymorphic View Rows Usages

You can work with either type of polymorphism, or you can combine the two.

While often even more useful when used *together*, the view row polymorphism and the polymorphic entity usage features are distinct and can be used separately. In particular, the view row polymorphism feature can be used for read-only view objects, as well as for entity-based view objects. When you combine both mechanisms, you can have both the entity row part being polymorphic, as well as the view row type.

Note to use view row polymorphism with either view objects or entity objects, you must configure the discriminator attribute property separately for each. This is necessary because read-only view objects contain no related entity usages from which to infer the discriminator information.

A limitation exists when defining polymorphic view objects that requires view objects with discriminator attributes to be defined no more than one level deep. Your project may define a polymorphic view object with discriminator attributes to specify the row subtypes, but using another view object to specify further subrows based on the first polymorphic view object will cause the display of subrows to fail. When creating polymorphic view objects, limit the discriminator attribute definition to one level.

In summary, to create a view object with a polymorphic entity usage:

1. Configure an attribute to be the discriminator at the entity object level in the root entity object in an inheritance hierarchy.
2. Define a hierarchy of inherited entity objects, each of which overrides and provides a distinct value for the Subtype Value property of that entity object level discriminator attribute.
3. List the subclassed entity objects in a view object's list of subtypes.
4. Update the application module to expose the entity-based subtype view objects in this hierarchy in the application module's list of subtypes.

Whereas, to work with view row polymorphism:

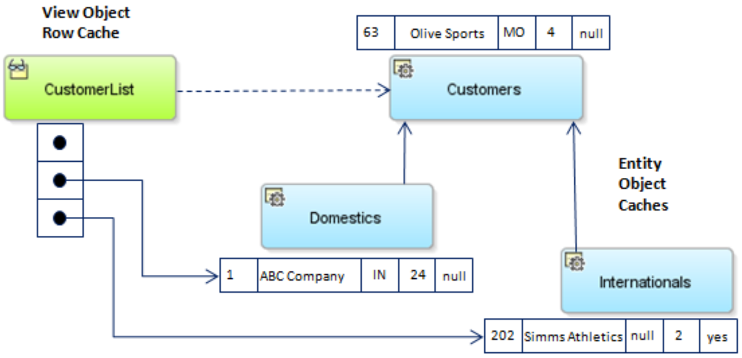
1. Configure an attribute to be the discriminator at the view object level in the base view object in an inheritance hierarchy.
2. Define a hierarchy of inherited view objects each of which provides a distinct value for the Subtype Value property of that discriminator attribute at the view object level (identified as `DefaultValue` for the attribute in the view object definition file).
3. Update the application module to expose the subclassed view objects in this hierarchy in the application module's list of subtypes.

Working with Polymorphic Entity Usages

You can use an ADF view object with a polymorphic entity usage to create a heterogeneous row set based on a base entity object and its subtypes as well.

A *polymorphic* entity usage is one that references a base entity object in an inheritance hierarchy and is configured to handle *subtypes* of that entity as well. Figure 7-1 shows the results of using a view object with a polymorphic entity usage. The entity-based `CustomerList` view object has the `Customers` entity object as its primary entity usage. The view object partitions each row retrieved from the database into an entity row with attributes specific to the various subtypes of `Customers`. It creates the appropriate entity row subtype based on consulting the value of the discriminator attribute. For example, if the `CustomerList` query retrieves one row for domestic company `ABC Company` and one row for international company `Simms Athletics`, the underlying entity rows would be as shown in the figure.

Figure 7-1 View Object with a Polymorphic Entity Usage Handles Entity Subtypes



> **Note:**
>
> The example in this section refers to the `oracle.summit.model.polymorphic` package in the `SummitADF_Examples` application workspace.

How to Create a Subtype View Object with a Polymorphic Entity Usage

The view object that you create with a polymorphic entity usage may inherit one or more of the attributes of the base entity object and the entity subtypes. Attributes that you select from the entity objects will be overridden by the view object attribute definitions. When an entity-based view object references an entity object with a discriminator attribute, then Oracle ADF enforces that the discriminator attribute is included in the query (in addition to the primary key attribute).

Before you begin:

It may be helpful to have an understanding of polymorphic view objects (also called in this section, subtype view objects). For more information, see About Polymorphic View Objects.

You will need to complete these tasks:

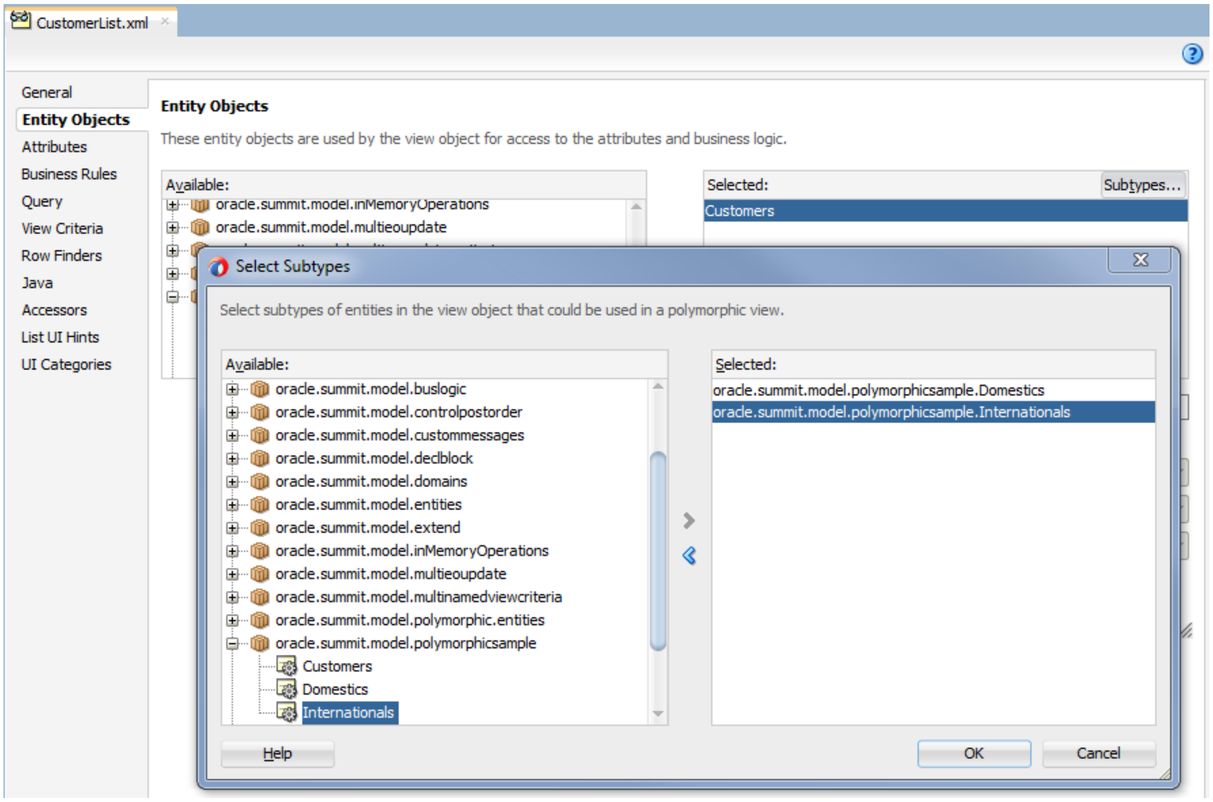
1. Create the base entity object and entity subtypes upon which the view objects will be based. For details about creating base entity objects and entity subtypes, see How to Create Entity Objects in an Inheritance Hierarchy.
2. Create an entity-based view object from the base entity object and select the attributes that are common to the subtype view objects. The base view object must contain the superset of attributes represented by each subtype view object. For details about creating an entity-based view object, see How to Create an Entity-Based View Object.
3. In the base view object that you create, select the entity subtype corresponding to each subtype view object that you will create. JDeveloper limits the subtype view objects that you may create to the polymorphic entity usage selections you make.

   To select entity subtypes in the base view object:

   a. Open the base view object in the overview editor and click the **Entity Objects** navigation tab.
   b. In the Entity Objects page, click **Subtypes** and in the Select Subtypes dialog, add the entity subtypes to the **Selected** list.

   For example, as Figure 7-2 shows, the entity subtype `Domestics` and `Internationals` have been selected to anticipate creating subtype view objects for each.

Figure 7-2 Entity Subtypes Define Possible Subtype View Objects



Entity Objects

These entity objects are used by the view object for access to the attributes and business logic.

Available:
oracle.summit.model.inMemoryOperations
oracle.summit.model.multieoupdate

Selected:
Customers

Select subtypes of entities in the view object that could be used in a polymorphic view.

Available:
oracle.summit.model.buslogic
oracle.summit.model.controlpostorder
oracle.summit.model.custommessages
oracle.summit.model.declblock
oracle.summit.model.domains
oracle.summit.model.entities
oracle.summit.model.extend
oracle.summit.model.inMemoryOperations
oracle.summit.model.multieoupdate
oracle.summit.model.multinamedviewcriteria
oracle.summit.model.polymorphic.entities
oracle.summit.model.polymorphicsample
Customers
Domestics
Internationals

Selected:
oracle.summit.model.polymorphicsample.Domestics
oracle.summit.model.polymorphicsample.Internationals

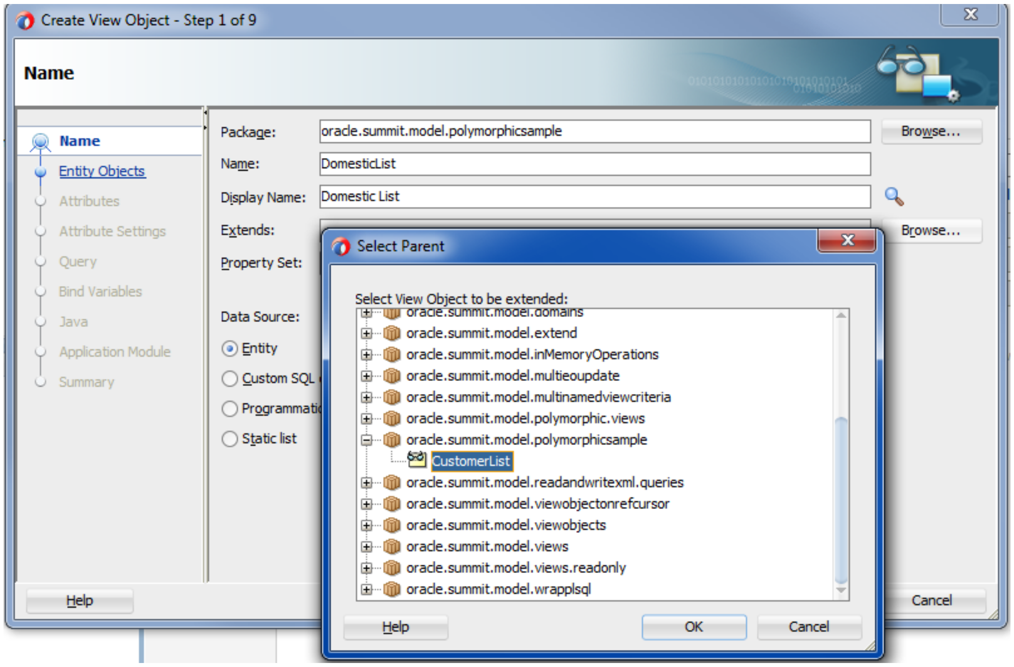
To create a subtype view object with a polymorphic entity usage:

1. In the Applications window, right-click your data model project and choose **New** and then **View Object**.
2. In the Create View Object wizard, name the subtype view object and then next to the **Extends** field, click **Browse**.

   For example, you might name the subtype view object `DomesticList` or `InternationalList` when the subtype view object extends a base view object `CustomerList`. In this case, the data model project would define a base `Customers` entity object in order to support the creation of subtype view objects with polymorphic entity usages for the `Domestics` and `Internationals` subtype entity objects.
3. In the Select Parent dialog, select the entity-based view object that you wish to extend and click **OK**.

   The view object you select is the base view object which defines the superset of attributes and is the view object that the subtype view object extends. For example, you select the base view object `CustomerList` to extend when you want to create the subtype view object `DomesticList`, as shown in Figure 7-3.

Figure 7-3 Subtype View Object Extends Base View Object



Package: oracle.summit.model.polymorphicsample

Name: DomesticList

Display Name: Domestic List

Extends:

Property Set:

Data Source:

● Entity

○ Custom SQL

○ Programmatic

○ Static list

Select View Object to be extended:

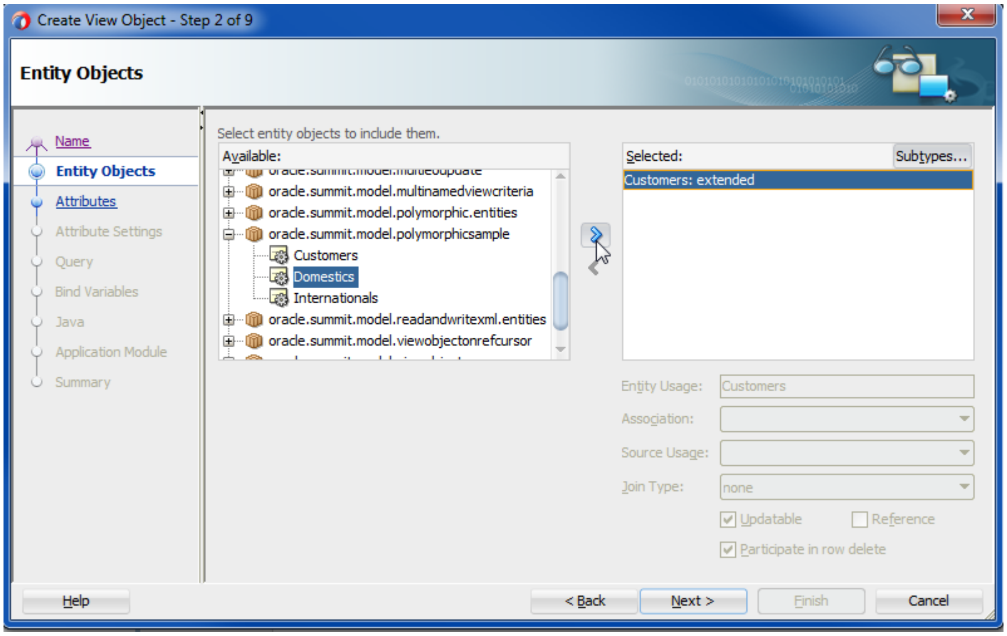
- oracle.summit.model.domains
- oracle.summit.model.extend
- oracle.summit.model.inMemoryOperations
- oracle.summit.model.multieoupdate
- oracle.summit.model.multinamedviewcriteria
- oracle.summit.model.polymorphic.views
- oracle.summit.model.polymorphicsample
  - CustomerList
- oracle.summit.model.readandwritexml.queries
- oracle.summit.model.viewobjectonrefcursor
- oracle.summit.model.viewobjects
- oracle.summit.model.views
- oracle.summit.model.views.readonly
- oracle.summit.model.wrapplsql

4. In the Create View Object wizard, make sure that **Entity** is selected as the data source and click **Next**.

5. In the Entity Objects page, in the **Available** list, locate the entity subtype that you want to define as the polymorphic entity usage for the subtype view object and click **Add**.

   The list of available entity subtypes is defined in the inheritance hierarchy of the base entity object, as explained in the prerequisites for this procedure. For example, when you create the subtype view object `DomesticList` for the base view object `CustomerList`, you would add `Domestics` as the entity subtype from the **Available** list, as shown in Figure 7-4.

Figure 7-4 Subtype View Object with a Polymorphic Entity Usage Selection


Create View Object - Step 2 of 9
Entity Objects
Name
Entity Objects
Attributes
Attribute Settings
Query
Bind Variables
Java
Application Module
Summary
Select entity objects to include them.
Available:
oracle.summit.model.multinamedviewcriteria
oracle.summit.model.polymorphic.entities
oracle.summit.model.polymorphicsample
Customers
Domestics
Internationals
oracle.summit.model.readandwritexml.entities
oracle.summit.model.viewobjectonrefcursor
Selected:
Subtypes...
Customers: extended
Entity Usage:
Association:
Source Usage:
Join Type:
Updatable
Reference
Participate in row delete
Help
< Back
Next >
Finish
Cancel

Entity Usage: Customers

Join Type: none

☑ Updatable ☐ Reference

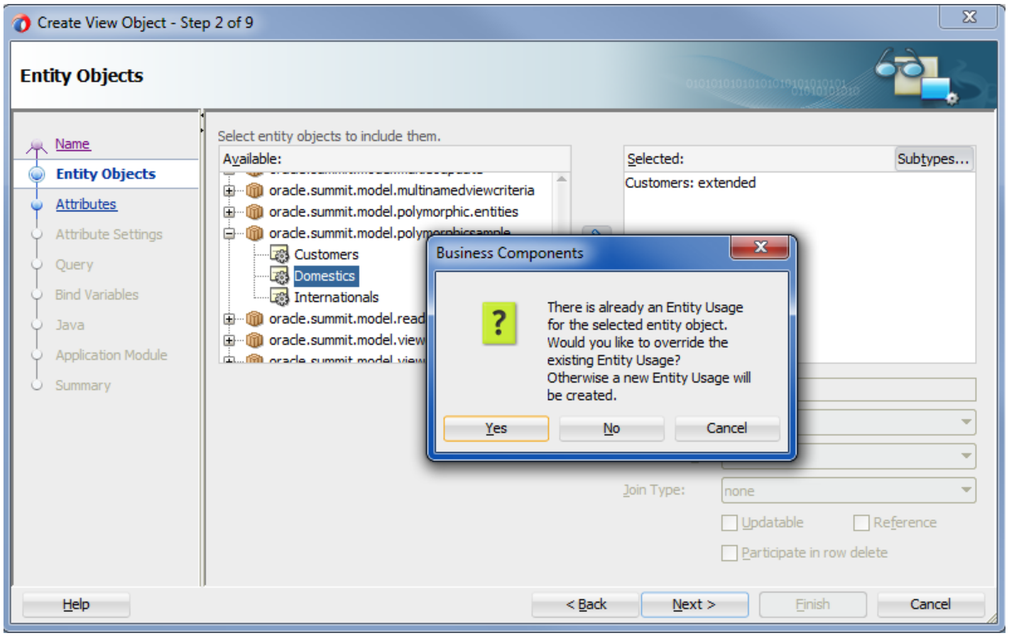
☑ Participate in row delete

6. In the Business Components dialog, click **Yes** to override the entity usage for your subtype view object with the desired polymorphic entity usage.

The Business Components dialog warns you that you will override the attributes of the base entity usage with the entity subtype, as shown in Figure 7-5.

Note that clicking **No** in the dialog will add an additional usage without overriding the attributes of the base entity usage. Normally, you click **Yes** to override the base entity usage with the desired entity subtype.

Figure 7-5 Polymorphic Entity Usage Overrides Subtype View Object



There is already an Entity Usage for the selected entity object. Would you like to override the existing Entity Usage? Otherwise a new Entity Usage will be created.

Join Type: none

☐ Updatable ☐ Reference

☐ Participate in row delete

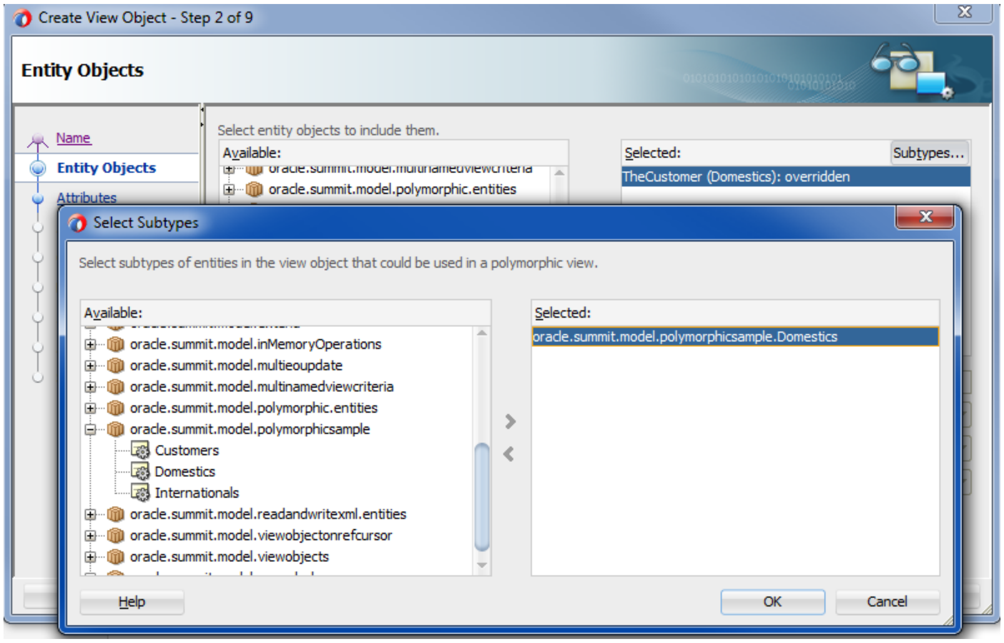
7. In the **Selected** list, select the overridden entity usage and click **Subtypes**.

8. In the Select Subtypes dialog, use the shuttle buttons to leave only the desired polymorphic entity subtype in the **Selected** list and click **OK**.

   The Select Subtypes dialog lets you specify the entity subtype to be referenced by the usages you have defined in the Entity Objects page of the Create View Object wizard. The list of available entity subtypes is defined by the base view object, as explained in the prerequisites for this procedure. Normally, you will pick the same entity subtype as the one you choose to override (as shown in Figure 7-5).

   For example, when you create the subtype view object `DomesticList`, you would select `Domestics` as the single entity subtype of the base entity object `Customers` (and deselect `Internationals` from the **Selected** list), as shown in Figure 7-6.

Figure 7-6 Subtype View Object with a Single Polymorphic Entity Usage Applied



Create View Object - Step 2 of 9

Entity Objects

Select entity objects to include them.

Available:
- oracle.summit.model.polymorphic.entities

Selected:
- TheCustomer (Domestics): overridden

Select Subtypes

Select subtypes of entities in the view object that could be used in a polymorphic view.

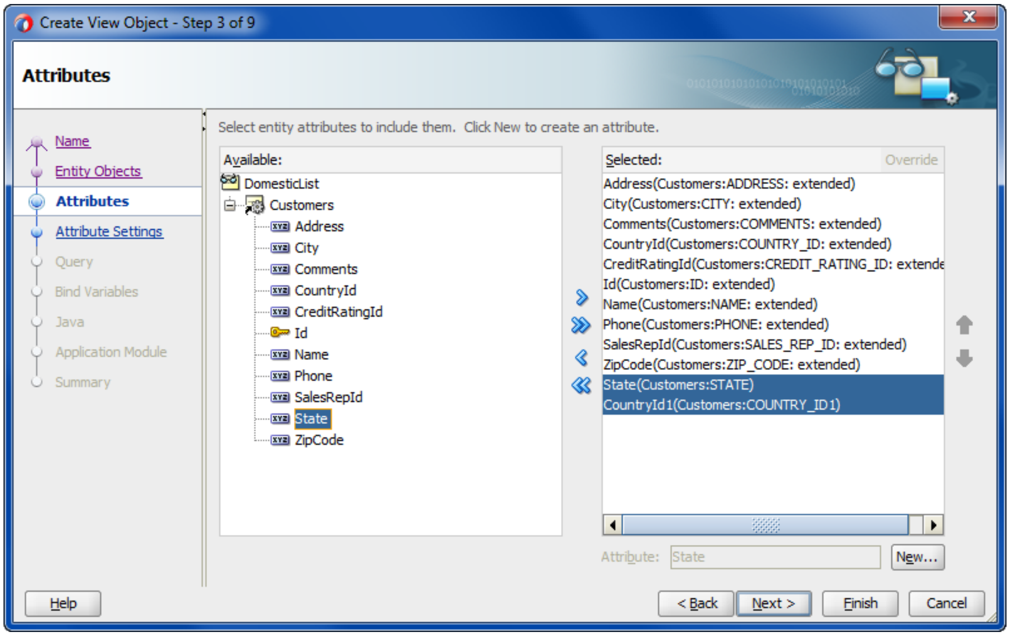
Available:
- oracle.summit.model.inMemoryOperations
- oracle.summit.model.multieoupdate
- oracle.summit.model.multinamedviewcriteria
- oracle.summit.model.polymorphic.entities
- oracle.summit.model.polymorphicsample
  - Customers
  - Domestics
  - Internationals
- oracle.summit.model.readandwritexml.entities
- oracle.summit.model.viewobjectonrefcursor
- oracle.summit.model.viewobjects

Selected:
- oracle.summit.model.polymorphicsample.Domestics

9. In the Create View Object wizard, in the Entity Objects page, click **Next** and on the Attributes page of the wizard, shuttle the desired attributes from the entity subtype to the **Selected** list.

   For example, for the `DomesticList` view object you would select the `State` attribute from the entity subtype `Domestics`, as shown in Figure 7-7.

Figure 7-7 View Object with a Entity Subtype Attribute Addition



Select entity attributes to include them. Click New to create an attribute.

Selected:

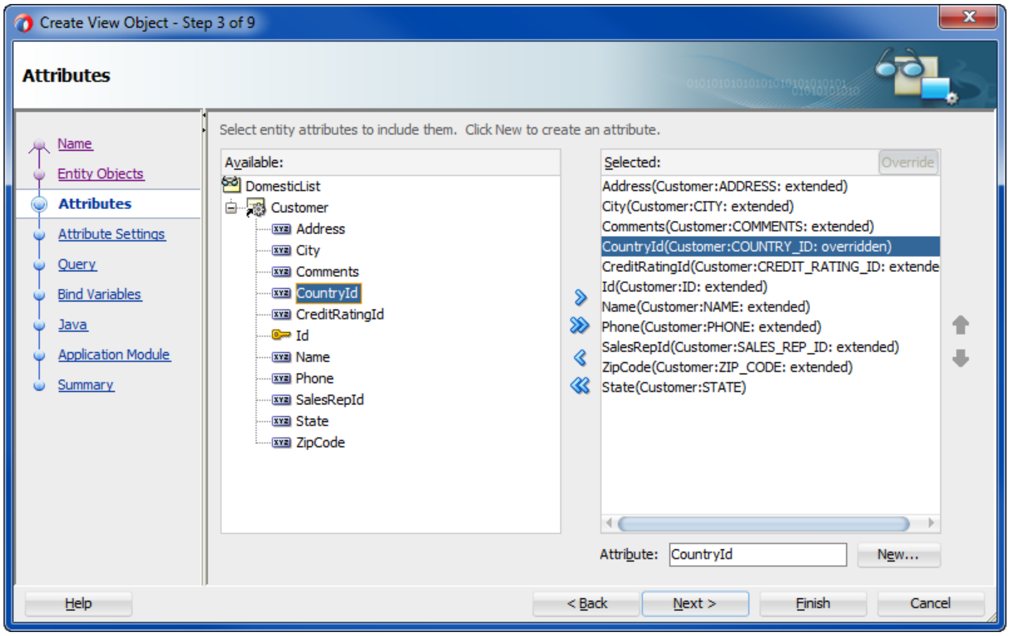
Address(Customers:ADDRESS: extended)
City(Customers:CITY: extended)
Comments(Customers:COMMENTS: extended)
CountryId(Customers:COUNTRY_ID: extended)
CreditRatingId(Customers:CREDIT_RATING_ID: extende
Id(Customers:ID: extended)
Name(Customers:NAME: extended)
Phone(Customers:PHONE: extended)
SalesRepId(Customers:SALES_REP_ID: extended)
ZipCode(Customers:ZIP_CODE: extended)
State(Customers:STATE)
CountryId1(Customers:COUNTRY_ID1)

Attribute: State

10. In the Attributes page, remove the primary key attribute so the extended view object defines a single primary key attribute and then select the discriminator attribute and click **Override** and click **Next**.

    For example, for the `DomesticList` view object you would remove the `CountryId` attribute from the subtype view object's selected list and override the `CountryId` attribute, as shown in Figure 7-8.

Figure 7-8 View Object with the Overridden Discriminator Attribute



Select entity attributes to include them. Click New to create an attribute.

Selected:

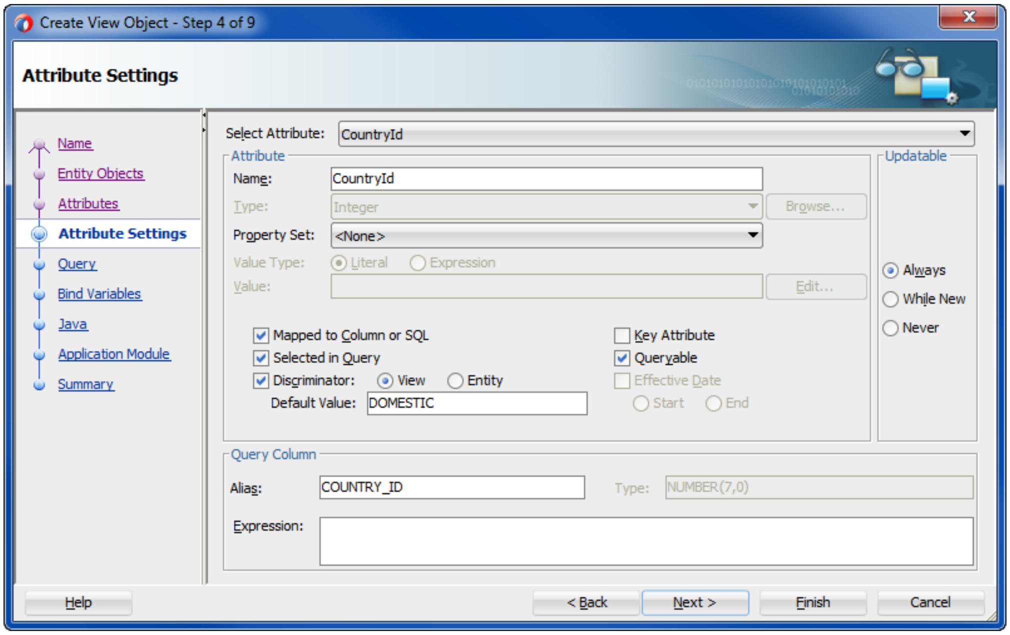
Address(Customer:ADDRESS: extended)
City(Customer:CITY: extended)
Comments(Customer:COMMENTS: extended)
CountryId(Customer:COUNTRY_ID: overridden)
CreditRatingId(Customer:CREDIT_RATING_ID: extende
Id(Customer:ID: extended)
Name(Customer:NAME: extended)
Phone(Customer:PHONE: extended)
SalesRepId(Customer:SALES_REP_ID: extended)
ZipCode(Customer:ZIP_CODE: extended)
State(Customer:STATE)

Attribute: CountryId

11. In the Attributes Settings page, select the discriminator attribute from the dropdown and enter the **Default Value** that defines the subtype and click **Finish** to complete the wizard.

For example, for the `DomesticList` view object you would enter `DOMESTIC`, as shown in Figure 7-9.

Figure 7-9 View Object with the Overridden Discriminator Attribute



Create View Object - Step 4 of 9

Attribute Settings

Select Attribute: CountryId

Attribute

Name: CountryId

Type: Integer

Property Set: <None>

Value Type: ● Literal ○ Expression

Value:

☑ Mapped to Column or SQL

☑ Selected in Query

☑ Discriminator: ● View ○ Entity

Default Value: DOMESTIC

☐ Key Attribute

☑ Queryable

☐ Effective Date ○ Start ○ End

Updatable

● Always

○ While New

○ Never

Query Column

Alias: COUNTRY_ID

Type: NUMBER(7,0)

Expression:

12. Repeat this procedure to create additional subtype view objects with unique polymorphic entity usages.

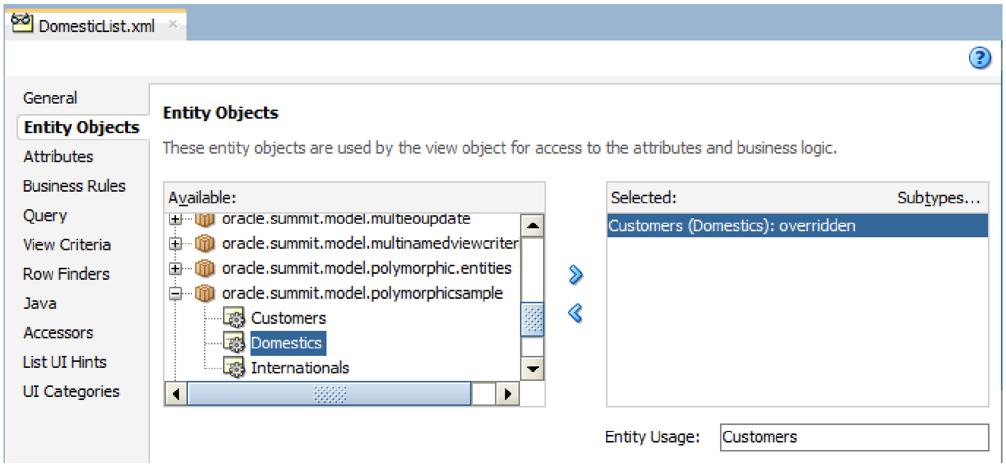
    For example, the `InternationalList` view object is a second subtype that extends `CustomerList`. It supplies the value `INTERNATIONAL` for the `CustomerTypeCode` discriminator attribute.

After creating the subtype view object, you must define the list of subtypes that participate in the view object polymorphism, as described in Updating the Application Module to Expose Subtype Usages. For example, the `CustomersModule` application module with view instance `CustomerList1` must be configured to name the subtype view objects `DomesticList` and `InternationalList`.

What Happens When You Create a Subtype View Object with a Polymorphic Entity Usage

The Entity Objects page of the overview editor identifies the selected entity object with the entity usage override. For example, the overview editor for the `DomesticList` view object identifies the overridden entity usage `Customers (Domestics): overridden` with the overriding subtype in parenthesis, as shown in Figure 7-10.

Figure 7-10 View Object Editor Shows Entity Usage is Overridden



Entity Objects

These entity objects are used by the view object for access to the attributes and business logic.

Customers (Domestics): overridden

Entity Usage: Customers

When you create an entity-based view object with a polymorphic entity usage, JDeveloper adds information about the allowed entity subtypes to the base view object's XML document. For example, when creating the `CustomerList` view object, the names of the allowed subtype entity objects are recorded in an `<AttrArray>` tag. The XML property `DiscrColumn` identifies `CountryTypeCode` as the base view object's polymorphic discriminator attribute like this:

```
<ViewObject Name="CustomerList" ... >
  <EntityUsage
    Name="Customers"
    Entity="oracle.summit.model.polymorphics.entities.Customers" >
  </EntityUsage>
  <AttrArray Name="EntityImports">
    <Item Value="oracle.summit.model.polymorphic.entities.Domestics" />
    <Item Value="oracle.summit.model.polymorphic.entities.Internationals" />
  </AttrArray>
...
  <ViewAttribute
    Name="CountryTypeCode"
    PrecisionRule="true"
    EntityAttrName="CountryTypeCode"
    EntityUsage="Customers"
    AliasName="COUNTRY_TYPE_CODE"
    DiscrColumn="true">
  </ViewAttribute>
   ...
</ViewObject>
```

Then, when you create a subtype view object with a polymorphic entity usage, JDeveloper adds information about the base view object to the subtype view object's XML document. For example, The following example shows the `DomesticList` subtype view object with the name of the base view object recorded in the `Extends` property. The XML property `DiscrColumn` identifies the overridden `CountryTypeCode` attribute as the polymorphic discriminator attribute, with a subtype value of `DOMESTIC`. Only the additional attribute `State`, specific to the subtype view object, appears in the XML document.

```
<ViewObject
  Name="DomesticList"
  Extends="oracle.summit.model.polymorphic.views.CustomerList"
  ...>
  <EntityUsage
    Name="Customers"
    Entity="oracle.summit.model.polymorphic.entities.Domestics"/>
  <AttrArray Name="EntityImports">
    <Item Value="oracle.summit.model.polymorphicsample.Domestics"/>
  </AttrArray>
  <ViewAttribute
    Name="CountryTypeCode"
    PrecisionRule="true"
    DiscrColumn="true"
    EntityAttrName="CountryTypeCode"
    EntityUsage="Customers"
    AliasName="COUNTRY_TYPE_CODE"
    DefaultValue="DOMESTIC">
    <DesignTime>
      <Attr Name="_OverrideAttr" Value="true"/>
    </DesignTime>
  </ViewAttribute>
  <ViewAttribute
    Name="State"
    PrecisionRule="true"
    EntityAttrName="State"
    EntityUsage="Customers"
    AliasName="STATE"/>
</ViewObject>
```

Similarly, the XML document for the `InternationalList` subtype view object names the same `CustomerList` base view object in the `Extends` property of its XML document, but defines the polymorphic discriminator attribute with a subtype value of `INTERNATIONAL` and the additional attribute `Language` specific to its subtype like this:

```
<ViewObject
  Name="InternationalList"
  Extends="oracle.summit.model.polymorphic.views.CustomerList"
  ... >
  <EntityUsage
     Name="Customers"
     Entity="oracle.summit.model.polymorphic.entities.Internationals"/>
  <AttrArray Name="EntityImports">
    <Item Value="oracle.summit.model.polymorphicsample.Internationals"/>
  </AttrArray>
  <ViewAttribute
    Name="CountryTypeCode"
    PrecisionRule="true"
    DiscrColumn="true"
    EntityAttrName="CountryTypeCode"
    EntityUsage="Customers"
    AliasName="COUNTRY_TYPE_CODE"
    DefaultValue="INTERNATIONAL">
    <DesignTime>
      <Attr Name="_OverrideAttr" Value="true"/>
    </DesignTime>
  </ViewAttribute>
  <ViewAttribute
    Name="Language"
    PrecisionRule="true"
    EntityAttrName="Language"
    EntityUsage="Customers"
    AliasName="LANGUAGE"/>
</ViewObject>
```

Note the <AttrArray> tag in the subtype view object definition identifies the allowed entity object for each subtype. Each subtype view object should therefore name a single subtype entity object. If the <AttrArray> tag of a subtype view object definition names more than one entity object, you may remove the unused one. This situation arises when using the Create View Object wizard to create the subtype view object but not reducing the list of default subtype entity objects in the Select Subtypes dialog, as shown in Figure 7-6. By default the dialog makes all subtype entity objects selected.

What You May Need to Know About Polymorphic Entity Usages

When you derive view objects from polymorphic entity usages, you will want to follow these best practices to allow the discriminator attribute to filter the query to only contain the expected subtypes.

Your Query Must Limit Rows to Expected Entity Subtypes

If your view object expects to work with only a *subset* of the available entity subtypes in a hierarchy, you need to include an appropriate WHERE clause that limits the query to only return rows whose discriminator column matches the expected entity type. You should not rely solely on the discriminator attribute to perform subtype filtering in the query. Without the addition of attributes to limit the query, the view object will effectively query all rows and cause the client to discard rows that do not match any discriminator values.

Exposing Selected Entity Methods in View Rows Using Delegation

By design, clients do not work directly with entity objects. Instead, they work indirectly with entity objects through the view rows of an appropriate view object that presents a relevant set of information related to the task as hand. Just as a view object can expose a particular set of the underlying *attributes* of one or more entity objects related to the task at hand, it can also expose a selected set of *methods* from those entities. You accomplish this by enabling a custom view row Java class and writing a method in the view row class that:

- Accesses the appropriate underlying entity row using the generated entity accessor in the view row, and
- Invokes a method on it

For example, assume that the Customers entity object contains a performCustomerFeature() method in its CustomersImpl class. To expose this method to clients on the CustomerList view row, you can enable a custom view row Java class and write the method, as shown in the following example.

```
// In CustomerListRowImpl.java
public void performCustomerFeature() {
  getTheCustomer().performCustomerFeature();
}
```

JDeveloper generates an entity accessor method in the view row class for each participating entity usage based on the entity usage *alias* name. Since the alias for the Customers entity in the CustomerList view object is "TheCustomer", it generates a getTheCustomer() method to return the entity row part related to that entity usage.

The code in the view row's `performCustomerFeature()` method uses this `getTheCustomer()` method to access the underlying `CustomersImpl` entity row class and then invokes *its* `performCustomerFeature()` method. This style of coding is known as **delegation**, where a view row method delegates the implementation of one of *its* methods to a corresponding method on an underlying entity object. When delegation is used in a view row with a polymorphic entity usage, the delegated method call is handled by appropriate underlying entity row subtype. This means that if the `CustomersImpl`, `DomesticsImpl`, and `InternationalsImpl` classes implement the `performCustomerFeature()` method in a different way, the appropriate implementation is used depending on the entity subtype for the current row.

After exposing this method on the client row interface, client programs can use the custom row interface to invoke custom business functionality on a particular view row. The following example shows the interesting lines of code from a `TestEntityPolymorphism` class. It iterates over all the rows in the `CustomerList` view object instance, casts each one to the custom `CustomerListRow` interface, and invokes the `performCustomerFeature()` method.

```
CustomerList customerlist = (CustomerList)am.findViewObject("CustomerList");
customerlist.executeQuery();
while (customerlist.hasNext()) {
  CustomerListRow customer = (CustomerListRow)customerlist.next();
  System.out.print(customer.getEmail()+"->");
  customer.performCustomerFeature();
}
```

Running the client code in the previous example produces the following output:

```
austin->## performCustomerFeature as Domestics
hbaer->## performCustomerFeature as Internationals
:
sking->## performCustomerFeature as Domestic
:
```

Rows related to `Customers` entities display a message confirming that the `performCustomerFeature()` method in the `CustomersImpl` class was used. Rows related to `Domestics` and `Internationals` entities display a different message, highlighting the different implementations that the respective `DomesticsImpl` and `InternationalsImpl` classes have for the inherited `performCustomerFeature()` method.

Creating New Rows With the Desired Entity Subtype

In a view object with a polymorphic entity usage, when you create a new view row it contains a new entity row part whose type matches the base entity usage. To create a new view row with one of the entity *subtypes* instead, use the `createAndInitRow()` method.

The following example shows two custom methods in the `CustomerList` view object's Java class that use `createAndInitRow()` to allow a client to create new rows having entity rows either of `Domestics` or `Internationals` subtypes. To use the `createAndInitRow()`, as shown in the example, create an instance of the `NameValuePairs` object and set it to have an appropriate value for the discriminator attribute. Then, pass that `NameValuePairs` to the `createAndInitRow()` method to create a new view row with the appropriate entity row subtype, based on the value of the discriminator attribute you passed in.

```
// In CustomerListImpl.java
public CustomerListRow createCustomersRow() {
  NameValuePairs nvp = new NameValuePairs();
  nvp.setAttribute("CustomerTypeCode","DOMESTIC");
  return (CustomerListRow)createAndInitRow(nvp);
}
public CustomerListRow createInternationalsRow() {
  NameValuePairs nvp = new NameValuePairs();
  nvp.setAttribute("CustomerTypeCode","INTERNATIONAL");
  return (CusotmersListRow)createAndInitRow(nvp);
}
```

If you expose methods like this on the view object's custom interface, then at runtime, a client can call them to create new view rows with appropriate entity subtypes. The following example shows the interesting lines relevant to this functionality from a `TestEntityPolymorphism` class. First, it uses the `createRow()`, `createDomesticsRow()`, and `createInternationalsRow()` methods to create three new view rows. Then, it invokes the `performCustomerFeature()` method from the `CustomerListRow` custom interface on each of the new rows.

```
// In TestEntityPolymorphism.java
CustomerListRow newCustomer = (CustomerListRow)CustomerList.createRow();
CustomerListRow newDomestic  = Customerlist.createDomesticsRow();
CustomerListRow newInternational = Customerlist.createInternationalsRow();
newCustomer.performCustomerFeature();
newDomestic.performCustomerFeature();
newInternational.performCustomerFeature();
```

As expected, each row handles the method in a way that is specific to the subtype of entity row related to it, producing the results:

```
## performCustomerFeature as Customer
## performCustomerFeature as Domestic
## performCustomerFeature as International
```

Working with Polymorphic View Rows

ADF Business Components allows you to configure a view object to have polymorphic view rows. This allows the client to access different view row types with specific view row interfaces.

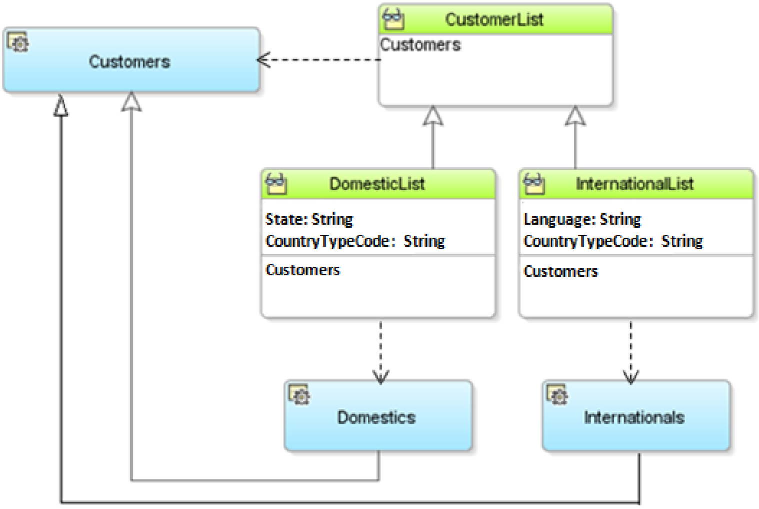
In the example shown in Working with Polymorphic Entity Usages, the polymorphism occurs "behind the scenes" at the entity object level. Since the client code works with all view rows using the same `CustomerListRow` interface, it cannot distinguish between rows based on a `Domestics` entity object from those based on a `Customers` entity object. The code works with all view rows using the same set of view row attributes and methods common to all types of underlying entity subtypes.

If you configure a view object to support polymorphic view rows, then the client can work with different types of view rows using a view row interface specific to the type of row it is. By doing this, the client can access view attributes or invoke view row methods that are specific to a given subtype as needed. Figure 7-11 illustrates the hierarchy of view objects that enables this feature for the `CustomerList` example considered above. `DomesticList` and `InternationalList` are child view objects that extend the base (or parent) `CustomerList` view object. Notice that each one includes an additional attribute specific to the subtype of `Customers` they have as their entity usage. `DomesticList` includes an additional `State` attribute, while `InternationalList` excludes the `State` attribute and includes the `Language` attribute. When configured for view row polymorphism, a client can work with the results of the `CustomerList` view object using:

- `CustomerListRow` interface for view rows related to customers
- `DomesticListRow` interface for view rows related to domestic customers
- `InternationalListRow` interface for view rows related to international customers

As you'll see, this allows the client to access the additional attributes and view row methods that are specific to a given subtype of view row.

Figure 7-11 Hierarchy of View Object Subtypes Enables View Row Polymorphism



Note:

The example in this section refers to the `oracle.summit.model.polymorphicvo` package in the `SummitADF_Examples` application workspace.

How to Create a View Object with Polymorphic View Rows

The view object that you create with polymorphic view rows may inherit one or more of the attributes from a hierarchy of view objects, each with their own base entity object. Attributes that you select from the extended view objects will be overridden by the polymorphic view row definitions in the parent view object.

To create a view object with polymorphic view rows:

1. In the Application Navigator, double-click the view object that you want to be the base view object in the inheritance hierarchy.

   For example, the `CustomerList` view object is the base for `DomesticList` view object.

2. In the overview editor for the view object, click the **Attributes** navigation tab and select the discriminator attribute that distinguishes which view row interface to use for the view row, and then click the **Edit selected attribute(s)** button.

   For example, the `CustomerTypeCode` is the discriminator attribute for `CustomerList`.

3. In the Edit Attribute dialog, select the **Discriminator** checkbox, enter a **Default Value**, and click **OK**.

   You must supply a value for the **Default Value** field that matches the attribute value for which you expect the base view object's view row interface to be used. For example, in the `CustomerList` view object, you would mark the `CustomerTypeCode` attribute as the discriminator attribute and supply a default value of `CUSTOMER`.

4. In the overview editor for the view object, in the Attributes page, delete all attributes that are specific to the subtype view object.

   For example, the attribute `States` would be removed from the base view object since it is used exclusively by the `DomesticList` subtype view object.

5. In the overview editor, click the **Java** navigation tab, and enable a custom view row class for the base view object, and expose at least one method on the client row interface. This can be one or all of the view row attribute accessor methods, as well as any custom view row methods.

6. Create the subtype new view object that *extends* the base view object:

   a. In the Application Navigator, right-click the base view object and choose **New Extended Object**, and in the Create Extended Object dialog, name the subtype view object.

      In this example, the `CustomerList` view object is the base and the subtype view object is named `DomesticList`.

   b. In the overview editor for the subtype view object, click the **Attributes** navigation tab and choose **Add Attribute From Entity** from the **Add** button dropdown, and then, in the Attributes dialog, add the attributes that are specific to the subtype view object and click **OK**.

      In this example, the subtype `DomesticList` defines the additional attribute `State`.

   c. In the Attributes page of the overview editor, select the discriminator attribute for the view row, and click **Override** and then click the **Edit selected attribute(s)** button.

   d. In the Edit Attribute dialog, give the discriminator attribute a *distinct* **Default Value** to define the subtype of the extended view object and click **OK**.

      For example, the `DomesticList` view object provides the value `DOMESTIC` for the `CustomerTypeCode` discriminator attribute.

   e. In the overview editor, click the **Java** navigation tab, and enable a custom view row class for the extended view object.

      If appropriate, add additional custom view row methods or override custom view row methods inherited from the parent view object's row class.

   f. Repeat to add additional extended view objects as needed.

For example, the InternationalList view object is a second one that extends CustomerList. It supplies the value INTERNATIONAL for the CustomerTypeCode discriminator attribute.

After setting up the view object hierarchy, you must define the list of view object subtypes that participate in the view row polymorphism, as described in Updating the Application Module to Expose Subtype Usages. For example, the CustomersModule application module with view instance CustomerList1 must be configured to name the subtype view objects DomesticList and InternationalList.

What Happens When You Create a View Object With Polymorphic View Rows

When you create the base view object that you use to define child view objects with polymorphic view row definitions, JDeveloper adds information about the discriminator attribute to the base view object's XML document. For example, when creating the CustomerList view object, the XML property DiscrColumn identifies CountryTypeCode as the base view object's polymorphic discriminator attribute, with a subtype value of CUSTOMER like this:

```
<ViewObject
  Name="CustomerList" ... >
  <EntityUsage
    Name="Customers"
    Entity="oracle.summit.model.polymorphics.entities.Customers" >
  </EntityUsage>
...
  <ViewAttribute
    Name="CountryTypeCode"
    PrecisionRule="true"
    EntityAttrName="CountryTypeCode"
    EntityUsage="Customers"
    AliasName="COUNTRY_TYPE_CODE"
    DiscrColumn="true"
    DefaultValue="CUSTOMER">
  </ViewAttribute>
   ...
</ViewObject>
```

Then, when you create a child view object with a polymorphic view row definition, JDeveloper adds information about the view row definition to the child view object's XML document. For example, The following example shows the DomesticList child view object with the name of the base view object recorded in the Extends property. The XML property DiscrColumn identifies the overridden CountryTypeCode attribute as the polymorphic discriminator attribute, with a subtype value of DOMESTIC. Only the additional attribute State, specific to the subtype, appears in the XML document of the child view object.

```
<ViewObject
  Name="DomesticList"
  Extends="oracle.summit.model.polymorphic.views.CustomerList"
  ...>
  <ViewAttribute
    Name="CountryTypeCode"
    PrecisionRule="true"
    DiscrColumn="true"
    EntityAttrName="CountryTypeCode"
    EntityUsage="Customers"
    AliasName="COUNTRY_TYPE_CODE"
    DefaultValue="DOMESTIC">
    <DesignTime>
      <Attr Name="_OverrideAttr" Value="true"/>
    </DesignTime>
  </ViewAttribute>
  <ViewAttribute
    Name="State"
    PrecisionRule="true"
    EntityAttrName="State"
    EntityUsage="Customers"
    AliasName="STATE"/>
</ViewObject>
```

Similarly, the XML document for the InternationalList child view object names the same base view object in the Extends property of its XML document, but defines the polymorphic discriminator attribute with a subtype value of INTERNATIONAL and the additional attribute Language specific to its subtype like this:

```
<ViewObject
  Name="InternationalList"
  Extends="oracle.summit.model.polymorphic.views.CustomerList"
  ... >
  <ViewAttribute
    Name="CountryTypeCode"
    PrecisionRule="true"
    DiscrColumn="true"
    EntityAttrName="CountryTypeCode"
    EntityUsage="Customers"
    AliasName="COUNTRY_TYPE_CODE"
    DefaultValue="INTERNATIONAL">
    <DesignTime>
      <Attr Name="_OverrideAttr" Value="true"/>
    </DesignTime>
  </ViewAttribute>
  <ViewAttribute
    Name="Language"
    PrecisionRule="true"
    EntityAttrName="Language"
    EntityUsage="Customers"
    AliasName="LANGUAGE"/>
</ViewObject>
```

Note that unlike the polymorphic entity usage example described in What Happens When You Create a Subtype View Object with a Polymorphic Entity Usage, view instances with polymorphic view rows use inheritance and overriding attributes to distinguish, for example, between rows based on a Domestics entity object from those based on a Customers entity object. Thus, the definition of allowed subtype entity objects is absent from the XML document of the view objects in the polymorphic view row case (but defined in <AttrArray> tags in the polymorphic entity usage case).

What You May Need to Know About Polymorphic View Rows

When you work with polymorphic view rows, you can interact with the row types to customize the attributes to display or to delegate to methods specific to the entity subtypes of the view rows.

Selecting Subtype-Specific Attributes in Extended View Objects

When you create an extended view object, it inherits the entity usage of its parent. If the parent view object's entity usage is based on an entity object with subtypes in your domain layer, you may want your extended view object to work with one of these subtypes instead of the inherited parent entity usage type. Two reasons you might want to do this are:

- To select attributes that are *specific* to the entity subtype
- To be able to write view row methods that delegate to methods *specific* to the entity subtype

In order to do this, you need to override the inherited entity usage to refer to the desired entity subtype. To do this, perform these steps in the overview editor for your extended view object.

To override the entity usage for the view object:

1. In the Applications window, double-click the view object that contains the entity usage that you want to override.
2. In the overview editor, click the **Entity Objects** navigation tab and verify that you are working with an extended entity usage.

   For example, when creating the `DomesticList` view object that extends the `DomesticList` view object, the entity usage with the alias `TheCustomer` will initially display in the **Selected** list as: **TheCustomer(Customers): extended**. The type of the entity usage is in parenthesis, and the "extended" label confirms that the entity usage is currently inherited from its parent.
3. Select the desired entity *subtype* in the **Available** list that you want to override the inherited one. It must be a subtype entity of the existing entity usage's type.

   For example, you would select the `Domestics` entity object in the **Available** list to override the inherited entity usage based on the `Customers` entity type.
4. Click **>** to shuttle it to the **Selected** list
5. Acknowledge the alert that appears, confirming that you want to override the existing, inherited entity usage.

When you have performed these steps, the **Selected** list updates to reflect the overridden entity usage. For example, for the `DomesticList` view object, after overriding the `Customers`-based entity usage with the `Domestics` entity subtype, it updates to show: **TheCustomer (Domestics): overridden**.

After overriding the entity usage to be related to an entity subtype, you can then use the **Attributes** tab of the editor to select additional attributes that are specific to the subtype. For example, the `DomesticsList` view object includes the additional attribute named `State` that is specific to the `Domestics` entity object.

Delegating to Subtype-Specific Methods After Overriding the Entity Usage

After overriding the entity usage in an extended view object to reference a subtype entity, you can write view row methods that delegate to methods specific to the subtype entity class.

The following example shows the code for a `performInternationalFeature()` method in the custom view row class for the `InternationalList` view object. It casts the return value from the `getTheCustomer()` entity row accessor to the subtype `InternationalsImpl`, and then invokes the `performInternationalFeature()` method that is specific to `Internationals` entity objects.

```
// In InternationalListRowImpl.java
public void performInternationalFeature() {
   InternationalsImpl international = (InternationalsImpl)getTheCustomer();
   international.performInternationalFeature();
}
```

> **Note:**
>
> You need to perform the explicit cast to the entity subtype here because JDeveloper does not yet take advantage of the JDK feature called covariant return types that would allow a subclass like `InternationalListRowImpl` to override a method like `getTheCustomer()` and change its return type.

Working with Different View Row Interface Types in Client Code

The following example shows the interesting lines of code from a `TestViewRowPolymorphism` class that performs the following steps:

1. Iterates over the rows in the `CustomerList` view object.

   For each row in the loop, it uses Java's `instanceof` operator to test whether the current row is an instance of the `DomesticListRow` or the `InternationalListRow`.

2. If the row is a `DomesticListRow`, then cast it to this more specific type and:
   - Call the `performDomesticFeature()` method specific to the `DomesticListRow` interface, and
   - Access the value of the `State` attribute that is specific to the `DomesticList` view object.

3. If the row is a `InternationalListRow`, then cast it to this more specific type and:
   - Call the `performInternationalFeature()` method specific to the `InternationalListRow` interface, and
   - Access the value of the `Language` attribute that is specific to the `InternationalList` view object.

4. Otherwise, just call a method on the `CustomerListRow`

```
// In TestViewRowPolymorphism.java
ViewObject vo = am.findViewObject("CustomerList");
vo.executeQuery();
// 1. Iterate over the rows in the CustomerList view object
while (vo.hasNext()) {
  CustomerListRow Customer = (CustomerListRow)vo.next();
  System.out.print(Customer.getEmail()+"->");
  if (Customer instanceof DomesticListRow) {
    // 2. If the row is a DomesticListRow, cast it
    DomesticListRow mgr = (DomesticListRow)Customer;
    mgr.performDomesticFeature();
    System.out.println("State: "+domestic.getState());
  }
  else if (Customer instanceof InternationalListRow) {
    // 3. If the row is an InternationalListRow, cast it
    InternationalListRow international = (InternationalListRow)Customer;
    international.performInternationalFeature();
    System.out.println("Speaks English: "+international.getLanguage());
  }
  else {
    // 4. Otherwise, just call a method on the CustomerListRow
    Customer.performCustomerFeature();
  }
}
```

Running the code in the previous example produces the following output:

```
daustin->## performInternationalFeature called
English spoken: Yes
hbaer->## performCustomerFeature as Customer
:
sking->## performDomesticFeature called
State: CA
:
```

This illustrates that by using the view row polymorphism feature the client was able to distinguish between view rows of different types and access methods and attributes specific to each subtype of view row.

What You May Need to Know About the Discriminator Attribute

You can choose to include or exclude a discriminator attribute in the query that an ADF view object uses to reference an entity object.

When a view object references an entity object with a discriminator attribute, you can decide whether or not that discriminator attribute should be included in the query (in addition to the primary key attribute) by selecting the **Polymorphic Discriminator** checkbox for the inherited discriminator attribute in the **Details** tab on the Attributes page of the overview editor for the view object.

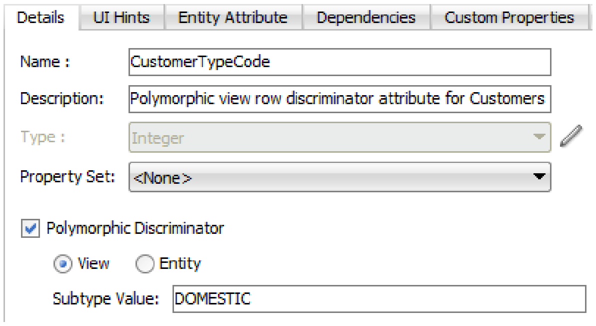
When you decide to enable the checkbox, you can then use the **Polymorphic Discriminator** options in the **Details** tab to configure whether or not the view object is defined as a subtype view object:

- Enabling the option **View** means you want to enable polymorphic view rows.
- Enabling the option **Entity** means you want to ignore the discriminator capability of the view row and will instead instantiate the attribute with a default value.

  The **Entity** option is useful in situations where a discriminator attribute is inherited by an entity-based view object with multiple backing entity objects and yet polymorphic view rows are not desired.

Figure 7-12 shows the Details tab for the DomesticList view object enables its CustomerTypeCode discriminator attribute inherited from the Customers entity object. The **View** selection indicates that this discriminator attribute participates in view object polymorphism and the value DOMESTIC determines the subtype view object's row type.

Figure 7-12 Discriminator Attribute Selection Enables View Row Polymorphism



Name : CustomerTypeCode

Description: Polymorphic view row discriminator attribute for Customers

Type : Integer

Property Set: <None>

☑ Polymorphic Discriminator

● View ○ Entity

Subtype Value: DOMESTIC

Updating the Application Module to Expose Subtype Usages

You can specify the list of view object subtypes for the base view object instance in an ADF application module definition.

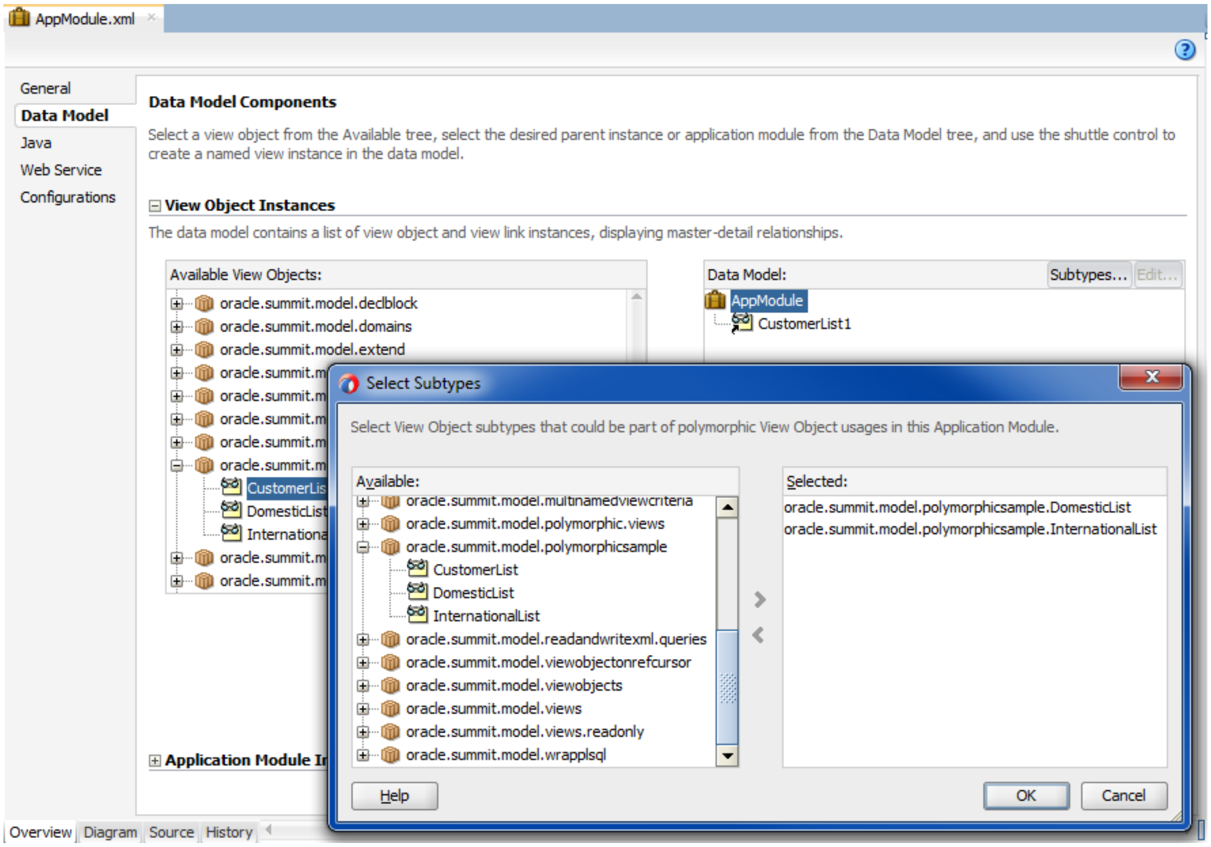
After creating the view object hierarchy using either view row polymorphism or entity object polymorphism, you need to define the list of view object subtypes that participate in the view object polymorphism at runtime. To accomplish this, you update the application module definition to specify the subtype view objects for the base view instance. At runtime, ADF creates polymorphic view rows based on the definitions of the invoked view instance's named subtypes.

How to Expose Subtype Usages on the Data Model

After creating the view object hierarchy using either view row polymorphism or entity object polymorphism, you need to define the list of view object subtypes that participate in the view object polymorphism. To accomplish this, you update the application module definition to specify the subtype view objects for the base view instance.

As Figure 7-13 shows, the data model that exposes the base view object usage `CustomerList1` remains unchanged, while the Select Subtypes dialog is used to define the possible polymorphic view object usages of the base view object.

Figure 7-13   Updating the Application Module Definition with Subtype View Objects



Select a view object from the Available tree, select the desired parent instance or application module from the Data Model tree, and use the shuttle control to create a named view instance in the data model.

The data model contains a list of view object and view link instances, displaying master-detail relationships.

Select View Object subtypes that could be part of polymorphic View Object usages in this Application Module.

Available:
oracle.summit.model.multinamedviewcriteria
oracle.summit.model.polymorphic.views
oracle.summit.model.polymorphicsample
CustomerList
DomesticList
InternationalList
oracle.summit.model.readandwritexml.queries
oracle.summit.model.viewobjectonrefcursor
oracle.summit.model.viewobjects
oracle.summit.model.views
oracle.summit.model.views.readonly
oracle.summit.model.wrapplsql
Selected:
oracle.summit.model.polymorphicsample.DomesticList
oracle.summit.model.polymorphicsample.InternationalList
Help
OK
Cancel
Overview
Diagram
Source
History

Before you begin:

It may be helpful to have an understanding of polymorphic view objects. For more information, see Working with Polymorphic View Rows.

When you want to create subtype view objects based on polymorphic entity usages, which the view objects derive from the inheritance of the entity subtypes, create the subtype view objects that extend the base entity-based view object. For more information, see Working with Polymorphic Entity Usages.

When you want to create subtype view objects based on polymorphic view rows, which derive from a view object hierarchy that you define, create the subtype view objects with polymorphic discriminator attribute. For more information, see Working with Polymorphic View Rows.

To add subtype view objects to the application module definition:

1. In the overview editor for the application module, click the **Data Model** navigation tab and add an instance of the base view object to the data model of an application module.

   For example, when the `CustomerList` view object supports two subtypes: `DomesticList` and `InternationalList` view objects, the `AppModule` application module will have base view instance `CustomerList1`.

2. In the Data Model page, expand the **View Object Instances** section and in the **Data Model** list, click the **Subtypes** button.

You do not need to make a view instance selection in **Data Model** list to click the button.

3. In the **Subtypes** dialog, shuttle the desired view object subtypes of the base view instance from the **Available** to the **Selected** list, and click **OK**.

   For example, for the `CustomerList1` view instance with subtype view objects `DomesticList` and `InternationalList`, you would shuttle both view objects to the **Selected** list, as shown in Figure 7-13.

What Happens When You Add Subtype View Objects to the Application Module

When you add a subtype view object to the application module, JDeveloper adds information about the allowed view subtypes to the application module's XML document. For example, when defining a view instance `CustomerList1` with two possible subtype usages, the names of the allowed subtype view objects are recorded in an `<AttrArray>` tag like this:

```
<AppModule
  ...
  <ViewUsage
    Name="CustomerList1"
    ViewObjectName="oracle.summit.model.polymorphicsample.CustomerList"/>
  <AttrArray Name="ViewImports">
    <Item Value="oracle.summit.model.polymorphicsample.DomesticList"/>
    <Item Value="oracle.summit.model.polymorphicsample.InternationalList"/>
  </AttrArray>
</AppModule>
```

8

Testing View Instance Queries

This chapter describes how to interactively test ADF view objects query results using the Oracle ADF Model Tester provided in JDeveloper. This chapter also explains how to use the ADF Business Components API to access view object instances in a test client outside of JDeveloper.
This chapter includes the following sections:

About View Instance Queries

Use the Oracle ADF Model Tester to achieve interactive ADF application module testing of the data model.

JDeveloper includes an interactive **application module** testing tool that you can use to test all aspects of its data model without having to use your application user interface or write a test client program. Running the Oracle ADF Model Tester can often be the quickest way of exercising the data functionality of your business service during development.

> **Note:**
>
> When you want to test an application module programmatically, you can write a test client. See How to Create a Command-Line Java Test Client.

View Instance Use Cases and Examples

Using the Oracle ADF Model Tester, you can simulate an end user interacting with your **application module data model** before you have started to build any custom user interface of your own. Even *after* you have your UI pages constructed, you will come to appreciate using the Oracle ADF Model Tester to assist in diagnosing problems when they arise. You can reproduce the issues in the Oracle ADF Model Tester to discover if the issue lies in the view or controller layers of the application, or is instead a problem in the business service layer application module itself.

Additional Functionality for Testing View Instances

You may find it helpful to understand other **Oracle ADF** features before you start working with view instances. Following are links to other functionality that may be of interest.

- For details about using debugging tools when you run the Oracle ADF Model Tester, see Using the Oracle ADF Model Tester for Testing and Debugging.
- For details about creating a data model consisting of **view object instances**, see Implementing Business Services with Application Modules.
- For a quick reference to the most common code that you will typically write, use, and override in your custom **ADF Business Components** classes, see Most Commonly Used ADF Business Components Methods.
- For API documentation related to the `oracle.jbo` package, see the following Javadoc reference document:
  - *Java API Reference for Oracle ADF Model*

Creating an Application Module to Test View Instances

Define an ADF application module and build a data model using view object instances. Use the Oracle ADF Model Tester to test the application module if required.

Before you can test view objects that you create in your data model project, you must create an application module where you will define instances of the view objects you want to test. The application module is the transactional component that the Oracle ADF Model Tester (or UI client) will use to work with application data. The set of view objects used by an application module defines its **data model**, in other words, the set of data that a client can display and manipulate through a user interface.

To test the view objects you added to an application module, use the Oracle ADF Model Tester, which is accessible from the Applications window. For details about using the Oracle ADF Model Tester, see Testing View Object Instances Using the Oracle ADF Model Tester.

How to Create the Application Module with Individual View Object Instances

To create an application module that will define instances of individual view objects, use the Create Application Module wizard, which is available in the New Gallery.

Before you begin:

It may be helpful to have an understanding of application modules. For more information, see Creating an Application Module to Test View Instances.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Testing View Instances.

You will need to complete this task:

> Create the desired view objects, as described in How to Create an Entity-Based View Object and How to Create a Custom SQL Mode View Object.

To create an application module to test individual view object instances:

1. In the Applications window, right-click the project in which you want to create the application module and choose **New** and then **Application Module**.
2. In the Create Application Module wizard, in the Name page, provide a package name and an application module name. Click **Next**.

3. On the Data Model page, include instances of the view objects you have previously defined and edit the view object instance names to be exactly what you want clients to see. Then click **Finish**.

   Instead of accepting the default instance name shown in the Data Model page, you can change the instance name to something more meaningful (for example, instead of the default name `OrderItems1` you can rename it to `AllOrderItems`).

How to Create the Application Module with Master-Detail View Object Instances

You can also use the Create Application Module wizard to create a hierarchy of view objects for an application module, based on a **master-detail relationship** that the view objects represent.

Before you begin:

It may be helpful to have an understanding of application modules. For more information, see Creating an Application Module to Test View Instances.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Testing View Instances.

You will need to complete this task:

> Create hierarchical relationships between view objects, as described in Working with Multiple Tables in a Master-Detail Hierarchy.

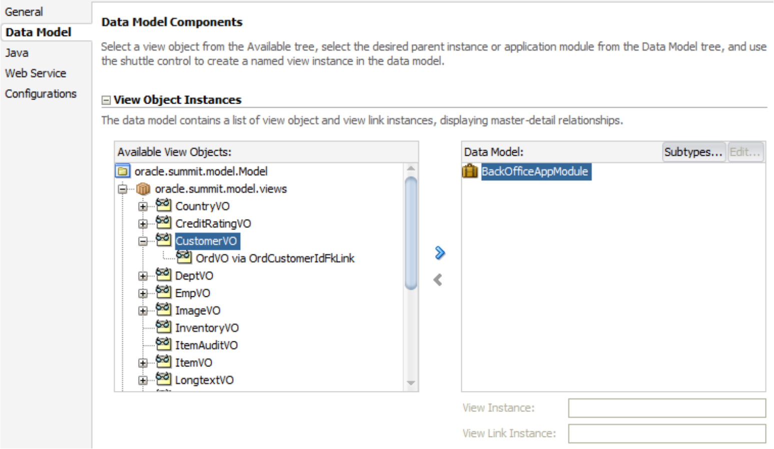
To create an application module based on view object relationships:

1. In the Applications window, right-click the project in which you want to create the application module and choose **New**.
2. In the New Gallery, expand **Business Tier**, select **ADF Business Components** and then **Application Module**, and click **OK**.
3. In the Create Application Module wizard, select the **Data Model** node.
4. In the **Available View Objects** list, select the instance of the view object that you want to be the actively coordinating master.

   The master view object will appear with a plus sign in the list indicating the available view links for this view object. The **view link** must exist to define a master-detail hierarchy.

   For example, Figure 8-1 shows `PersonsVO` selected and renamed `AuthenticatedUser` in the **New View Instance** field.

Figure 8-1 Master View Object Selected



Data Model Components

Select a view object from the Available tree, select the desired parent instance or application module from the Data Model tree, and use the shuttle control to create a named view instance in the data model.

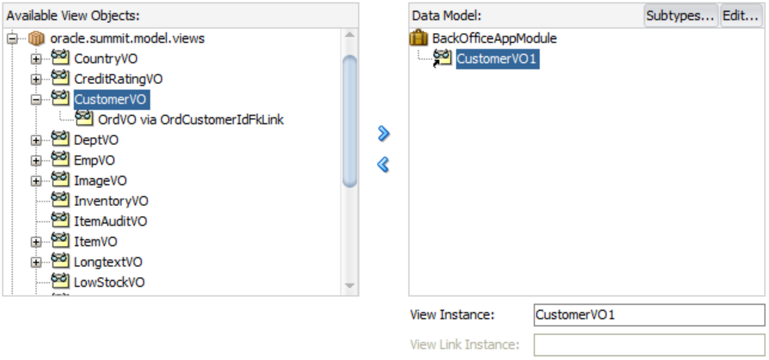
View Object Instances

The data model contains a list of view object and view link instances, displaying master-detail relationships.

5. Shuttle the selected master view object to the **Data Model** list

   For example, Figure 8-2 shows the newly created master view instance `AuthenticatedUser` in the **Data Model** list after you add it to the list.

Figure 8-2 Master View Instance Created

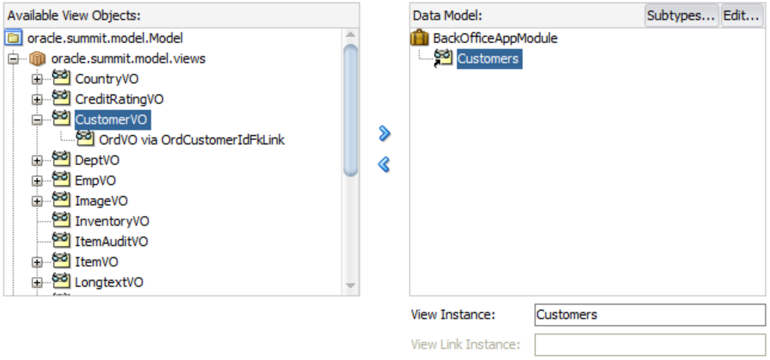


View Instance: CustomerVO1

6. In the **Data Model** list, leave the newly created master view instance selected, so that it appears highlighted. This will be the target of the detail view instance you will add. Then locate and select the detail view object beneath the master view object in the **Available View Objects** list.

   For example, Figure 8-3 shows the detail `OrdersVO` indented beneath master `PersonsVO` with the name `OrdersVO via PersonsToOrders`. The name identifies the view link `PersonsToOrders`, which defines the master-detail hierarchy between `PersonsVO` and `OrdersVO`. The detail view instance is renamed to `MyOrders`.

Figure 8-3 Detail View Object Selected



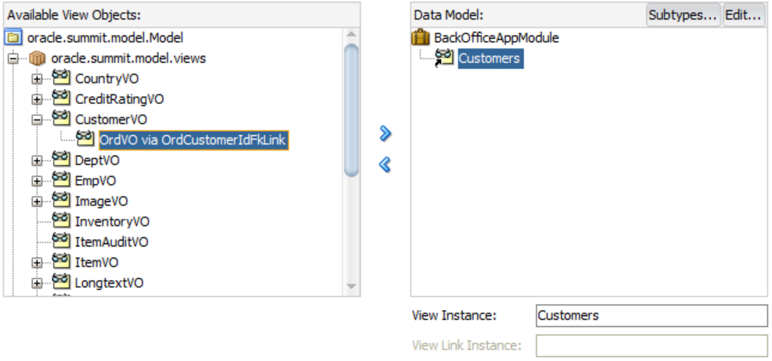
View Instance: Customers

View Link Instance:

7. To add the detail instance to the previously added master instance, shuttle the detail view object to the **Data Model** list below the selected master view instance.

   Figure 8-4 shows the newly created detail view instance `MyOrders` is a detail of the `AuthenticatedUser` in the data model.

Figure 8-4 Master View Instance Created



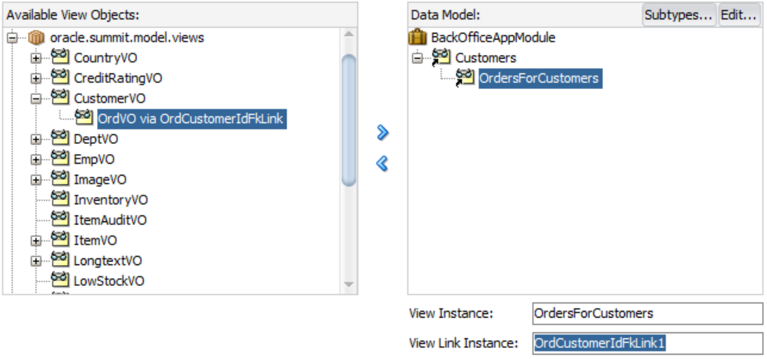
View Instance: Customers

View Link Instance:

8. To add another level of hierarchy, select the newly added detail in the **Data Model** list, then shuttle over the new detail which itself has a master-detail relationship with the previously added detail instance.

   Your data model can contain as many levels of hierarchy as your view object relationships support. For example, Figure 8-5 shows the **Data Model** list with instance `AuthenticatedUser` (renamed for `PersonsVO`) as the master of `MyOrders` (renamed for `OrdersVO via PersonsToOrders`), which in turn is a master for `MyOrderItems` (renamed from `OrderItemsVO via OrdersToOrderItems`). The detail view object `MyOrderItems` is the last level of the hierarchy possible because this view object is itself not a master for another view object.

Figure 8-5   Master-Detail-Detail Hierarchy Created



View Instance: OrdersForCustomers

View Link Instance: OrdCustomerIdFkLink1

Testing View Object Instances Using the Oracle ADF Model Tester

Use Oracle ADF Model Tester to test and diagnose the ADF application module using simulated end user interaction before building a custom user interface.

Using the Oracle ADF Model Tester, you can simulate an end user interacting with your **application module data model** before you have started to build any custom user interface of your own. Even *after* you have your UI pages constructed, you will come to appreciate using the Oracle ADF Model Tester to assist in diagnosing problems when they arise. You can reproduce the issues in the Oracle ADF Model Tester to discover whether the problem lies in the view or controller layers of the application, or whether there is instead a problem in the business service layer application module itself.

How to Run the Oracle ADF Model Tester

To test the view objects you added to an application module, use the Oracle ADF Model Tester, which is accessible from the Applications window. When you run the tester from the Applications window, the default application module configuration is used to connect to the database.

Before you begin:

It may be helpful to have an understanding of the Oracle ADF Model Tester. For more information, see Testing View Object Instances Using the Oracle ADF Model Tester.

You may also find it helpful to understand how to use debugging tools when you run the tester. For more information, see Using the Oracle ADF Model Tester for Testing and Debugging.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Testing View Instances.

To test view objects using an application module configuration:

1. In the Applications window, right-click the application module that you want to run and choose **Run**.

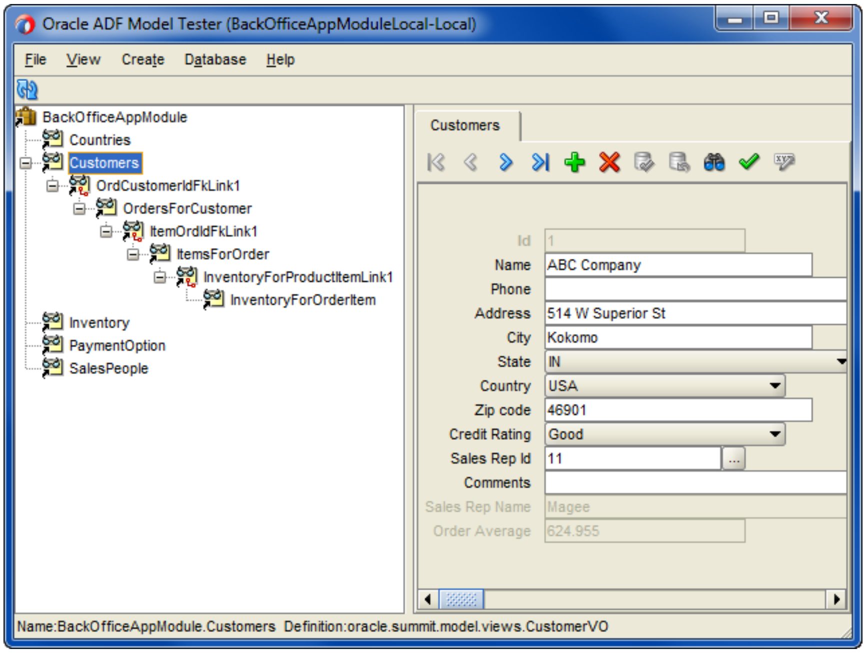
Alternatively, choose **Debug** when you want to run the application in the Oracle ADF Model Tester with debugging enabled. JDeveloper opens the debugger process panel in the Log window and the various debugger windows. For example, when debugging using the Oracle ADF Model Tester, you can view status message and exceptions, step in and out of source code, and manage breakpoints.

For information about receiving diagnostic messages specific to ADF Business Component debugging, see How to Enable ADF Business Components Debug Diagnostics.

2. In the Oracle ADF Model Tester, to execute a view object, expand the data model tree and double-click the desired view object node.

   Note that the view object instance may already appear executed in the testing session. In this case, the Oracle ADF Model Tester data view page on the right already displays query results for the view object instance. The fields in the Oracle ADF Model Tester data view page of a read-only view object will always appear disabled since the data it represents is not editable. For example, in Figure 8-6, data for the view instance `Products` appears in the tester. Fields like **Product Id**, **Language**, and **Category** appear disabled because the attributes themselves are not editable.

Figure 8-6 Testing the Data Model in the Oracle ADF Model Tester



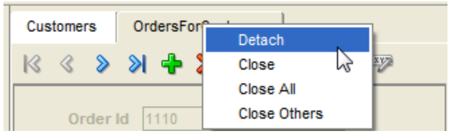
Id	1
Name	ABC Company
Phone	
Address	514 W Superior St
City	Kokomo
State	IN
Country	USA
Zip code	46901
Credit Rating	Good
Sales Rep Id	11
Comments	
Sales Rep Name	Magee
Order Average	624.955

Name:BackOfficeAppModule.Customers Definition:oracle.summit.model.views.CustomerVO

3. Right-click a node in the data model tree to display the context menu for that node. For example, on a view object node you can reexecute the query if needed, to remove the view object from the data model tree, and perform other tasks.

4. Right-click the tab of an open data viewer to display the context menu for that tab, as shown in Figure 8-7. For example, you can close the data viewer or open it in a separate window.

Figure 8-7 Context Menu for Data Viewer Tabs in the Oracle ADF Model Tester



How to Run the Oracle ADF Model Tester Using Configurations

To test the view objects you added to an application module, use the Oracle ADF Model Tester, and run it using a specific configuration. The configuration you select will determine the database connection used by the tester.

Before you begin:

It may be helpful to have an understanding of the Oracle ADF Model Tester. For more information, see Testing View Object Instances Using the Oracle ADF Model Tester.

You may also find it helpful to understand how to use debugging tools when you run the tester. For more information, see Using the Oracle ADF Model Tester for Testing and Debugging.

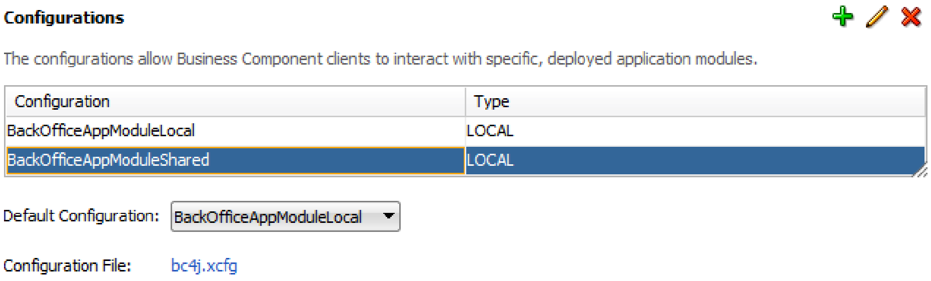
You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Testing View Instances.

To test view objects using a specific application module configuration:

1. In the Applications window, double-click the application module that you want to run.
2. In the overview editor, click the **Configurations** navigation tab.
3. In the Configurations page, select the desired application module configuration from the **Default Configuration** dropdown list that you want to use to run the Oracle ADF Model Tester.

   By default, an application module has only its default configurations, named `*AppModuleName*Local` and `*AppModuleName*Shared`. For example, Figure 8-8 shows the `BackOfficeAppModuleShared` configuration is selected to connect to the database.

Figure 8-8 Configuration Selection in Application Module Overview Editor



Configurations

The configurations allow Business Component clients to interact with specific, deployed application modules.

Configuration	Type
BackOfficeAppModuleLocal	LOCAL
BackOfficeAppModuleShared	LOCAL

Default Configuration: BackOfficeAppModuleLocal

Configuration File: bc4j.xcfg

4. Right-click the selected configuration and choose **Run**.

Alternatively, choose **Debug** when you want to run the application in the Oracle ADF Model Tester with debugging enabled. JDeveloper opens the debugger process panel in the Log window and the various debugger windows. For example, when debugging using the Oracle ADF Model Tester, you can view status message and exceptions, step in and out of source code, and manage breakpoints.

For information about receiving diagnostic messages specific to ADF Business Component debugging, see How to Enable ADF Business Components Debug Diagnostics.

How to Test Language Message Bundles and UI Hints

When your application defines alternative languages in your resource message bundles, you can configure the Oracle ADF Model Tester to recognize these languages. In the Oracle ADF Model Tester, you can then display the **Locale** menu and select among the available language choices.

To specify a default language for the Oracle ADF Model Tester:

1. In the main menu, choose **Tools** and then **Preferences**.
2. In the Preferences dialog, expand **ADF Business Components** and select **Tester**.
3. In the Oracle ADF Model Tester page, add any locale for which you have created a resource message bundle to the **Selected** list.

Alternatively, you can configure the default language choice by setting ADF Business Components runtime configuration properties for a specific application module configuration. These runtime properties also determine which language the Oracle ADF Model Tester will display as the default. To set runtime configuration properties, double-click the application module in the Applications window and, in the overview editor, select the **Configurations** navigation tab. Then, in the Configurations page of the overview editor, click the configuration hyperlink. In the application module configuration overview editor (on the `bc4j.xcfg` file), select the **Properties** tab and click **Add Property** to select the following properties from the Add Property dialog and click **OK**.

- `jbo.default.country`
- `jbo.default.language`

Then, in the **Properties** list enter the desired country code for the country and language. For example, to specify the Italian language, you would enter `IT` and `it` for these two properties:

- `jbo.default.country` = `IT`
- `jbo.default.language` = `it`

Testing the language message bundles in the Oracle ADF Model Tester lets you verify that the translations of the UI hints are correctly located. Or, if the message bundle defines date formats for specific attributes, the tool lets you verify that date formats change (like `04/12/2013` to `12/04/2013`).

How to Test Entity-Based View Objects Interactively

You test entity-based view objects interactively in the same way as read-only ones. Just add instances of the desired view objects to the data model of some application module, and then test that application module using the Oracle ADF Model Tester.

You'll find the Oracle ADF Model Tester invaluable for quickly testing and debugging your application modules. Table 8-1 gives an overview of the operations that the Oracle ADF Model Tester toolbar buttons perform when you display an entity-based view object.

Table 8-1 Oracle ADF Model Tester Toolbar Buttons

Button	Operation	Usage
⏮ ◀ ▶ ⏭	**Move to ... row**	Changes the current row displayed by the Oracle ADF Model Tester. Moves to the first, previous, next, or last row.
➕	**Insert a new row**	Creates and inserts a new row.
✕	**Delete the current row**	Deletes the current row.
✓	**Save changes to the database**	Posts and commits changes that you made in the ADF Business Components cache.
↺	**Discard all changes since last save**	Discards changes that you made in the ADF Business Components cache and restores the original values, rolling back any changes posted to the database.
🔍	**Specify view criteria**	Displays the View Criteria dialog that you can use to create and apply **view criteria** to the master view object instance.

Table 8-1 (Cont.) Oracle ADF Model Tester Toolbar Buttons

Button	Operation	Usage
✓	**Validate row**	Validates the current row by applying validation rules defined for all **entity object** instances. Disabled unless at least one field is editable.
✎	**Edit bind variables**	Displays the Bind Variable dialog that you can use to enter values for bind parameters used in the view object query. Disabled unless the view object query uses bind parameters in the query statement.

To test the entity-based view objects you added to an application module, use the Oracle ADF Model Tester, which is accessible from the Applications window.

Before you begin:

It may be helpful to have an understanding of the Oracle ADF Model Tester. See Testing View Object Instances Using the Oracle ADF Model Tester.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. See Additional Functionality for Testing View Instances.

To test entity-based view objects using an application module configuration:

1. In the Applications window, right-click the application module that you want to test and choose **Run**.
2. In the Oracle ADF Model Tester, expand the data model tree and double-click the desired entity-based view object node.

   Unlike the fields of a read-only view object, the fields displayed in the data view page will appear enabled, because the data it represents is editable.
3. To update individual values and perform validation checks on the entered data, use the editable fields.

   In the case of a view instance with referenced entities, you can change the foreign key value and observe that the referenced part changes.
4. To perform row-level operations, such as navigate rows, create row, remove row, and validate the current row, use the toolbar buttons.

   For further discussion about simulating end-user interaction in the data view page, see How to Simulate End-User Interaction in the Oracle ADF Model Tester.

How to Update the Oracle ADF Model Tester to Display Project Changes

Normally, changes that you make to the data model project will not be picked up automatically by running the Oracle ADF Model Tester. You can, however, force the Oracle ADF Model Tester to reload metadata from the data model project any time you want to synchronize the displayed data model and the data model project. This option is an alternative to quitting the Oracle ADF Model Tester, editing your project, and rerunning the Oracle ADF Model Tester to view the latest changes.

Using the **Reload Application** option saves time, especially as you work iteratively between the Oracle ADF Model Tester and JDeveloper. For example, while running the Oracle ADF Model Tester you might determine the need to modify the data model with a new view instance or you might find that a view instance is missing an LOV attribute definition. You can return to JDeveloper and use the Business Components overview editors to make the changes that alter the data model metadata. Then, after you recompile the project (a necessary step), you can return to the Oracle ADF Model Tester to reload the updated metadata from the project's class path.

Before you begin:

It may be helpful to have an understanding of the Oracle ADF Model Tester. For more information, see Testing View Object Instances Using the Oracle ADF Model Tester.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Testing View Instances.

To reload the data model metadata in the running Oracle ADF Model Tester:

1. In the Applications window, right-click the application module and choose **Run**.
2. In the Oracle ADF Model Tester, test the data model and determine any changes you want to make. Do not exit the Oracle ADF Model Tester.
3. In JDeveloper, make the desired changes and recompile the data model project. (For example, you can right-click the data model project in the Applications window and choose **Make** to complete the recompile step.)

   Although the metadata changes that you make are not involved in compiling the project, the compile step is necessary to copy the metadata to the class path and to allow the Oracle ADF Model Tester to reload it.
4. Return to the Oracle ADF Model Tester and click the **Reload the Application Metadata** button above the data model tree. The Oracle ADF Model Tester closes all open windows.

   Alternatively, you can choose **Reload Application** from the **File** menu of the Oracle ADF Model Tester.
5. Reopen the desired windows and view your changes.

What Happens When You Use the Oracle ADF Model Tester

When you launch the Oracle ADF Model Tester, JDeveloper starts the tool in a separate process and the Oracle ADF Model Tester appears. The tree at the left of the dialog displays all of the view object instances in your application module's data model. After you double-click the desired view object instance, the Oracle ADF Model Tester will display a data view page to inspect the query results. For example, Figure 8-9 shows the view instance `Products` that has been double-clicked in the expanded tree to display the data for this view instance in the data view page on the right.

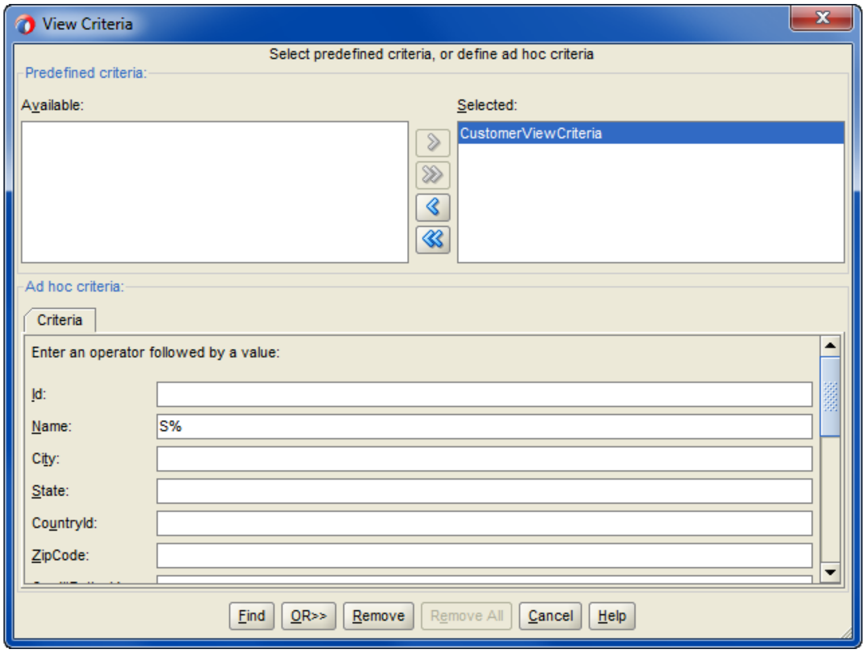
The data view page will appear disabled for any read-only view objects you display because the data is not editable. But even for a read-only view object, the tool affords some useful features:

- You can validate that the UI hints based on the Label Text hint and format masks are defined correctly.
- You can also scroll through the data using the toolbar buttons.
- You can enter Query-by-Example criteria to find a particular row whose data you want to inspect. By clicking the **Specify View Criteria** button in the toolbar, the View Criteria dialog displays the list of available Query-by-Example criteria.

  For example, as shown in Figure 8-9, you can select a view criteria like `FindByProductNameCriteria` and enter a query criteria like "`P%`" for a `ProductName` attribute and click **Find** to narrow the search to only those products with a name that begins with the letter `P`.

The Oracle ADF Model Tester becomes even more useful when you create entity-based view objects that allow you to simulate inserting, updating, and deleting rows, as described in How to Test Entity-Based View Objects Interactively.

Figure 8-9 Built-in Query-by-Example Functionality



Id:

Name: S%

City:

State:

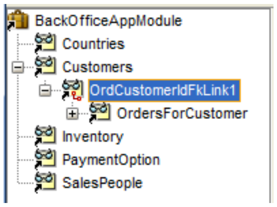
CountryId:

ZipCode:

How to Simulate End-User Interaction in the Oracle ADF Model Tester

When you launch the Oracle ADF Model Tester, the tree at the left of the display shows the hierarchy of the view object instances that the data model of your application module defines. If the data model defines master-detail view instance relationships, the tree will display them as parent and child nodes. A node between the master-detail view instances represent the view link instance that performs the active master-detail coordination as the current row changes in the master. For example, in Figure 8-10 the tree is expanded to show the master-detail relationship between the master `Products` view instance and the detail `WarehouseStockLevels` view instance. The selected node, `ProductsToWarehouseStockLevels1`, is the view link instance that defines the master-detail relationship.

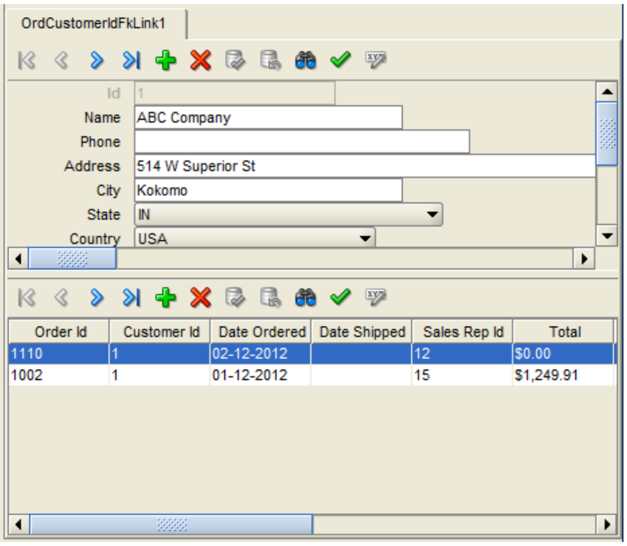
Figure 8-10 Application Module Data Model in the Oracle ADF Model Tester



Double-clicking the view link instance executes the master object and displays the master-detail data in the data view page. For example, in Figure 8-11, double-clicking the `ProductsToWarehouseStockLevels1` view link instance in the tree executes the `Products` master view instance in the top portion of the data view page and the `WarehouseStockLevels` view instance in the bottom portion of the data view page. Additional context menu items on the view object node allow you to reexecute the query if needed, remove the view object from the data model panel, and perform other tasks.

In the master-detail data view page, you can scroll through the query results. Additionally, because instance of entity-based view objects are fully editable, Instead of displaying *disabled* UI controls showing read-only data for a read-only view object, the data view page displays editable fields. You are free to experiment with creating, inserting, updating, validating, committing, and rolling back.

Figure 8-11 Master-Detail Data View Page in the Oracle ADF Model Tester



Id 1
Name ABC Company
Phone
Address 514 W Superior St
City Kokomo
State IN
Country USA

Order Id	Customer Id	Date Ordered	Date Shipped	Sales Rep Id	Total
1110	1	02-12-2012		12	$0.00
1002	1	01-12-2012		15	$1,249.91

For example, you can view multiple levels of master-detail hierarchies, opening multiple data view pages at the same time. Use the **Detach** context menu item to open any tab into a separate window and visualize multiple view object's data at the same time.

Using just the master-detail data view page, you can test several functional areas of your application.

Testing Master-Detail Coordination

When you click the navigation buttons on the Oracle ADF Model Tester toolbar, you can see that the rows for the current master view object are correctly coordinated. For example, Figure 8-11 shows a master-detail hierarchy with products and warehouses. If you click the **Next Row** button in the master panel, the master panel will display the next product (identified by a product ID) and the detail panel will update to display the list of warehouses and quantities available for the product.

Testing UI Hints

The entity-based view object attributes inherit their UI hints from those on the underlying entity object attribute. The prompts displayed in the data view page help you see whether you have correctly defined a user-friendly label text UI hint for each attribute. For details on setting up the hint on your entity object, see Defining UI Hints for View Objects.

Testing Business Domain Layer Validation

Depending on the validation rules you have defined, you can try entering invalid values to trigger and verify validation exceptions. For example, when you have defined a range validation rule, enter a value outside the range and see an error similar to:

```
(oracle.jbo.AttrSetValException) Valid product codes are between 100 and 999
```

Click the rollback button in the toolbar to revert data to the previous state.

Testing View Objects That Reference Entity Usages

By scrolling through the data — or using the **Specify View Criteria** button in the Oracle ADF Model Tester toolbar to search — you can verify whether you have correctly altered the `WHERE` clause in an entity-based view object's query to use an outer join. The rows should appear as expected.

You also can try changing a primary key attribute of a master view object. This will allow you to verify that the corresponding reference information is automatically updated to reflect the new primary key value.

Use the Oracle ADF Model Tester to verify that UI hints defined at the view object level override the ones it would normally inherit from the underlying entity object. If you notice that several attributes share the same label text, you can edit the UI hint for the desired attributes at the view object level. For example, you can set the **Label Text** hint to **Member Since** for the `RegisteredDate` attribute and **Provisioned?** for the `ProvisionedFlag` attribute.

Testing Row Creation and Default Value Generation

When displaying an entity-based view object, click the **Create Row** button in the Oracle ADF Model Tester toolbar for the view object instance to create a new blank row. Any fields that have a declarative default value will appear with that value in the blank row. If the `DBSequence`-valued attribute is used, a temporary value will appear in the new row. After entering all the required fields, click the **Commit** button to commit the transaction. The actual, trigger-assigned primary key should appear in the field after successful commit.

Testing That New Detail Rows Have Correct Foreign Keys

If you click **Create Row** in the Oracle ADF Model Tester toolbar to try adding a new row to an existing detail entity-based view object instance, you'll notice that the view link automatically ensures that the foreign key attribute value in the new row is set to the value of the current master view instance row.

How to Test Multiuser Scenarios in the Oracle ADF Model Tester

When view objects and entity objects cooperate at runtime, two exceptions can occur when you run the application in a multiuser environment. To anticipate these exceptions, you can simulate a multiuser environment for testing purposes using the Oracle ADF Model Tester. For example, when the application displays edit forms for view object queries, what is the expected behavior when two users attempt to modify the same attribute in their forms?

To understand the expected behavior, open two instances of the Oracle ADF Model Tester on the application module to simulate two users editing the same view object attribute. Keep both instances open and perform the following two tests to demonstrate how multiuser exceptions can arise:

- In one instance of the Oracle ADF Model Tester, modify an attribute of an existing view object and tab out of the field. Then, in the other tester instance, try to modify the same view object attribute in some way. You'll see that the second user gets the `oracle.jbo.AlreadyLockedException`.

  You can then change the value of `jbo.locking.mode` to be `pessimistic` on the Properties page of the Oracle ADF Model Tester Connect dialog and try repeating the test (the default mode is set to `optimistic`). You'll see the error occurs for the second user immediately after changing the value instead of after committing the change.
- In one instance of the Oracle ADF Model Tester, modify an attribute of an existing view object and tab out of the field. Then, in the other tester instance, retrieve (but don't modify) the same view object attribute. Back in the first window, commit the change. If the second user then tries to modify that same attribute, you'll see that the second user gets the `oracle.jbo.RowInconsistentException`. The row has been modified and committed by another user since the second user retrieved the row into the entity cache.

How to Customize Configuration Options Before Running the Tester

Using the overview editor for the application module, you can select a predefined configuration to run the tool using that named set of runtime configuration properties. The Configurations page of the overview editor also lets you open the overview editor for application module configurations (defined in the `bc4j.xcfg` file) to specify properties of the configuration before running the tester.

To display the application module configuration overview editor, double-click the application module in the Applications window and, in the overview editor, select the **Configurations** navigation tab. Then, in the Configurations page of the overview editor, click the configuration hyperlink. In the application module configuration overview editor, select the **Properties** tab and click **Add Property** to select the desired property from the Add Property dialog and click **OK**.

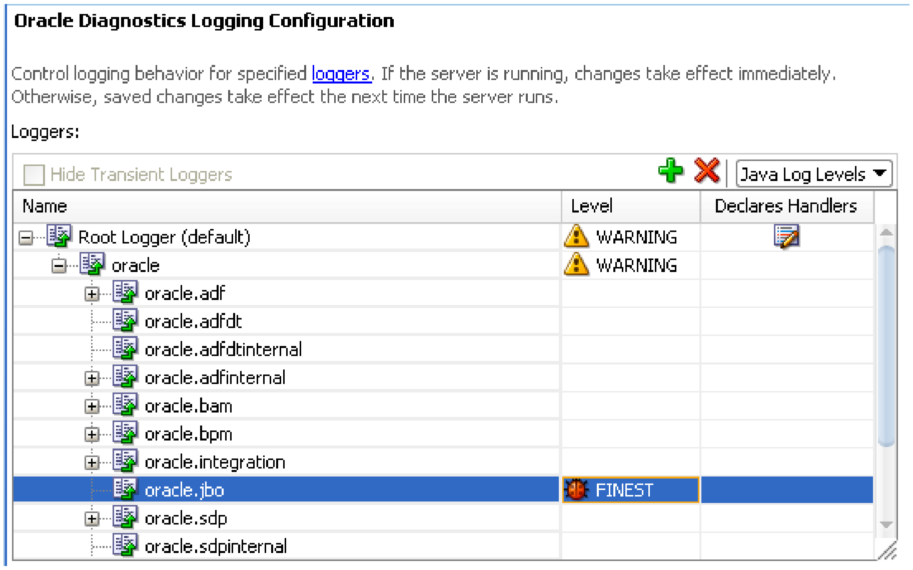
For example, you could alter the default language for the UI hints by editing the configuration and choosing the **Properties** tab to add and then set the following two properties with the desired country code (in this case, IT for Italy):

- jbo.default.country = IT
- jbo.default.language = it

How to Enable ADF Business Components Debug Diagnostics

When launching the Oracle ADF Model Tester, if you have configured diagnostic logging for the oracle.jbo logger, JDeveloper will direct ADF Business Components debug diagnostics messages to the JDeveloper Log window. Figure 8-12 shows the oracle.jbo logger set to a log level of FINEST to enable ADF Business Components debug diagnostics.

Figure 8-12 Configuring a Logger for ADF Business Components Debugging



Oracle Diagnostics Logging Configuration

Control logging behavior for specified loggers. If the server is running, changes take effect immediately. Otherwise, saved changes take effect the next time the server runs.

Loggers:

□ Hide Transient Loggers

Name	Level	Declares Handlers
Root Logger (default)	WARNING	
oracle	WARNING	
oracle.adf		
oracle.adfdt		
oracle.adfdtinternal		
oracle.adfinternal		
oracle.bam		
oracle.bpm		
oracle.integration		
oracle.jbo	FINEST	
oracle.sdp		
oracle.sdpinternal		

The oracle.jbo logger can be configured either before running the application or while the application is running in Integrated WebLogic Server. The logging will begin without the need to restart the server. You do not need to run the application in debug mode to log diagnostic messages.

With the oracle.jbo logger configured, the next time you run the Oracle ADF Model Tester and double-click the view object, you'll see detailed diagnostic output in the Log window, as shown in the following example. Configuring the oracle.jbo logger with a log level FINEST will allow you to visualize everything the ADF Business Components framework components are doing for your application.

```
   :
[355] Oracle SQLBuilder: Registered driver: oracle.jdbc.OracleDriver
[356] Creating a new pool resource
[357] **** DBTransactionImpl establishNewConnection
[358] Successfully logged in
[359] JDBCDriverVersion: xx.xx.x.x-Production
[360] DatabaseProductName: Oracle
```

```
[361] DBTransactionImpl initTransaction
[362] Replacing: null with: StoreServiceAM_AddressesPageDef
[363] Replacing: null with: StoreServiceAM_MostPopularProductsByCategoriesPageDef
...
[537] Orders ViewRowSetImpl.execute caused params to be "un"changed
[538] Column count: 41
[539] ViewObject: Orders Created new QUERY statement
[540] Orders>#q computed SQLStmtBufLen: 952, actual=865, storing=895
[541] SELECT OrderEO.ORDER_ID, OrderEO.ORDER_DATE, OrderEO.ORDER_SHIPPED_DATE,
       FROM ORDERS OrderEO ORDER BY OrderEO.ORDER_DATE desc
[542] Bind params for ViewObject: Orders
```

For backward compatibility, it remains possible to enable ADF Business Components debug diagnostics on the data model project using the Java system property `jbo.debugoutput=console`. To set the property, open the Run/Debug/Profile page in the Project Properties dialog for your data model project. Click **Edit** to edit the chosen run configuration, and add `-Djbo.debugoutput=console` to the **Java Options** field in the page. Other legal values for this property are `silent` (the default, if not specified) and `file`. If you choose the file option, diagnostics are written to the system `temp` directory.

Other legal values for the `-Djbo.debugoutput` system property are `silent` (the default, if not specified) and `file`. If you enter the `file` option, diagnostics are written to the system `temp` directory.

What Happens at Runtime: How View Objects and Entity Objects Cooperate

On their own, view objects and entity objects simplify two important jobs that every enterprise application developer needs to do:

- Work with SQL query results
- Modify and validate rows in database tables

Entity-based view objects can query any selection of data that you want the end user to be able to view and modify. Any data the end user is allowed to change will be validated and saved by your reusable business domain layer. The key ingredients you provide as the developer are the ones that only *you* can know:

- You decide what business logic should be enforced in your business domain layer
- You decide what queries describe the data you need to put on the screen

These are the things that make *your* application unique. The built-in functionality of your entity-based view objects handles the rest of the implementation details.

> **Note:**
>
> Understanding row keys and what role the entity cache plays in the transaction are important concepts that help to clarify the nature of the entity-based view objects. These two concepts are addressed in ViewObject Interface Methods for Working with the View Object's Default RowSet.

What Happens at Runtime: After a View Object Executes Its Query

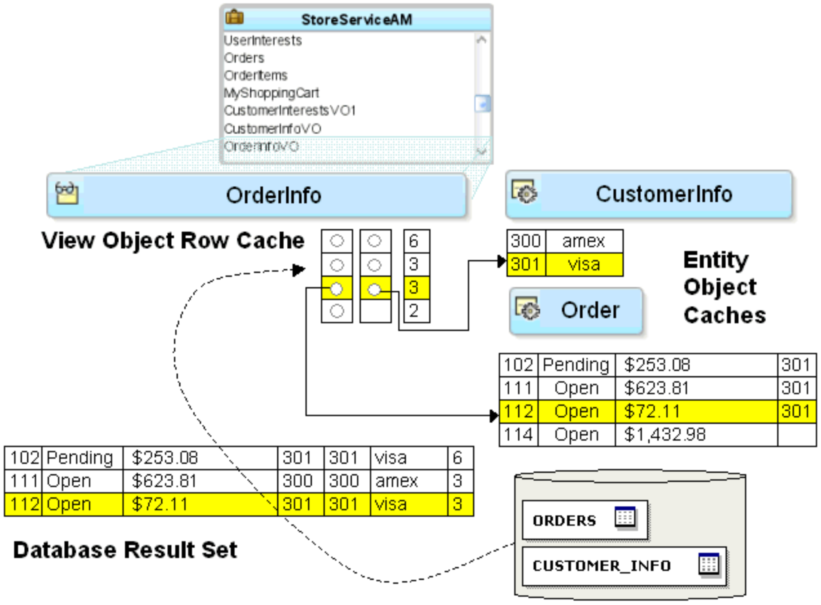
After adding an instance of an entity-based view object to the application module's data model, you can see what happens at runtime when you execute the query. Like a read-only view object, an entity-based view object sends its SQL query straight to the database using the standard Java Database Connectivity (JDBC) API, and the database produces a result set. In contrast to its read-only counterpart, however, as the entity-based view object retrieves each row of the database result set, it partitions the row attributes based on which entity usage they relate to. This partitioning occurs by creating an entity object row of the appropriate type for each of the view object's entity usages, populating them with the relevant attributes retrieved by the query, and storing each of these entity rows in its respective entity cache. Then, rather than storing duplicate copies of the data, the view row simply *points* at the entity row parts that comprise it.

Figure 8-13 illustrates how the entity cache partitions the result set attributes of two entity-based view objects. In this example, the highlighted row in the database result set is partitioned into an `Order` entity row with primary key `112` and a `CustomerInfo` entity row with primary key `301`.

As described in The Role of the Entity Cache in the Transaction, the entity row that is brought into the cache using `findByPrimaryKey()` contains *all* attributes of the entity object. In contrast, an entity row created by partitioning rows from the entity-based view object's query result contains values only for attributes that appear in the query. It does *not* include the complete set of attributes. This partially populated entity row represents an important runtime performance optimization.

Since the ratio of rows retrieved to rows modified in a typical enterprise application is very high, you can save memory by bringing only the attributes into memory that you need to display instead of bringing all attributes into memory all the time.

Figure 8-13 Entity Cache Partitions View Rows into Entity Rows



○	○	6
○	○	3
○	○	3
○		2

300	amex
301	visa

102	Pending	$253.08	301
111	Open	$623.81	301
112	Open	$72.11	301
114	Open	$1,432.98	

102	Pending	$253.08	301	301	visa	6
111	Open	$623.81	300	300	amex	3
112	Open	$72.11	301	301	visa	3

By partitioning queried data this way into its underlying entity row constituent parts, the first benefit you gain is that all of the rows that include some data queried will display a consistent result when changes are made in the current transaction. In other words, if one view object allows the PaymentType attribute of customer 301 to be modified, then all rows in any entity-based view object showing the PaymentType attribute for customer 301 will update instantly to reflect the change. Since the data related to customer 301 is stored exactly once in the CustomerInfo entity cache in the entity row with primary key 301, any view row that has queried the order's PaymentType attribute is just pointing at this single entity row.

Luckily, these implementation details are completely hidden from a client working with the rows in a view object's row set. The client works with a view row, getting and setting the attributes, and is unaware of how those attributes might be related to entity rows behind the scenes.

What Happens at Runtime: After a View Row Attribute Is Modified

When a user attempts to update the attribute of a view row, a series of steps occur to automatically coordinate this view row attribute modification with the underlying entity row. These steps ensure that a validation rule defined on the entity-mapped attribute will be triggered before the value is changed.

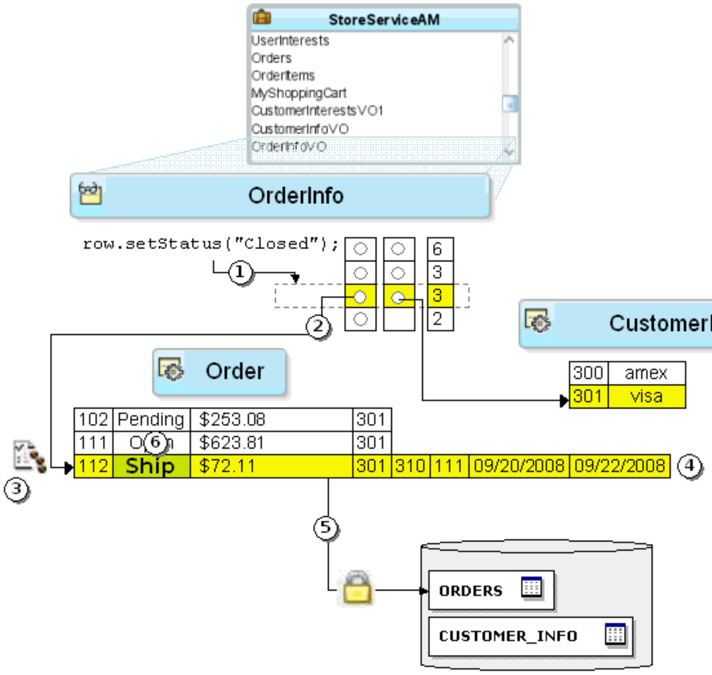
Figure 8-14 illustrates the basic steps that occur at runtime when the user attempts to update an entity-mapped attribute. In this example, the modified attribute Status is mapped to an entity usage where a validation rule is defined.

1. The user attempts to set the Status attribute to the value Ship.
2. Since Status is an entity-mapped attribute from the Order entity usage, the view row delegates the attribute set to the appropriate underlying entity row in the Order entity cache having primary key 112.
3. Any attribute-level validation rules on the Status attribute of the Order entity object are evaluated and the modification attempt will fail if any rule does not succeed.

   Assume that some validation rule for the Status attribute programmatically references the ShipDate attribute (for example, to enforce a business rule that an Order cannot be shipped the same day it is placed). The ShipDate was not one of the Order attributes retrieved by the query, so it is not present in the partially populated entity row in the Order entity cache.
4. To ensure that business rules can always reference all attributes of the entity object, the entity object detects this situation and "faults-in" the entire set of Order entity object attributes for the entity row being modified using the primary key (which must be present for each entity usage that participates in the view object).
5. After the attribute-level validations all succeed, the entity object attempts to acquire a lock on the row in the ORDERS table before allowing the first attribute to be modified.
6. If the row can be locked, the attempt to set the Status attribute in the row succeeds and the value is changed in the entity row.

> **Note:**
>
> The `jbo.locking.mode` configuration property controls how rows are locked. The default value is `optimistic`. Typically, Fusion web applications will use the default setting `optimistic`, so that rows aren't locked until transaction commit time. In `pessimistic` locking mode, the row must be lockable before any change is allowed to it in the entity cache.

Figure 8-14 View Row Attribute Updates Delegate to the Entity



What Happens at Runtime: After a Foreign Key Attribute is Changed

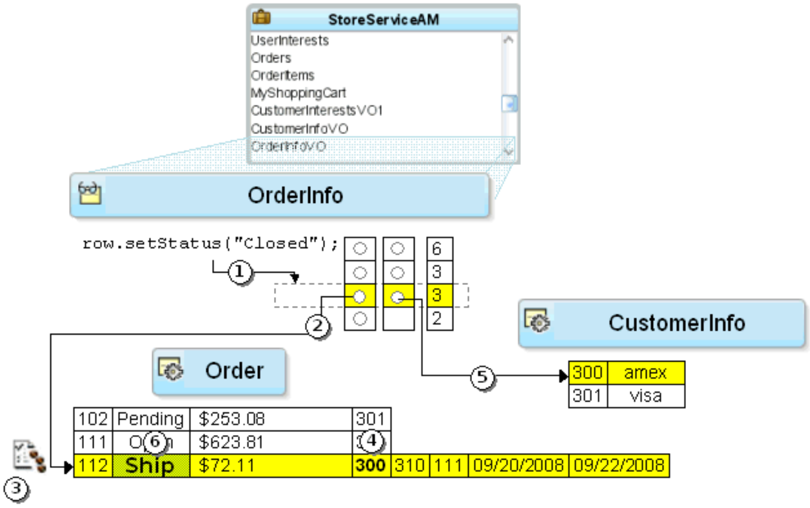
When a user attempts to update a foreign key attribute, a series of steps occur to automatically coordinate this view row attribute modification with the underlying entity row. These steps ensure that a validation rule defined on the foreign key, entity-mapped attribute will be triggered before the value is changed. They also ensure that the view row for the changed foreign key attribute reflects the correct attributes of all referenced entity objects.

Figure 8-15 illustrates the basic steps that occur at runtime when the user attempts to update a foreign key, entity-mapped attribute. In this example, the modified attribute `CustomerInfoId` is mapped to an entity usage `Order` where the attribute is associated with another entity object `CustomerInfo`.

1. The user attempts to set the `CustomerInfoId` attribute to the value `300`.
2. Since `CustomerInfoId` is an entity-mapped attribute from the `Order` entity usage, the view row delegates the attribute set to the appropriate underlying entity row in the `Order` entity cache, which has primary key `112`.

3. Any attribute-level validation rules on the `CustomerInfoId` attribute of the `Order` entity object are evaluated and the modification attempt will fail if any rule does not succeed.
4. The row is already locked, so the attempt to set the `CustomerInfoId` attribute in the row succeeds and the value is changed in the entity row.
5. Since the `CustomerInfoId` attribute on the `Order` entity usage is associated with the `CustomerInfo` entity object, this change of foreign key value causes the view row to replace its current entity row part for customer `301` with the entity row corresponding to the new `CustomerInfoId = 300`. This effectively makes the view row for order `112` point to the entity row for `300`, so the value of the `PaymentType` in the view row updates to reflect the correct reference information for this newly assigned customer.

Figure 8-15 After Updating a Foreign Key, View Row Points to a New Entity



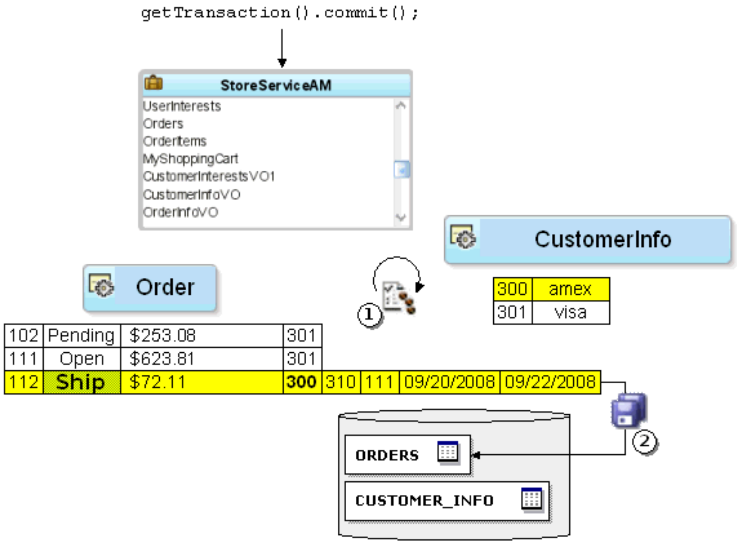
What Happens at Runtime: After a Transaction is Committed

Suppose the user is satisfied with the changes, and commits the transaction. As shown in Figure 8-16, there are two basic steps:

1. The `Transaction` object validates any invalid entity rows in its pending changes list.
2. The entity rows in the pending changes list are saved to the database.

The figure depicts a loop in Step 1 before the act of validating one modified entity object might programmatically affect changes to other entity objects. Once the transaction has processed its list of invalid entities on the pending changes list, if the list has entities, the transaction will complete another pass. It will attempt up to ten passes through the list. If by that point there are still invalid entity rows, it will throw an exception because this typically means you have an error in your business logic that needs to be investigated.

Figure 8-16 Committing the Transaction Validates Invalid Entities, Then Saves Them



What Happens at Runtime: After a View Object Requeries Data

When you reexecute a view object's query, by default the view rows in its current row set are "forgotten" in preparation for reading in a fresh result set. This view object operation does not directly affect the entity cache, however. The view object then sends the SQL to the database and the process begins again to retrieve the database result set rows and partition them into entity row parts.

> **Note:**
>
> Typically when the view object requeries data, you expect it to retrieve the latest database information. If instead you want to avoid a database roundtrip by restricting your view object to querying only over existing entity rows in the cache, or over existing rows already in the view object's row set, see Performing In-Memory Sorting and Filtering of Row Sets.

How Unmodified Attributes are Handled During Requery

As part of the entity row partitioning process during a requery, if an attribute on the entity row is unmodified, then its value in the entity cache is updated to reflect the newly queried value.

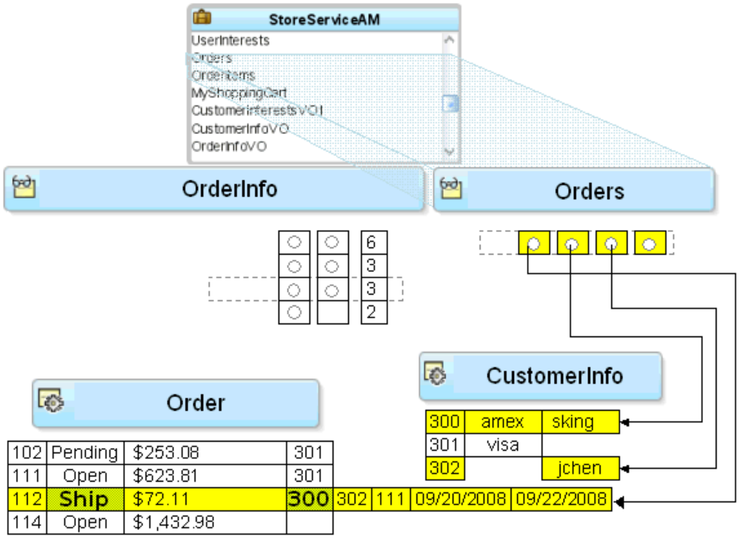
How Modified Attributes are Handled During Requery

However, if the value of an entity row attribute has been *modified* in the current transaction, then during a requery the entity row partitioning process does not refresh its value. Uncommitted changes in the current transaction are left intact so the end-user's logical unit of work is preserved. As with any entity attribute value, these pending modifications continue to be consistently displayed in any entity-based view object rows that reference the modified entity rows.

> **Note:**
>
> End-user row inserts and deletes are also managed by the entity cache, which permits new rows to appear and deleted rows to be skipped during requerying. For more information about new row behavior, see Maintaining New Row Consistency Between View Objects Based on the Same Entity.

For example, Figure 8-17 illustrates the scenario where a user "drills down" to a different page that uses the `Orders` view object instance to retrieve all details about order `112` and that this happens in the context of the current transaction's pending changes. That view object has two entity usages: a primary `Orders` usage and a reference usage for `CustomerInfo`. When its query result is partitioned into entity rows, it ends up pointing at the same `Order` entity row that the previous `OrderInfo` view row had modified. This means the end user will correctly see the pending change, that the order is assigned to `sking` in this transaction.

Figure 8-17 Entity Cache Merges Sets of Entity Attributes from Different View Objects



How Overlapping Subsets of Attributes are Handled During Requery

Two different view objects can retrieve two *different* subsets of reference information and the results are merged whether or not they have matching sets of attributes. For example, Figure 8-17 also illustrates the situation, where the `Orders` view object queries the user's `Email`, while the `OrderInfo` view object queried the user's `PaymentOption`. The figure shows what happens at runtime: if while partitioning the retrieved row, the entity row part contains a different set of attributes than does the partially populated entity row that is already in the cache, the attributes get "merged". The result is a partially populated entity row in the cache with the *union* of the overlapping subsets of user attributes. In contrast, for `jchen` (user 302), who wasn't in the cache already, the resulting new entity row contains only the `Email` attribute, but not the `PaymentOption`.

What You May Need to Know About Optimizing View Object Runtime Performance

The view object provides tuning parameters that let you control how SQL is executed and how data is fetched from the database. These tuning parameters play a significant role in the runtime performance of the view object. If the fetch options are not tuned correctly for the application, then your view object may fetch an excessive amount of data and may make too many roundtrips to the database.

You can use the **Tuning** section of the General page of the overview editor to configure the fetch options shown in Table 8-2.

Table 8-2 Parameters to Tune View Object Performance

Fetch Tuning Parameters	Usage
Fetch Mode	The default fetch option is the **All Rows** option, which will be retrieved **As Needed** (`FetchMode="FETCH_AS_NEEDED"`) or **All at Once** (`FetchMode="FETCH_ALL"`), depending on which option is desired. The **As Needed** option ensures that an `executeQuery()` operation on the view object initially retrieves only as many rows as necessary to fill the first page of a display, whose number of rows is set based on the view object's range size.
Fetch Size	In conjunction with the **Fetch Mode** option, the **in Batches of** field controls the number of records fetched at one time from the database (`FetchSize` in the view object XML). The default value is 1, which will give poor performance unless only one row will be fetched. The suggested configuration is to set this value to *n*+3 where *n* is the number of rows to be displayed in the user interface. For more information about how this parameter may affect memory requirements, see Consider Whether Fetching One Row at a Time is Appropriate.

Table 8-2 (Cont.) Parameters to Tune View Object Performance

Fetch Tuning Parameters	Usage
Max Fetch Size	The default max fetch size for a view object is -1, which means that there is no limit to the number of rows the view object can fetch. In cases where the result set should contain only *n* rows of data, the option **Only up to row number** should be selected and set to *n*. The developer can alternatively call `setMaxFetchSize(n)` to set this programmatically or manually add the parameter `MaxFetchSize` to the view object XML. For view objects whose `WHERE` clause expects to retrieve a single row, set the option **At Most One Row**. This way the view object knows you don't expect any more rows and it will skip its normal test for that situation. As mentioned earlier, setting a maximum fetch size of 0 (zero) makes the view object insert-only. In this case, no select query will be issued, so no rows will be fetched. When you want to specify a global threshold for all view object queries in the application, you can configure the **Row Fetch Limit** property in the `adf-config.xml` file. Setting this property means you can avoid changing the **Max Fetch Size** for individual query operations. If you do specify a fetch limit for individual view objects, the **Row Fetch Limit** setting will be ignored in those cases. For more details about **Row Fetch Limit**, see Using Fetch Size to Limit the Maximum Number of Records Fetched for a View Object.
Forward-only Mode	If a data set will only be traversed going forward, then forward-only mode can help performance when iterating through the data set. This can be configured by programmatically calling `setForwardOnly(true)` on the view object. Setting forward-only will also prevent caching previous sets of rows as the data set is traversed.

When you tune view objects, you should also consider these issues:

- Large data sets: View objects provide a mechanism to page through large data sets such that a user can jump to a specific page in the results. This is configured by calling `setRangeSize(n)` followed by `setAccessMode(RowSet.RANGE_PAGING)` on the view object where *n* is the number of rows contained within one page. When the user navigates to a specific page in the data set, the application can call `scrollToRangePage(P)` on the view object to navigate to page *P*. Range paging fetches and caches only the current page of rows in the view object row cache at the cost of another query execution to retrieve each page of data. Range paging is not appropriate where it is beneficial to have all fetched rows in the view object row cache (for example, when the application needs to read all rows in a dataset for an LOV or page back and forth in records of a small data set.

- Spillover: There is a facility to use the data source as "virtual memory" when the JVM container runs out of memory. By default, this is disabled and can be turned on as a last resort by setting `jbo.use.pers.coll=true`. Enabling spillover can have a large performance impact.
- SQL platform: If the generic SQL92 SQL platform is used to connect to generic SQL92-compliant databases, then some view object tuning options will not function correctly. The parameter that choosing the generic SQL92 SQL platform affects the most is the fetch size. When SQL92 SQL platform is used, the fetch size defaults to 10 rows regardless of what is configured for the view object. You can set the SQL platform when you define the database connection or you can define it as global project setting in the `adf-config.xml` file. By default, the SQL platform will be `Oracle`. To manually override the SQL platform, you can also pass the parameter `-Djbo.SQLBuilder="SQL92"` to the JVM upon startup.

Additionally, you have some options to tune the view objects' associated SQL for better database performance:

- Bind variables: If the query associated with the view object contains values that may change from execution to execution, use bind variables. Using bind variables in the query allows the query to reexecute without needing to reparse the query on the database. You can add bind variables to the view object in the Query page of the overview editor for the view object. For more information, see Working with Bind Variables.
- Query optimizer hints: The view object can pass hints to the database to influence which execution plan to use for the associated query. The optimizer hints can be specified in the **Retrieve from the Database** group box in the Tuning section of the overview editor for the view object. For information about optimizer hints, see Specify a Query Optimizer Hint if Necessary.

Testing View Object Instances Programmatically

You can use client programs to test an ADF application module programmatically.

When you are ready to test a working application module containing at least one view object instance, you can build a simple test client program to illustrate the basics of working programmatically with the data in the contained view object instances.

From the point of view of a client accessing your application module's data model, the API's to work with a read-only view object and an entity-based view object are identical. The key functional difference is that entity-based view objects allow the data in a view object to be fully updatable. The application module that contains the entity-based view objects defines the unit of work and manages the transaction. This section presents test client programs that work with the `SummitADF_Examples` workspace to illustrate:

- Iterating master-detail-detail hierarchy
- Finding a row and updating a foreign key value
- Creating a new order
- Retrieving the row key identifying a row

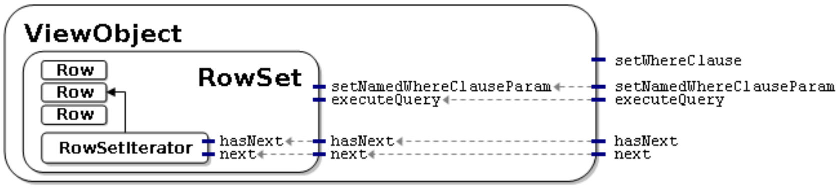
ViewObject Interface Methods for Working with the View Object's Default RowSet

The `ViewObject` interface in the `oracle.jbo` package provides the methods to easily perform any data-retrieval task. Some of these methods used in the example include:

- `executeQuery()`, to execute the view object's query and populate its row set of results
- `setWhereClause()`, to add a dynamic predicate at runtime to narrow a search
- `setNamedWhereClauseParam()`, to set the value of a named bind variable
- `hasNext()`, to test whether the **row set iterator** has reached the last row of results
- `next()`, to advance the row set iterator to the next row in the row set
- `getEstimatedRowCount()`, to count the number of rows a view object's query would return

Typically, when you work with a view object, you will work with only a single row set of results at a time. To simplify this overwhelmingly common use case, as shown in Figure 8-18, the view object contains a default `RowSet`, which, in turn, contains a default `RowSetIterator`. The default `RowSetIterator` allows you to call all of the data-retrieval methods directly on the `ViewObject` component itself, knowing that they will apply automatically to its default row set.

Figure 8-18 ViewObject Contains a Default RowSet and RowSetIterator



> **Note:**
>
> Defining Polymorphic View Objects presents situations when you might want a single view object to produce *multiple* distinct row sets of results. You can also find scenarios for creating *multiple* distinct row set iterators for a row set. Most of the time, however, you'll need only a single iterator.

The phrase "working with the rows in a view object," when used in this guide more precisely means working with the rows in the view object's default row set. Similarly, the phrase "iterate over the rows in a view object," more precisely means you will use the default row set iterator of the view object's default row set to loop over its rows.

The Role of the Key Object in a View Row or Entity Row

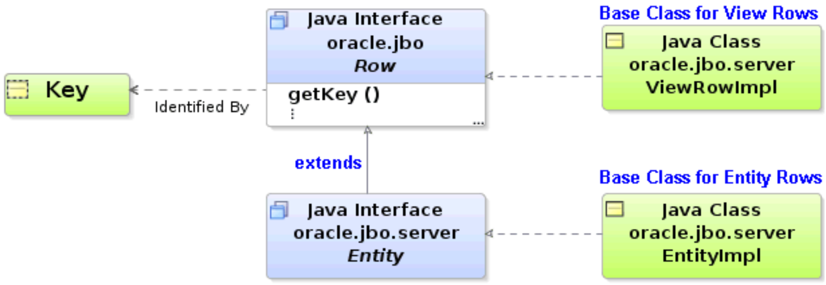
When you work with view rows you use the `Row` interface in the `oracle.jbo` package. As shown in Figure 8-19, the interface contains a method called `getKey()` that you can use to access the Key object that identifies any row. Notice that the Entity interface in the oracle.jbo.server package extends the Row interface. This relationship provides a concrete explanation of why the term *entity row* is so appropriate. Even though an entity row supports additional features for encapsulating business logic and handling database access, you can still treat any entity row as a Row.

An entity-based view object delegates the task of finding rows by key to its underlying entity row parts.

Recall that both view rows and entity rows support either single-attribute or multiattribute keys, so the Key object related to any given Row will encapsulate all of the attributes that comprise its key. Once you have a Key object, you can use the findByKey() method on any row set to find a row based on its Key object. When you use the findByKey() method to find a view row by key, the view row proceeds to use the entity definition's findByPrimaryKey() method to find each entity row contributing attributes to the view row key.

In the case of a read-only view object with no underlying entity row to which to delegate this task, the view object implementation automatically enables the manageRowsByKey flag when at least one primary key attribute is detected. This ensures that the findByKey() method is successful in the case of read-only view objects. If the manageRowsByKey flag is not enabled, then UI operations like setting the current row with the key, which depend on the findByKey() method, would not work.

Figure 8-19 Any View Row or Entity Row Supports Retrieving Its Identifying Key



Note:

When you define an entity-based view object, by default the primary key attributes for all of its entity usages are marked with their **Key Attribute** property set to true. In any *nonupdatable* reference entity usages, you should disable the **Key Attribute** property for the key attributes. Since view object attributes related to the primary keys of *updatable* entity usages must be part of the composite view row key, their **Key Attribute** property cannot be disabled.

The Role of the Entity Cache in the Transaction

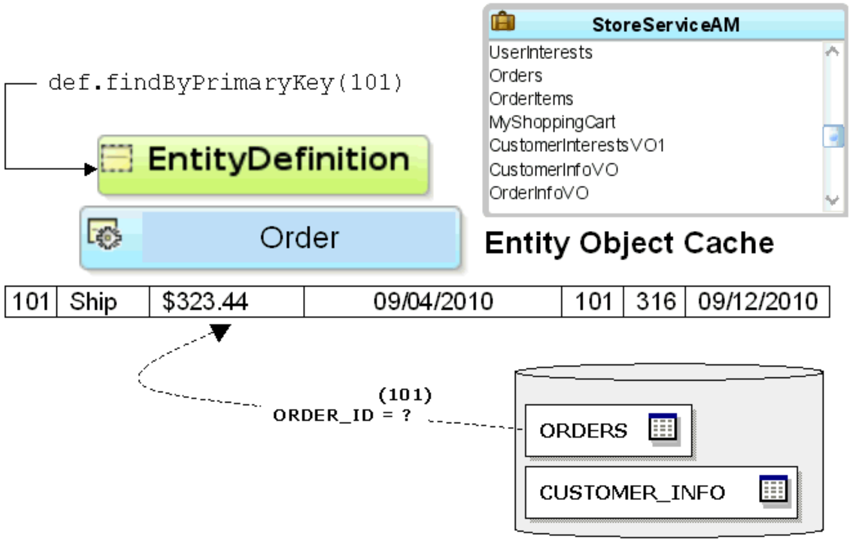
An application module is a transactional container for a logical unit of work. At runtime, it acquires a database connection using information from the named configuration you supply, and it delegates transaction management to a companion Transaction object.

Since a logical unit of work may involve finding and modifying multiple entity rows of different types, the `Transaction` object provides an entity cache as a "work area" to hold entity rows involved in the current user's transaction. Each entity cache contains rows of a single entity type, so a transaction involving two or more entity objects holds the working copies of those entity rows in separate caches.

By using an entity object's related entity definition, you can write code in an application module to find and modify existing entity rows. As shown in Figure 8-20, by calling `findByPrimaryKey()` on the entity definition for the `Order` entity object, you can retrieve the row with that key. If it is not already in the entity cache, the entity object executes a query to retrieve it from the database. This query selects all of the entity object's persistent attributes from its underlying table, and finds the row using an appropriate `WHERE` clause against the column corresponding to the entity object's primary key attribute. Subsequent attempts to find the same entity row by key during the same transaction will find it in the cache, preventing the need for a trip to the database. In a given entity cache, entity rows are indexed by their primary key. This makes finding an entity row in the cache a fast operation.

When you access related entity rows using association accessor methods, they are also retrieved from the entity cache. If related entity rows are not in the cache, then they are retrieved from the database. Finally, the entity cache is also the place where new entity rows wait to be saved. In other words, when you use the `createInstance2()` method on the entity definition to create a new entity row, it is added to the entity cache.

Figure 8-20 Entity Cache Stores Entity Rows During the Transaction



When an entity row is created, modified, or removed, it is automatically enrolled in the transaction's list of pending changes. When you call `commit()` on the `Transaction` object, it processes its pending changes list, validating new or modified entity rows that might still be invalid. When the entity rows in the pending list are all valid, the `Transaction` issues a database `SAVEPOINT` and coordinates saving the entity rows to the database. If all goes successfully, it issues the final database `COMMIT` statement. If anything fails, the `Transaction` performs a `ROLLBACK TO SAVEPOINT` to allow the user to fix the error and try again.

The Transaction object used by an application module represents the working set of entity rows for a single end-user transaction. By design, it is *not* a shared, global cache. The database engine itself is an extremely efficient shared, global cache for multiple, simultaneous users. Rather than attempting to duplicate all the work of fine-tuning that has gone into the database's shared, global cache functionality, ADF Business Components consciously embraces it. To refresh a single entity object's data from the database at any time, you can call its refresh() method. You can setClearCacheOnCommit() or setClearCacheOnRollback() on the Transaction object to control whether entity caches are cleared at commit or rollback. The defaults are false and true, respectively. The Transaction object also provides a clearEntityCache() method you can use to programmatically clear entity rows of a given entity type (or all types). When you clear an entity cache, you allow entity rows of that type to be retrieved from the database fresh the next time they are either found by primary key or retrieved by an entity-based view object.

How to Create a Command-Line Java Test Client

To the create a test client program, use the Create Java Class wizard, which is accessible from the New Gallery.

Generating a Test Client with Skeleton Code

When you use the Create Java Class wizard to create the test client program, JDeveloper will open your program file in the source editor and allow you to add code from a predefined code template to complete the test client.

Before you begin:

It may be helpful to have an understanding of programmatic testing. For more information, see Testing View Object Instances Programmatically.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Testing View Instances.

To generate a skeleton Java test client:

1. In the Applications window, right-click the project in which you want to create the test client and choose **New** and then **Java Class**.
2. In the Create Java Class dialog, enter a class name and a package name and ensure that the **Extends** field shows java.lang.Object.
3. In **Optional Attributes**, deselect **Constructors from Superclass** and select **Main Method**.
4. Click **OK**.

   The .java file opens in the source editor to show the skeleton code, as shown in the following example. In this example, TestClient is the class name and oracle.summit.model.test is the package name.

```
package oracle.summit.model.test;

public class TestClient {
  public static void main(String[] args) {
     TestClient testClient = new TestClient();
  }
}
```

Modifying the Skeleton Code to Create the Test Client

After you generate skeleton code for the test client, you can proceed to edit the file using the skeleton code template you created.

Before you begin:

It may be helpful to have an understanding of programmatic testing. For more information, see Testing View Object Instances Programmatically.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Testing View Instances.

You will need to complete this task:

Create create the test client skeleton code, as described in Generating a Test Client with Skeleton Code.

To insert the skeleton code:

1. Place the cursor on a blank line inside the body of the `main()` method and type the name of your test client class and then press Ctrl + Enter.
2. Type the characters `bc4jclient` and then press Ctrl + Enter.
3. Adjust the values of the `amDef` and`config` variables to reflect the names of the application module definition and the configuration that you want to use, respectively.

   For the example below, the changed lines look like this:

   ```
   String amDef = "oracle.summit.model.viewobjects.AppModule";
   String config = "AppModuleLocal";
   ```
4. Finally, change the view object instance name in the call to `findViewObject()` to the one you want to work with. Specify the name exactly as it appears in the **Data Model** tree on the Data Model page of the overview editor for the application module.

   the changed line looks like this:

   ```
   ViewObject customerList = am.findViewObject("SCustomerView1");
   ```

Your skeleton test client for your application module should contain source code like what you see in the following example.

> **Note:**
>
> The examples throughout Working Programmatically with an Application Module's Client Interface expand this test client sample code to illustrate calling custom application module service methods, too.

```
package oracle.summit.model.viewobjects;

import oracle.jbo.ApplicationModule;
import oracle.jbo.Row;
import oracle.jbo.RowSet;
```

```
import oracle.jbo.ViewObject;
import oracle.jbo.client.Configuration;

public class TestClient {

  public static void main(String[] args) {
    String  amDef = "oracle.summit.model.viewobjects.AppModule";
    String  config = "AppModuleLocal";
    ApplicationModule am =
                      Configuration.createRootApplicationModule(amDef,config);
    // 1. Find the Customer view object instance.
    ViewObject customerList = am.findViewObject("SCustomerView1");
    // Work with your appmodule and view object here
    Configuration.releaseRootApplicationModule(am,true);
  }
}
```

The method call `createRootApplicationModule()` allows test clients to create an **application module instance** where the **ADF Model** layer is unavailable. Normally, in the Fusion web applications your code finds the view object on the application module through **ADF binding** objects. For more information about how to work with the ADF Model layer, see How to Access an Application Module Client Interface in a Fusion Web Application.

Replace `// Work with your appmodule and view object here` , with code that will execute the view objects you want to test. For example, to execute the view object's query, display the number of rows it will return, and loop over the result to fetch the data and print it out to the console, you can adapt the code shown in the following example for your data model project components.

```
    // 2. Execute the query
    customerList.executeQuery();
    // 3. Iterate over the resulting rows
    while (customerList.hasNext()) {
        Row customer = customerList.next();
        // 4. Print the customer's name
        System.out.println("Customer: " + customer.getAttribute("Name"));
        // 5. Get related rowset of Orders using view link accessor attribute
        RowSet orders = (RowSet)customer.getAttribute("SOrdView");
        // 6. Iterate over the Orders rows
        while (orders.hasNext()) {
            Row order = orders.next();
            // 7. Print out some order attribute values
              System.out.println(" ["+order.getAttribute("CustomerId")+"] "+
                                   order.getAttribute("Id")+": "+
                                   order.getAttribute("Total"));
              if(!order.getAttribute("OrderFilled").equals("Y")) {
                  // 8. Get related rowset of OrderItems
                  RowSet items = (RowSet)order.getAttribute("SItemView");
                  // 9. Iterate over the OrderItems rows
                  while (items.hasNext()) {
                     Row item = items.next();
                     // 10. Print out some order items attributes
                     System.out.println("  "+item.getAttribute("ItemId")+": "+
                                          item.getAttribute("Price"));
                }
            }
        }
    }
```

The first line calls the `executeQuery()` method to execute the view object's query. This produces a row set of zero or more rows that you can loop over using a `while` statement that iterates until the view object's `hasNext()` method returns `false`. Inside the loop, the code puts the current `Row` in a variable named `customer`, then invokes the `getAttribute()` method twice on that current `Row` object to get the value of the `Name` and `Orders` attributes to print order information to the console. A second `while` statement performs the same task for the line items of the order.

What Happens When You Run a Test Client Program

The call to `createRootApplicationModule()` on the `Configuration` object returns an instance of the application module to work with. As you might have noticed in the debug diagnostic output, the ADF Business Components runtime classes load XML documents as necessary to instantiate the application module and the instance of the view object component that you've defined in its data model at design time. The `findViewObject()` method on the application module finds a view object instance by name from the application module's data model. After the loop described in Modifying the Skeleton Code to Create the Test Client, the test client executes `releaseRootApplicationModule()` on the `Configuration` object. This signals that you're done using the application module and it allows the framework to clean up resources, like the database connection that was used by the application module.

What You May Need to Know About Running a Test Client

The `createRootApplicationModule()` and `releaseRootApplicationModule()` methods are very useful for command-line access to application module components. However, you typically won't need to write these two lines of code in the context of an ADF-based web application. The ADF Model layer cooperates automatically with the ADF **business services layer** to acquire and release ADF application module components for you in those scenarios. For more information about how to work with the ADF Model layer, see How to Access an Application Module Client Interface in a Fusion Web Application.

How to Count the Number of Rows in a Row Set

The `getEstimatedRowCount()` method is used on a `RowSet` to determine how many rows it contains:

```
long numOrders = orders.getEstimatedRowCount();
```

The implementation of the `getEstimatedRowCount()` initially issues a `SELECT COUNT(*)` query to calculate the number of rows that the query will return. The query is formulated by "wrapping" your view object's entire query in a statement like:

```
SELECT COUNT(*) FROM ( ... your view object's SQL query here ... )
```

The `SELECT COUNT(*)` query allows you to access the count of rows for a view object without necessarily retrieving all the rows themselves. This approach permits an important optimization for working with queries that return a large number of rows, or for testing how many rows a query *would* return before proceeding to work with the results of the query.

Once the estimated row count is calculated, subsequent calls to the method do not reexecute the `COUNT(*)` query. The value is cached until the next time the view object's query is executed, since the fresh query result set returned from the database could potentially contain more, fewer, or different rows compared with the last time the query was run. The estimated row count is automatically adjusted to account for pending changes in the current transaction, adding the number of relevant new rows and subtracting the number of removed rows from the count returned.

You can also override `getEstimatedRowCount()` to perform a custom count query that suits your application's needs.

How to Access a Detail Collection Using the View Link Accessor

Once you've retrieved the `RowSet` of detail rows using a **view link accessor**, as described in Programmatically Accessing a Detail Collection Using the View Link Accessor, you can loop over the rows it contains using the same pattern used by the view object's row set of results, as shown in the following example.

```
while (orders.hasNext()) {
  Row curOrder = orders.next();
  System.out.println("--> (" + curOrder.getAttribute("Id") + ") " +
                     curOrder.getAttribute("Total"));
}
```

The following example shows the `main()` method sets a dynamic `WHERE` clause to restrict the `CustomerView` view object instance to show only customers whose `credit_rating_id` has the value `4`. Additionally, the `executeAndShowResults()` method accesses the view link accessor attribute (`SOrdView`) and prints out the order `Id` and `Total` attribute for each customer.

To access the a detail collection using a view link accessor, follow these basic steps (as illustrated in the following example):

1. Find the master view object instance.
2. Execute the query.
3. Iterate over the master view object rows.
4. Get the related row set of the detail view object using the view link accessor attribute.
5. Iterate over the detail view object rows.
6. Optionally, do something with the detail row set attributes.

> **Performance Tip:**
>
> If the code you write to loop over the rows does not need to display them, then you can call the `closeRowSet()` method on the row set when you're done. This technique will make memory use more efficient. The next time you access the row set, its query will be reexecuted.

```
package oracle.summit.model.viewobjects;

import oracle.jbo.ApplicationModule;
import oracle.jbo.Row;
import oracle.jbo.RowSet;
import oracle.jbo.ViewObject;
import oracle.jbo.client.Configuration;
```

```
public class TestClient2 {
    public static void main(String[] args) {
            String        amDef = "oracle.summit.model.viewobjects.AppModule";
            String config = "AppModuleLocal";
            ApplicationModule am =
                Configuration.createRootApplicationModule(amDef, config);
            ViewObject vo = am.findViewObject("SCustomerView1");
          // Add an extra where clause with a new named bind variable
            vo.setWhereClause("credit_rating_id = :theCreditRating");
            vo.defineNamedWhereClauseParam("theCreditRating", null, null);
            vo.setNamedWhereClauseParam("theCreditRating", "4");
          // Show results when :theCreditRating = '4'
          executeAndShowResults(vo);
          Configuration.releaseRootApplicationModule(am, true);
        }
        private static void executeAndShowResults(ViewObject vo) {
          System.out.println("---");
          vo.executeQuery();
          while (vo.hasNext()) {
            Row curCustomer = vo.next();
            // Access the row set of details using the view link accessor attr.
            RowSet orders = (RowSet)curCustomer.getAttribute("SOrdView");
            long numOrders = orders.getEstimatedRowCount();
            System.out.println(curCustomer.getAttribute("Id") + " " +
                               curCustomer.getAttribute("Name")+" ["+
                               numOrders+" orders]");
            while (orders.hasNext()) {
              Row curOrder = orders.next();
              System.out.println("--> (" + curOrder.getAttribute("Id") + ") " +
                                 curOrder.getAttribute("Total"));
            }
          }
        }
}
```

Running `TestClient2.java` produces output in the Log window, as shown in the following example. Each customer is listed, and for each customer that has some orders, the order ID and total appears beneath their name.

```
---
202 Simms Athletics [11 orders]
--> (98) 595
--> (113) 4990
--> (114) 567
--> (115) 866.7
--> (116) 3661.24
--> (117) 17489
212 Hamada Sport [22 orders]
--> (108) 149570
--> (196) 761
--> (197) 516.75
...
```

If you run `TestClient2.java` with debug diagnostics enabled, you will see the SQL queries that the view object performed. The view link `WHERE` clause predicate is used to automatically perform the filtering of the detail service request rows for the current row in the `CustomerView` view object.

How to Iterate Over a Master-Detail-Detail Hierarchy

To iterate over a master-detail with an additional level of nesting, follow these basic steps (as illustrated in the following example):

1. Find the master view object instance.
2. Executes the query.
3. Iterate over the resulting rows.
4. Optionally, do something with the attributes of the master row set.
5. Get the related row set of the detail view object using the view link accessor attribute.
6. Iterate over the detail row set rows.
7. Optionally, do something with the attributes of the detail row set.
8. Get the related row set of the second detail view object using the view link accessor attribute.
9. Iterates over the second detail row set rows.
10. Optionally, do something with the second detail row set attributes.

Other than having one additional level of nesting, the following example uses the same API's used in the `TestClient` program that was iterating over master-detail read-only view objects in How to Access a Detail Collection Using the View Link Accessor.

If you use JDeveloper's **Refactor > Duplicate** functionality on an existing `TestClient.java` class, you can quickly "clone" it to create a `TestClient2.java` class. For example, the `TestClient.java` class described in How to Access a Detail Collection Using the View Link Accessor is suited to this technique.

```
package oracle.summit.model.viewobjects;

import oracle.jbo.ApplicationModule;
import oracle.jbo.Row;
import oracle.jbo.RowSet;
import oracle.jbo.ViewObject;
import oracle.jbo.client.Configuration;

public class TestClient {
    public static void main(String[] args) {
        TestClient testClient = new TestClient();
        String amDef = "oracle.summit.model.viewobjects.AppModule";
        String config = "AppModuleLocal";
        ApplicationModule am =

Configuration.createRootApplicationModule(amDef,config);
        // Work with your appmodule and view object here
        // 1. Find the Customer view object instance.
        ViewObject customerList = am.findViewObject("SCustomerView1");
        // 2. Execute the query
          customerList.executeQuery();
          // 3. Iterate over the resulting rows
          while (customerList.hasNext()) {
              Row customer = customerList.next();
              // 4. Print the person's email
              System.out.println("Customer: " + customer.getAttribute("Name"));
              // 5. Get related rowset of Orders using view link accessor attr.
              RowSet orders = (RowSet)customer.getAttribute("SOrdView");
              // 6. Iterate over the Orders rows
              while (orders.hasNext()) {
                  Row order = orders.next();
                  // 7. Print out some order attribute values
                    System.out.println(" ["+order.getAttribute("CustomerId")+"]
"+
                                          order.getAttribute("Id")+": "+
                                          order.getAttribute("Total"));
                    if(order.getAttribute("OrderFilled").equals("Y")) {
                        // 8. Get related rowset of Items
                        RowSet items = (RowSet)order.getAttribute("SItemView");
                        // 9. Iterate over the Items rows
                        while (items.hasNext()) {
                           Row item = items.next();
                           // 10. Print out some order items attributes
                           System.out.println("  "+
                                                item.getAttribute("ItemId")+":
$"+

                                                item.getAttribute("Price"));
                      }
                  }
              }
          }
    }
}
```

Running the program produces the output shown in the following example.

```
Customer: Unisports
 [201] 97: 84000
  210: $9
  211: $1500
Customer: Simms Athletics
 [202] 98: 595
  212: $85
 [202] 113: 4990
  328: $9
 [202] 114: 567
  329: $28
  330: $40.95
  331: $25
...
```

How to Find a Row and Update a Foreign Key Value

To find a row and update a foreign key value, follow these basic steps (as illustrated in the following example):

1. Find the view object instance.
2. Construct a `Key` object to look up the row for the view instance.
3. Use `findByKey()` to find the row.
4. Optionally, do something with the row's attribute.

The following example shows the `main()` method finds and updates a foreign key value to find a row of the `Orders` view object instance. The sample then prints out the existing value of the `OrderStatusCode` attribute before changing the value on the row.

```
package oracle.summit.model.viewobjects;

import java.text.ParseException;
import java.text.SimpleDateFormat;

import java.util.Calendar;
import java.util.Date;

import oracle.jbo.ApplicationModule;
import oracle.jbo.JboException;
import oracle.jbo.Key;
import oracle.jbo.Row;
import oracle.jbo.RowSet;
import oracle.jbo.ViewObject;
import oracle.jbo.client.Configuration;

public class TestClient3 {
    public TestClient3() {
        super();
    }

    public static void main(String[] args) {
        String amDef = "oracle.summit.model.viewobjects.AppModule";
        String config = "AppModuleLocal";
        ApplicationModule am =
                         Configuration.createRootApplicationModule(amDef, 
config);
        // 1. Find the Orders view object instance
        ViewObject vo = am.findViewObject("SOrdView1");
        // 2. Construct a new Key to find Order # 100
        Key orderKey = new Key(new Object[] { 100 });
        // 3. Find the row matching this key
        Row[] ordersFound = vo.findByKey(orderKey, 1);
        if (ordersFound != null && ordersFound.length > 0) {
            Row order = ordersFound[0];
            // 4. Print some order information
            Date dateOrdered = (Date) order.getAttribute("DateOrdered");
            String orderDate = dateOrdered.toString();
            System.out.println("Order Date is: " + orderDate);

            SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");
            Calendar c = Calendar.getInstance();
            try {
                c.setTime(sdf.parse(orderDate));
            } catch (ParseException e) {
            }
            c.add(Calendar.DAY_OF_MONTH, -2);
            // // number of days to add
            orderDate = sdf.format(c.getTime());
            System.out.println(orderDate);
            try {
                // 5. Try setting the ship date to an illegal value
                order.setAttribute("DateShipped", orderDate);
                am.getTransaction().commit();
            } catch (JboException ex) {
                System.out.println("ERROR: " + ex.getMessage());
            }
            // 6. Set the ship date to a legal value
            order.setAttribute("DateShipped", orderDate);
            // 7. Show the value of the ship date was updated successfully
            System.out.println("Ship date is: " +
                                                  order.getAttribute("DateShipped"));
                // 8. Show the current value of the customer for this order
                System.out.println("Customer: " + order.getAttribute("CustomerId"));
                // 9. Reassign the order to customer # 210
                order.setAttribute("CustomerId", 210);
                // 10. Show the value of the reference information now
                System.out.println("Customer: " + order.getAttribute("CustomerId"));
                // 11. Rollback the transaction
                am.getTransaction().rollback();
                System.out.println("Transaction canceled");
            }
            Configuration.releaseRootApplicationModule(am, true);
        }

}
```

Running this example produces the output shown in the following example.

```
Order Date is: 2012-08-29
2012-08-27
ERROR: JBO-oracle.summit.model.viewobjects.SOrd_Rule_0:
       You cannot have a shipping date that is before the order date
Ship date is: 2012-08-27
Customer: 204
Customer: 210
Transaction canceled
```

How to Create a New Row for a View Object Instance

To create a new view row instance, follow these basic steps (as illustrated in the following example):

1. Find the view object instance.
2. Create a new row and insert it into the row set.
3. Set the values of the required attributes in the new row.
4. Commit the transaction.

The following example shows the `main()` method finds the `OrdView` view object instance and inserts a new row into the row set. Because the `OrdView` view object is entity-based, the `CreatedBy` attribute derives its value from the mapped entity object attribute. The sample then sets values for the remaining attributes before committing the transaction.

```
package oracle.summit.model.viewobjects;

import java.util.Date;

import oracle.jbo.ApplicationModule;
import oracle.jbo.Row;
import oracle.jbo.ViewObject;
import oracle.jbo.client.Configuration;
import oracle.jbo.domain.DBSequence;

public class TestClient4 {

  public static void main(String[] args) {
        String amDef = "oracle.summit.model.viewobjects.AppModule";
        String config = "AppModuleLocal";
        ApplicationModule am =
                         Configuration.createRootApplicationModule(amDef,
config);

        // 1. Find the Orders view object instance.
        ViewObject orders = am.findViewObject("SOrdView1");

        // 2. Create a new row and insert it into the row set
        Row newOrder = orders.createRow();
        orders.insertRow(newOrder);
        // Show the entity object-related defaulting for DateOrdered attribute
        System.out.println("DateOrdered defaults to: " +
                                    newOrder.getAttribute("DateOrdered"));

        // 3. Set values for some of the required attributes
        newOrder.setAttribute("CustomerId", 201);
        newOrder.setAttribute("SalesRepId", 12);
        newOrder.setAttribute("PaymentTypeId", 2);
        newOrder.setAttribute("OrderFilled", "N");

        // 4. Commit the transaction
        am.getTransaction().commit();

        // 5. Retrieve and display the trigger-assigned order id
        DBSequence id = (DBSequence)newOrder.getAttribute("Id");
        System.out.println("Thanks, reference number is " +
                                                  id.getSequenceNumber());
        Configuration.releaseRootApplicationModule(am, true);
  }
}
```

Running this example produces the results shown in the following example.

```
DateOrdered defaults to: 2013-3-11
Thanks, reference number is 6
```

How to Retrieve the Row Key Identifying a Row

To retrieve a row key to identify a row, follow these basic steps (as illustrated in the following example):

1. Find the view object instance.
2. Construct a key using a supplied value.
3. Find the row with this key.
4. Optionally, do something with the key of the row.

The following example shows the `main()` method finds the `OrdView` view object instance and constructs a row key to find an order number. The `findByKey()` method find the `OrdView` rows with the specified key. The sample then displays the key of the row, accesses the row set using the `ItemView` view link accessor, and iterates over the rows to display the key of each `ItemView` row.

```
package oracle.summit.model.viewobjects;

import oracle.jbo.ApplicationModule;
import oracle.jbo.Key;
import oracle.jbo.Row;
import oracle.jbo.RowSet;
import oracle.jbo.ViewObject;
import oracle.jbo.client.Configuration;

public class TestClient5 {
    public static void main(String[] args) {
        String amDef = "oracle.summit.model.viewobjects.AppModule";
        String config = "AppModuleLocal";
        ApplicationModule am =
                        Configuration.createRootApplicationModule(amDef,
config);
        // 1. Find the Orders view object instance
        ViewObject vo = am.findViewObject("SOrdView1");
        // 2. Construct a key to find order number 100
        Key orderKey = new Key(new Object[] { 100 });
        // 3. Find the Orders row with the key
        Row[] ordersFound = vo.findByKey(orderKey, 1);
        if (ordersFound != null && ordersFound.length > 0) {
             Row order = ordersFound[0];
             // 4. Displays the key of the Orders row
             showKeyFor(order);
             // 5. Accesses row set of Orders using OrderItems view link accessor
             RowSet items = (RowSet)order.getAttribute("SItemView");
             // 6. Iterates over the Items row
             while (items.hasNext()) {
                 Row itemRow = items.next();
                 // 4. Displays the key of each Items row
                 showKeyFor(itemRow);
             }
        }
        Configuration.releaseRootApplicationModule(am, true);
    }

    private static void showKeyFor(Row r) {
       // get the key for the row passed in
       Key k = r.getKey();
       // format the key as "(val1,val2)"
       String keyAttrs = formatKeyAttributeNamesAndValues(k);
       // get the serialized string format of the key, too
       String keyStringFmt = r.getKey().toStringFormat(false);
       System.out.println("Key " + keyAttrs + " has string format " +
                          keyStringFmt);
    }

    // Build up "(val1,val2)" string for key attributes
    private static String formatKeyAttributeNamesAndValues(Key k) {
       StringBuffer sb = new StringBuffer("(");
       int attrsInKey = k.getAttributeCount();
       for (int i = 0; i < attrsInKey; i++) {
           if (i > 0)
               sb.append(",");
               sb.append(k.getAttributeValues()[i]);
       }
       sb.append(")");
       return sb.toString();
    }
}
```

Running the example produces the results shown in the following example. Notice that the serialized string format of a key is a hexadecimal number that includes information in a single string that represents all the attributes in a key.

```
Key (100) has string format 000100000003313030
Key (100,156) has string format 000200000003C2020200000002C102
Key (100,157) has string format 000200000003C2020200000002C102
...
```

What You May Need to Know About Using Partial Keys with findByKey()

View objects based on multiple entity usages support the ability to find view rows by specifying a partially populated key. A partial key is a multi-attribute `Key` object with some of its attributes set to `null`. However, there are strict rules about what kinds of partial keys can be used to perform the `findByKey()`.

If a view object is based on *n* entity usages, where *n* > 1, then the view row key is by default comprised of all of the primary key attributes from *all* of the participating entity usages. Only the ones from the *first* entity object are *required* to participate in the view row key, but by default all of them do.

If you allow the key attributes from some *secondary* entity usages to remain as key attributes at the view row level, then you should leave *all* of the attributes that form the primary key for that entity object as part of the view row key. Assuming you have left the one or more key attributes in the view row for *m* of the *n* entity usages, where (*m* <= *n*), then you can use `findByKey()` to find rows based on any subset of these *m* entity usages. Each entity usage for which you provide values in the `Key` object, requires that you must provide non-null values for *all* of the attributes in that entity's primary key.

You have to follow this rule because when a view object is based on at least one or more entity usages, its `findByKey()` method finds rows by delegating to the `findByPrimaryKey()` method on the entity definition corresponding to the first entity usage whose attributes in the view row key are non-null. The entity definition's `findByPrimaryKey()` method requires all key attributes for any given entity object to be non-null in order to find the entity row in the cache.

As a concrete example, imagine that you have a `OrderInfoVO` view object with a `OrderEO` entity object as its primary entity usage, and an `AddressEO` entity as secondary reference entity usage. Furthermore, assume that you leave the **Key Attribute** property of *both* of the following view row attributes set to `true`:

- `OrderId` — primary key for the `OrderEO` entity
- `AddressId` — primary key for the `AddressEO` entity

The view row key will therefore be the (`OrderId`, `AddressId`) combination. When you do a `findByKey()`, you can provide a `Key` object that provides:

- A completely specified key for the underlying `OrderEO` entity

  ```
  Key k = new Key(new Object[]{new Number(200), null});
  ```

- A completely specified key for the underlying `AddressEO` entity

  ```
  Key k = new Key(new Object[]{null, new Number(118)});
  ```

- A completely specified key for both entities

  ```
  Key k = new Key(new Object[]{new Number(200), new Number(118)});
  ```

When a valid partial key is specified, the `findByKey()` method can return multiple rows as a result, treating the missing entity usage attributes in the Key object as a wildcard.

How to Authenticate Test Users in the Test Client

If you have enabled ADF Security for your application and provisioned the `jazn-data.xml` file with test users, you will need to include method calls to authenticate a user before you run the test client. To authenticate a user in the test client, follow these basic steps (as illustrated in the following exxample):

1. Create the authentication service.
2. Pass in the login credentials for a test user defined in the `jazn-data.xml` file.
3. If authentication succeeds, then test the application module.
4. Log the user out.

For details about how to run the Configure ADF Security wizard to enable ADF Security and how to create test users in JDeveloper's identity store, see ADF Security Process Overview.

```
package oracle.summit.model.test;
import oracle.jbo.ApplicationModule;
import oracle.jbo.Key;
import oracle.jbo.Row;
import oracle.jbo.RowSet;
import oracle.jbo.ViewObject;
import oracle.jbo.client.Configuration;

import oracle.adf.share.security.AuthenticationService;
import oracle.adf.share.security.authentication.AuthenticationServiceUtil;
import oracle.adf.share.ADFContext;
import oracle.adf.share.security.SecurityContext;
import oracle.adf.share.security.SecurityEnv;

import javax.security.auth.Subject;

public class TestAuthenticationClient {
    public static void main(String[] args) {
        String amDef = "test.TestAuthModule";
        String config = "TestAuthModuleLocal";

        // 1. Create authentication service.
        AuthenticationService authService =
                            AuthenticationServiceUtil.getAuthenticationService();
        try
           {
              // 2. Pass in user id and password defined in local identity store.
              authService.login("tester1", "welcome1");
           }
           catch (Exception e)
           {

           }

        // Uncomment to output authentication status
        // String userName =
                       ADFContext.getCurrent().getSecurityContext().getUserName();

        // System.out.println("*** userName : " + userName);
        // Subject subject =
                       ADFContext.getCurrent().getSecurityContext().getSubject();
        // System.out.println("Subject : " + subject);

        // 3. Test application module if authentication succeeds.
        if (ADFContext.getCurrent().getSecurityContext().isAuthenticated()) {
           ApplicationModule am =
                       Configuration.createRootApplicationModule(amDef,config);
           ViewObject vo = am.findViewObject("TestView");
           // Work with your appmodule and view object here
           Configuration.releaseRootApplicationModule(am,true);

           // 4. Log out test user.
           authService.logout();

           // Uncomment to report logout success
           // boolean isAuthenticated =
                   ADFContext.getCurrent().getSecurityContext().isAuthenticated();
           // System.out.println("*** isAuthenticated : " + isAuthenticated);
        }

    }
}
```

9

Tuning View Object Performance

This chapter describes advanced techniques you can use while designing and working with your ADF view objects in an Oracle ADF application.
This chapter includes the following sections:

About View Object Tuning

JDeveloper provides you many tools to configure ADF Business Components view objects and improve their performance during transactions.

You can use view objects to read rows of data, create and store rows of transient data, as well as automatically coordinate inserts, updates, and deletes made by end users with your underlying business objects. How you design and use your view objects can definitely affect their performance at runtime. The JDeveloper design time for ADF Business Components provides a number of features that you can use to configure your view objects to get the best possible performance.

Maintaining New Row Consistency Between View Objects Based on the Same Entity

ADF Business Components automatically inserts a new row created in one view object instance into other view object instances based on the same entity object.

When multiple instances of entity-based view objects in an **application module** are based on the same underlying **entity object**, a new row created in one of them can be automatically added (without having to requery) to the row sets of the others to keep your user interface consistent or simply to consistently reflect new rows in different application pages for a pending transaction. Consider an application that displays a customer's list of orders. If the customer goes to create a new order, this task is performed through a different view object and handled by a custom application module method. Using the view object new row consistency feature, the newly created order automatically appears in the customer's list of open orders without having to requery the database.

For historical reasons, this capability is known as the **view link consistency** feature because in prior releases of **Oracle Application Development Framework** (Oracle ADF) the addition of new rows to other relevant row sets only was supported for detail view object instances in a view link based on an association. Now this view link consistency feature works for any view objects for which it is enabled, regardless of whether they are involved in a view link or not.

What Happens at Runtime: When View Link Consistency is Enabled

Consider two entity-based view objects `OrdersViewSummary` and `OrdersView` both based on the same underlying `Orders` entity object. When a new row is created in a row set for one of these view objects (like `OrdersView`) and the row's primary key is set, any of the *other* row sets for view objects based on the same `Orders` entity object (like `OrdersViewSummary`) receive an event indicating a new row has been created. If their view link consistency flag is enabled, then a copy of the new row is inserted into their row set as well.

> **Note:**
>
> By default the view link consistency mechanism adds new rows in an unqualified way. If you want to prevent the new rows from being added to the row sets of all participating view objects, you can write a filter expression on the RowMatch object to qualify the row before it is added to the row set. For more information, see Using RowMatch to Qualify Which New, Unposted Rows Get Added to a Row Set.

What You May Need to Know About Changing the Default View Link Consistency Setting

You can use the overview editor for application module configurations (on the `bc4j.xcfg` file) to control the default setting for the view link consistency feature using the `jbo.viewlink.consistent` configuration parameter.

To display the application module configuration overview editor, double-click the application module in the Applications window and, in the overview editor, select the **Configurations** navigation tab. Then, in the Configurations page of the overview editor, click the configuration hyperlink. In the application module configuration overview editor, select the **Properties** tab and click **Add Property** to select the property from the Add Property dialog and click **OK**.

The default setting for this parameter is the word `DEFAULT`which has the following meaning. If your view object has:

- A single entity usage, view link consistency is enabled
- Multiple entity usages, and:
  - If all secondary entity usages are marked as contributing reference information, then view link consistency is enabled
  - If any secondary entity usage marked as *not* being a reference view link consistency is disabled.

You can globally disable this feature by setting the `jbo.viewlink.consistent` to the value `false` in your configuration. Conversely, you could globally enable this feature by setting `jbo.viewlink.consistent` to the value `true`, but Oracle does *not* recommend doing this. Doing so would force view link consistency to be set on for view objects with secondary entity usages that are not marked as a reference which presently do not support the view link consistency feature well.

To set the feature programmatically, use the `setAssociationConsistent()` API on any `RowSet`. When you call this method on a view object, it affects its default row set.

Using RowMatch to Qualify Which New, Unposted Rows Get Added to a Row Set

If a view object has view link consistency enabled, any new row created by another view object based on the same entity object is added to its row set. By default the mechanism adds new rows in an unqualified way. If your view object has a design-time `WHERE` clause that queries only a certain subset of rows, you can apply a `RowMatch` object to your view object to perform the same filtering in-memory. The filtering expression of the `RowMatch` object you specify prevents new rows from being added that wouldn't make sense to appear in that view object.

For example, an `OrdersByStatus` view object might include a design time `WHERE` clause like this:

```
WHERE /* ... */ AND STATUS LIKE NVL(:StatusCode,'%')
```

Its custom Java class overrides the `create()` method as shown in the following example to force view link consistency to be enabled. It also applies a `RowMatch` object whose filtering expression matches rows whose `Status` attribute matches the value of the `:StatusCode` named bind variable (or matches any row if `:StatusCode` = '%'). This `RowMatch` filter is used by the view link consistency mechanism to qualify the row that is a candidate to add to the row set. If the row qualifies by the `RowMatch`, it is added. Otherwise, it is not.

```
// In OrdersByStatusImpl.java
protected void create() {
  super.create();
  setAssociationConsistent(true);
  setRowMatch(new RowMatch("Status = :StatusCode or :StatusCode = '%'"));
}
```

See Performing In-Memory Filtering with RowMatch for more information on creating and using a `RowMatch` object. For a list of supported SQL operators see Table 11-2. For a list of supported SQL function, see Table 11-3.

> **Note:**
>
> If the `RowMatch` facility does not provide enough control, you can override the view object's `rowQualifies()` method to implement a custom filtering solution. Your code can determine whether a new row qualifies to be added by the view link consistency mechanism or not.

What You May Need to Know About the Dynamic WHERE Clause and View Link Consistency

If you call `setWhereClause()` on a view object to set a dynamic `WHERE` clause, the view link consistency feature is disabled on that view object. If you have provided an appropriate custom `RowMatch` object to qualify new rows for adding to the row set, you can call `setAssociationConsistent(true)` after `setWhereClause()` to reenable view link consistency.

If a row set has view link consistency enabled, then new rows added due to creation by other row sets are added to the bottom of the row set.

If a row set has view link consistency enabled, then when you call the `executeQuery()` method, any qualifying new, unposted rows are added to the top of the row set before the queried rows from the database are added.

Using Bind Variables for Parameterized Queries

Use bind variables when values of the `WHERE` clause can change with every execution.

Whenever the `WHERE` clause of your query includes values that might change from execution to execution, you should use named bind variables. The Create View Criteria dialog that you display from the Query page of the view object overview editor makes this an easy task. Their use also protects your application against abuse through SQL injection attacks by malicious end-users. For information about defining **view criteria** with bind variables, see How to Create Named View Criteria Declaratively.

Use Bind Variables to Avoid Reparsing of Queries

Bind variables are place holders in the SQL string whose value you can easily change at runtime without altering the text of the SQL string itself. Since the query text doesn't change from execution to execution, the database can efficiently reuse the same parsed statement each time. Avoiding reparsing of your statement alleviates the database from having to continually re-determine its query optimization plan and eliminates contention by multiple end-users on other costly database resources used during this parsing operation. This savings leads to higher runtime performance of your application. See How to Add WHERE Clause Bind Variables to a View Object Definition for details on how to use named bind variables.

Use Bind Variables to Prevent SQL-Injection Attacks

Using bind variables for parameterized WHERE clause values is especially important if their values will be supplied by *end-users* of your application. Consider the following example. It adds a dynamic WHERE clause formed by concatenating a user-supplied parameter value into the statement.

```
// EXAMPLE OF BAD PRACTICE, Do not follow this approach!
String userSuppliedValue = ... ;
yourViewObject.setWhereClause("BANK_ACCOUNT_ID = "+userSuppliedValue);
```

A user with malicious intentions — if able to learn any details about your application's underlying database schema — could supply a carefully constructed "bank account number" as a field value or URL parameter like:

```
BANK_ACCOUNT_ID
```

When the code in the above example concatenates this value into the dynamically applied where clause, what the database sees is a query predicate like this:

```
WHERE (BANK_ACCOUNT_ID = BANK_ACCOUNT_ID)
```

This WHERE clause retrieves *all* bank accounts instead of just the current user's, perhaps allowing the hacker to view private information of another person's account. This technique of short-circuiting an application's WHERE clause by trying to supply a maliciously constructed parameter value into a SQL statement is called a SQL injection attack. Using named bind variables instead for these situations as shown in the followign example prevents the vulnerability.

```
// Best practice using named bind variables
String userSuppliedValue = ... ;
yourViewObject.setWhereClause("BANK_ACCOUNT_ID = :BankAcccountId");
yourViewObject.defineNamedWhereClauseParam("BankAcccountId", null, null);
yourViewObject.setNamedWhereClauseParam("BankAcccountId",userSuppliedValue);
```

If a malicious user supplies an illegal value in this case, they receive an error your application can handle instead of obtaining data they are not suppose to see.

Log Bind Parameter Values

When you want to be able to determine the bind parameter values that are used to execute the queries of your view objects, you can configure the log level before a test run or during a debug session. The log level you specify will determine the type and quantity of log messages. To view bind parameter information, you must set a minimum Java log level of CONFIG. For details about using and configuring the Log Analyzer, see How to Use the Log Analyzer to View Log Messages.

Alternatively, you can log bind parameter values programmatically, by overriding the bindParametersForCollection() method on the view object's ViewObjectImpl class. This method is used to log bind parameter names and values used in the query. Your implementation may log the bind parameter values, which is especially useful in those scenarios where the bind parameter is generated by the runtime. For example, when an ADF Faces Tree component is built using view link association, the nodes will be populated by bind parameters that you may log. The following example shows how to define the bindParametersForCollection() method for this purpose.

```
@Override protected void bindParametersForCollection(QueryCollection qc,
                                               Object[] params,
                                               PreparedStatement stmt) throws
SQLException {        String vrsiName = null;
        if (qc != null) {
            ViewRowSetImpl vrsi = qc.getRowSetImpl();
            vrsiName = vrsi.isDefaultRS() ? "<Default>" : vrsi.getName();
        }
        String voName = getName();
        String voDefName = getDefFullName();
        if (qc != null) {
            System.out.println("----[Exec query for VO=" + voName + ", RS=" +
                               vrsiName + "]----");
        } else {
            System.out.println("----[Exec COUNT query for VO=" + voName +
                               "]----");
        }
        System.out.println("VODef =" + voDefName);
        System.out.println(getQuery());
        if (params != null) {
            if (getBindingStyle() == SQLBuilder.BINDING_STYLE_ORACLE_NAME) {
                StringBuilder binds = new StringBuilder("BindVars:(");
                int paramNum = 0;
                for (Object param : params) {
                    paramNum++;
                    Object[] nameValue = (Object[])param;
                    String name = (String)nameValue[0];
                    Object value = nameValue[1];
                    if (paramNum > 1) {
                        binds.append(",");
                    }
                    binds.append(name).append("=").append(value);
                }
                binds.append(")");
                System.out.println(binds);
            }
        }
        super.bindParametersForCollection(qc, params, stmt);
```

Working with Multiple Row Sets and Row Set Iterators

JDeveloper allows you to create secondary, named rowsets and secondary, named rows set iterators.

While you typically work with a view object's *default* row set, you can call the `createRowSet()` method on the `ViewObject` interface to create secondary, named row sets based on the same view object's query. One situation where this could make sense is when your view object's SQL query contains named bind variables. Since each `RowSet` object stores its own copy of bind variable values, you could use a single view object to produce and process multiple row sets based on different combinations of bind variable values. You can find a named row set you've created using the `findRowSet()` method. When you're done using a secondary row set, call its `closeRowSet()` method.

For any `RowSet`, while you typically work with its *default* row set iterator, you can call the `createRowSetIterator()` method of the `RowSet` interface to create secondary, named row set iterators. You can find a named row set iterator you've created using the `findRowSetIterator()` method. When you're done using a secondary row set iterator, call its `closeRowSetIterator()` method.

> **Performance Tip:**
>
> When you need to perform programmatic iteration over a result set, create a secondary iterator to avoid disturbing the current row of the *default* row set iterator. For example, through the ADF Model declarative data binding layer, user interface pages in your application work with the default row set iterator of the default row set of view objects in the application module's data model. In this scenario, if you did not create a secondary row set iterator for the business logic you write to iterate over a view object's default row set, you would consequently change the current row of the *default* row set iterator used by the user interface layer.

Using Entity-Based View Objects for Read-Only Data

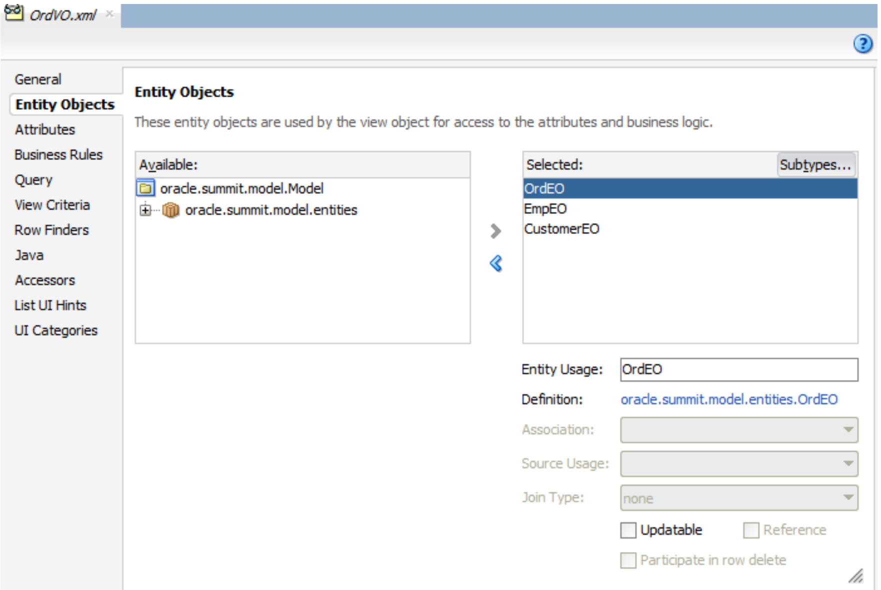
Depending on functional requirements, JDeveloper allows you to design read-only view objects. You use such view objects for SQL-validation and displaying list of values in a selection component to name a few.

Typically view objects used for SQL-based validation purposes, as well as for displaying the list of valid selections in a dropdown list, can be read-only. You need to decide what kind of functionality your application requires and design the view object accordingly.

> **Best Practice:**
>
> In previous releases, an Oracle best practice recommendation for read-only data had been to create read-only, custom SQL view objects. The purpose of this recommendation had been to avoid the overhead of storing the relating rows in an entity object cache. Oracle has now optimized a solution such that this best practice is no longer relevant.
>
> In the current release, when you need to create a read-only view object for data lookup, use the entity-based view object and deselect the **Updatable** option in the Entity Objects page of the view object overview editor. This approach has two benefits: First, it benefits from the design time editors which aid in generating the SQL query, and second, if you ever decide to make the view object updatable, it is already backed by an updatable entity object.
>
> The alternative of creating a read-only, custom SQL view object still has merit when you can specify a fully fledged SQL query, and will be especially useful for cases where Unions and Group By Queries cannot be expressed using entity objects.

View objects can either be related to underlying entity objects or not. When a view object is related to one or more underlying entity objects the default behavior supports creating new rows and modifying or removing queried rows. However, the update feature can be disabled by deselecting **Updatable** in the overview editor for the entity-based view object, as shown in Figure 9-1.

Figure 9-1 Deselecting the Updatable Option for an Entity-based View Object



Entity Objects

These entity objects are used by the view object for access to the attributes and business logic.

Available:
- oracle.summit.model.Model
- oracle.summit.model.entities

Selected:
- OrdEO
- EmpEO
- CustomerEO

Entity Usage: OrdEO

Definition: oracle.summit.model.entities.OrdEO

Association:

Source Usage:

Join Type: none

☐ Updatable ☐ Reference

☐ Participate in row delete

The alternative is to create a read-only view object and define the SQL query by selecting **Write Custom SQL** in the Query page of the overview editor. For the Business Components developer not comfortable with constructing a complex SQL statement, it will always be more convenient to create a nonupdatable view object based on an entity object since the editor simplifies the task of creating the query. Entity-based view objects that you set to nonupdatable compare favorably to read-only, custom SQL view objects:

- There is the ability to optimize the select list at runtime to include only those attributes that are required by the user interface
- There is no significant performance degradation incurred by using the entity object to create the local cache
- The data in the view object will reflect the state of the local cache rather than need to return to the database for each read operation
- The data in the local cache will stay consistent should another view object you define need to perform an update on the nonupdatable view object's base entity object.

So, while there is a small amount of runtime overhead associated with the coordination between view object rows and entity object rows (estimates show less than 5% overhead), weigh this against the ability to keep the view object definition entirely declarative and maintain a customizable view object. Custom SQL view objects are not customizable but they can be used to perform Unions and Group By queries that cannot be expressed in entity objects. Custom SQL view objects are also useful in SQL-based validation queries used by the view object-based Key Exists validator.

When data is not read-only, the best (and only) choice is to create entity-based view objects. Entity-based view objects that are updatable (default behavior) are the only way to pickup entity-derived attribute default values, reflect pending changes made to relevant entity object attributes through other view objects in the same transaction, and reflect updated reference information when foreign key attribute values are changed is to use an entity-based view object.

Using SQL Tracing to Identify Ill-Performing Queries

JDeveloper allows you to tune queries so that the query plan that is used by the database query optimizer is not doing a full table scan and creating performance overheads.

After deciding whether your view object should be mapped to entities or not, your attention should turn to the query itself. On the Query page of the view object overview editor, click the **Test and Explain** button to see the query plan that the database query optimizer will use. If you see that it is doing a full table scan, you should consider adding indexes or providing a value for the **Query Optimizer Hint** field on the **Tuning** section of the overview editor's General page. This will let you explicitly control which query plan will be used. These facilities provide some useful tools to the developer to evaluate the query plans for individual view object SQL statements. However, their use is not a substitute for tracing the SQL of the entire application to identify poorly performing queries in the presence of a production environment's amount of data and number of end users.

You can use the Oracle database's SQL Tracing facilities to produce a complete log of all SQL statements your application performs. The approach that works in all versions of the Oracle database is to issue the command:

```
ALTER SESSION SET SQL_TRACE TRUE
```

Specifically in version 10g of Oracle, the DBA would need to grant `ALTER SESSION` privilege in order to execute this command.

This command enables tracing of the current database session and logs all SQL statements to a server-side trace file until you either enter `ALTER SESSION SET SQL_TRACE FALSE` or close the connection. To simplify enabling this option to trace your Fusion web applications, override the `afterConnect()` method of your application module (or custom application module framework extension class) to conditionally perform the `ALTER SESSION` command to enable SQL tracing based on the presence of a Java system property as shown in the following example.

```
// In YourCustomApplicationModuleImpl.java
protected void afterConnect() {
  super.afterConnect();
  if (System.getProperty("enableTrace") != null) {
    getDBTransaction().executeCommand("ALTER SESSION SET SQL_TRACE TRUE");
  }
}
```

After producing a trace file, you use the `TKPROF` utility supplied with the database to format the information and to better understand information about each query executed like:

- The number of times it was (re)parsed
- The number of times it was executed
- How many round-trips were made between application server and the database
- Various quantitative measurements of query execution time

Using these techniques, you can decide which additional indexes might be required to speed up particular queries your application performs, or which queries could be changed to improve their query optimization plan. For details about working with the `TKPROF` utility, see sections "Understanding SQL Trace and TKPROF" and "Using the SQL Trace Facility and TKPROF" in the "Performing Application Tracing" chapter of the *Oracle Database SQL Tuning Guide*.

Note:

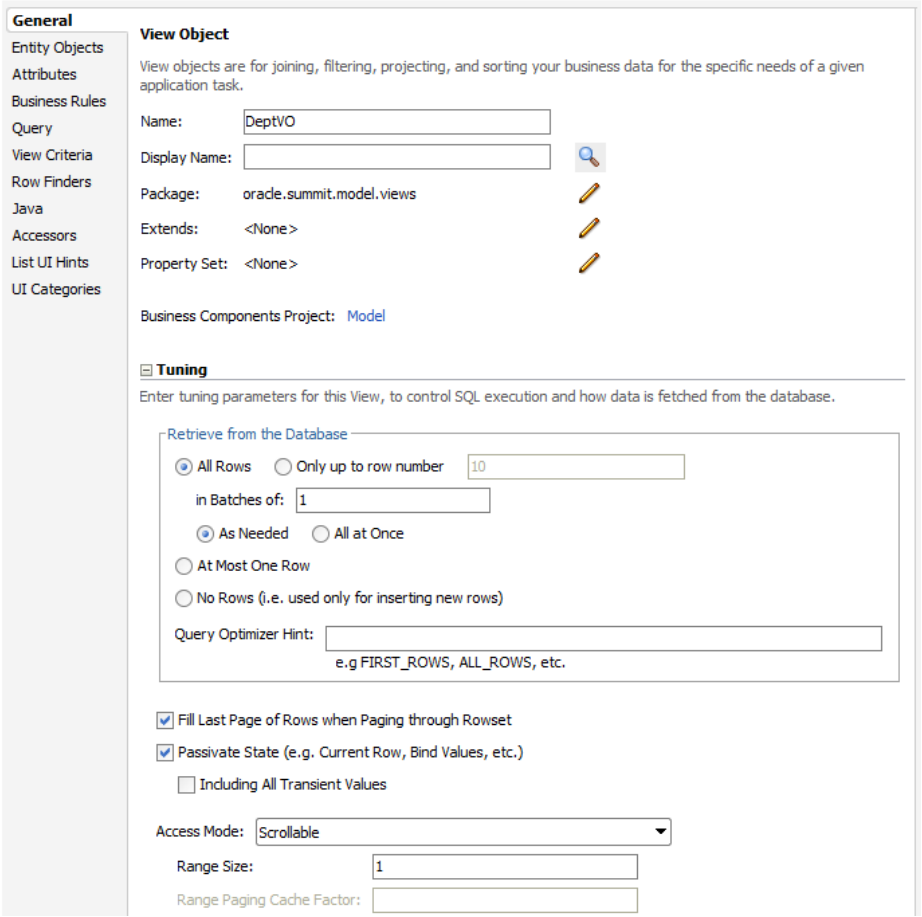
The Oracle database provides the `DBMS_MONITOR` package that further simplifies SQL tracing and integrates it with Oracle Enterprise Manager for visually monitoring the most frequently performed query statements your applications perform.

Using the Appropriate View Object Tuning Settings

The view object overview editor’s Tuning section provides various options to tune the query’s performance.

The Tuning section on the General page of the view object overview editor lets you set various options that can dramatically effect your query's performance. Figure 9-2 shows the default options that the new view object defines.

Figure 9-2 View Object Default Tuning Options



View Object

View objects are for joining, filtering, projecting, and sorting your business data for the specific needs of a given application task.

Name: DeptVO

Display Name:

Package: oracle.summit.model.views

Extends: <None>

Property Set: <None>

Business Components Project: Model

⊟ **Tuning**

Enter tuning parameters for this View, to control SQL execution and how data is fetched from the database.

Retrieve from the Database

● All Rows ○ Only up to row number 10

in Batches of: 1

● As Needed ○ All at Once

○ At Most One Row

○ No Rows (i.e. used only for inserting new rows)

Query Optimizer Hint:

e.g FIRST_ROWS, ALL_ROWS, etc.

☑ Fill Last Page of Rows when Paging through Rowset

☑ Passivate State (e.g. Current Row, Bind Values, etc.)

☐ Including All Transient Values

Access Mode: Scrollable

Range Size: 1

Range Paging Cache Factor:

What You May Need to Know About the Retrieve from Database Options

In the Tuning section of the view object overview editor General page, the **Retrieve from the Database** group box controls how the view object retrieves rows from the database server. The options for the fetch mode are **All Rows**, **Only Up To Row Number, At Most One Row**, and **No Rows**. Most view objects will use the default **All Rows** option, which will be retrieved **As Needed** (default) or **All at Once** depending on which option you choose.

> **Note:**
>
> The **All at Once** option does not enforce a single database round trip to fetch the rows specified by the view object query. The **As Needed** and **All at Once** options work in conjunction with the value of **in Batches of** (also known as fetch size) to determine the number of round trips. For best database access performance, you should consider changing the fetch size as described in Consider Whether Fetching One Row at a Time is Appropriate.

The **As Needed** option ensures that an `executeQuery()` operation on the view object initially retrieves only as many rows as necessary to fill the first page of a display, whose number of rows is set based on the view object's range size. If you use **As Needed**, then you will require only as many database round trips as necessary to deliver the number of rows specified by the initial range size, as defined by the **Range Size** option. Whereas, if you use **All at Once**, then the application will perform as many round trips as necessary to deliver all the rows based on the value of **in Batches of** (fetch size) and the number of rows identified by the query.

For view objects whose `WHERE` clause expects to retrieve a *single* row, set the option to **At Most One Row** for best performance. This way, the view object knows you don't expect any more rows and will skip its normal test for that situation. Finally, if you use the view object only for creating new rows, for optimal performance set the option to **No Rows** so no query will ever be performed.

Before you begin:

It may be helpful to have an understanding of view objects. For more information, see About View Objects.

To set view object row fetch options:

1. In the Applications window, double-click the view object that you want to edit.
2. In the overview editor, click the **General** navigation tab and expand the **Tuning** section.
3. In the **Retrieve from the Database** group box, select the option to control how many rows should be fetched from the database.
4. Enter a **Query Optimizer Hint** hat the database will use to optimize the query for this view object.

   At runtime, the hint is added immediately after the SELECT keyword in the query. For example, you would enter `ALL_ROWS` when you want to retrieve all rows as quickly as possible or `FIRST_ROWS` when you want to retrieve the first rows as quickly as possible (rather than optimize the retrieval of all rows).

Consider Whether Fetching One Row at a Time is Appropriate

The fetch size controls how many rows will be returned in each round trip to the database. By default, the framework will fetch rows in batches of one row at a time. If you are fetching any more than one row, you will gain efficiency by setting this **in Batches of** value on the Tuning section of the General page of the view object overview editor.

Unless your query *actually* fetches just one row, leaving the *default* fetch size of one (1) in the **in Batches of** field is not recommended due to many unnecessary round trips between the application server and the database. Oracle strongly recommends considering the appropriate value for each view object's fetch size.

If you are displaying results *n* rows at a time in the user interface, for simple web pages, it is good to set the fetch size to at least *n*+3 so that each page of results can be retrieved in a single round trip to the database. However the higher the number, the larger the client-side buffer required, so avoid setting this number to values the exceed Oracle JDBC fetch size recommendations.

In general, fetch size should be consistent with Oracle database 12*c* recommendations, which state fetch size should be no more than 100, although in some cases larger size may be appropriate. For more information, see *Oracle JDBC Memory Management* for Oracle Database 12*c* located at `http://www.oracle.com/technetwork/database/application-development/jdbc-memory-management-12c-1964666.pdf`.

Before you begin:

It may be helpful to have an understanding of view objects. For more information, see About View Objects.

To limit the number of database roundtrips for a view object:

1. In the Applications window, double-click the view object that you want to edit.
2. In the overview editor, click the **General** navigation tab and expand the **Tuning** section.
3. In the **Retrieve from the Database** group box, select **All Rows** or **Only up to Row Number**.
4. In the **In Batches of** field, enter the number of rows to return from each database roundtrip.
5. Select the option **All at Once** when you want to fetch all rows whether or not the number of rows fetched fits within the web page displaying the row set. Otherwise, to fetch only the number of rows necessary to fill the first page of the web page displaying the row se, leave the default setting **As Needed** selected.

   If you select **All at Once**, then the application will perform as many roundtrips as necessary to deliver all the rows based on the value of **In Batches of** (fetch size) and the number of rows identified by the query.

Specify a Query Optimizer Hint if Necessary

The **Query Optimizer Hint** field allows you to specify an optional hint to the Oracle query optimizer to influence what execution plan it will use. You can set this hint in the Tuning page of the overview editor for the view object, as shown in Figure 9-2.

At runtime, the hint you provide is added immediately after the `SELECT` keyword in the query, wrapped by the special comment syntax `/*+` *`YOUR_HINT`* `*/`. Two common optimizer hints are:

- `FIRST_ROWS` — to hint that you want the first rows as quickly as possible
- `ALL_ROWS` — to hint that you want all rows as quickly as possible

There are many other optimizer hints that are beyond the scope of this manual to document. Reference the Oracle database reference manuals for more information on available hints.

Before you begin:

It may be helpful to have an understanding of view objects. For more information, see About View Objects.

To edit a view object definition:

1. In the Applications window, double-click the view object that you want to edit.
2. In the overview editor, click the **General** navigation tab and expand the **Tuning** section.

3. In the **Retrieve from the Database** group box, select the option to control how many rows should be fetched from the database.
4. Enter a **Query Optimizer Hint** hat the database will use to optimize the query for this view object.

   At runtime, the hint is added immediately after the SELECT keyword in the query. For example, you would enter `ALL_ROWS` when you want to retrieve all rows as quickly as possible or `FIRST_ROWS` when you want to retrieve the first rows as quickly as possible (rather than optimize the retrieval of all rows).

Using Fetch Size to Limit the Maximum Number of Records Fetched for a View Object

You can set an upper bound on the number of rows a view object retrieves. You can configure a global threshold for all view object queries or a threshold for a specific ADF view object query.

When your application needs to iterate over the rows of the view object query result, you may want to specify a maximum number of records to fetch. The default maximum fetch size of a view object is minus one (-1), which indicates there should be no limit to the number of rows that can be fetched. By default, rows are fetched as needed, so -1 does not imply a view object will necessarily fetch all the rows. It does means that if you attempt to iterate through all the rows in the query result, you will get them all.

To put an upper bound on the maximum number of rows that a view object will retrieve, use the following settings:

- You can configure a global threshold for all view objects queries using the **Row Fetch Limit** property on the Business Components page of the overview editor for the `adf-config.xml` file. You can locate the file in the Application Resources panel by expanding the **Descriptors** and **ADF META-INF** nodes.

  Note: Since **Row Fetch Limit** specifies a global threshold for all query operations in the application (including iterator binding property **RowCountThreshold** used to determine an estimated row count for the iterator result set), using this property means you can avoid changing settings for individual query operations where that operation's default behavior allows all rows to be fetched. If you do specify a fetch limit for individual view objects using the **Max Fetch Size**, then the **Row Fetch Limit** setting will be ignored in those cases. Note that the **Row Fetch Limit** setting is also ignored if the view object implementation class overrides `getRowLimit()` to return an appropriate value for a view object.
- You can configure a threshold for a specific view object query using the **Only up to row number** field selected in the Tuning section of the General page of the overview editor for the view object. Note that the Property Inspector displays this setting in the **Max Fetch Size** property.

> **Tip:**
>
> If you want to set the global threshold for query operations using **Row Fetch Limit** and you still need to allow specific view object queries to return all available rows, then you can set the **Max Fetch Size** with the **Only up to row number** field for those view objects to a very large number.

For example, if you write a query containing an `ORDER BY` clause and only want to return the first *n* rows to display the "Top-N" entries in a page, you can use the overview editor for the view object to specify a value for the **Only up to row number** field in the Tuning section of the General page. For example, to fetch only the first five rows, you would enter "5" in this field. This is equivalent to calling the `setMaxFetchSize()` method on your view object to set the maximum fetch size to `5`. The view object will stop fetching rows when it hits the maximum fetch size. Often you will combine this technique with specifying a **Query Optimizer Hint** of `FIRST_ROWS` also on the Tuning section of the General page of the overview editor. This gives a hint to the database that you want to retrieve the first rows as quickly as possible, rather than trying to optimize the retrieval of all rows.

Using Range Size to Present and Scroll Data a Page at a Time

You can configure an ADF view object to retrieve and manage a predetermined number of data-rows on each page.

To present and scroll through data a page at a time, you can configure a view object to manage for you an appropriately sized range of rows. The range facility allows a client to easily display and update a subset of the rows in a row set, as well as easily scroll to subsequent pages, *n* rows as a time. You call `setRangeSize()` to define how many rows of data should appear on each page. The default range size is one (`1`) row. A range size of minus one (`-1`) indicates the range should include all rows in the row set.

> **Note:**
>
> When using the declarative bindings in the **ADF Model** layer, the **iterator binding** in the **page definition file** has a `RangeSize` attribute. At runtime, the iterator binding invokes the `setRangeSize()` method on its corresponding **row set iterator**, passing the value of this `RangeSize` attribute. The ADF design time by default sets this `RangeSize` attribute to `10` rows for most iterator bindings. An exception is the range size specified for a List binding to supply the set of valid values for a UI component like a dropdown list. In this case, the default range size is minus one (`-1`) to allow the range to include all rows in the row set.

When you set a range size greater than one, you control the row set paging behavior using the iterator mode. The two iterator mode flags you can pass to the `setIterMode()` method are:

- `RowIterator.ITER_MODE_LAST_PAGE_PARTIAL`

  In this mode, the last page of rows may contain fewer rows than the range size. For example, if you set the range size to 10 and your row set contains 23 rows, the third page of rows will contain only three rows. This is the style that works best for Fusion web applications.

- `RowIterator.ITER_MODE_LAST_PAGE_FULL`

  In this mode, the last page of rows is kept full, possibly including rows at the top of the page that had appeared at the bottom of the previous page. For example, if you set the range size to 10 and your row set contains 23 rows, the third page of rows will contain 10 rows, the first seven of which appeared as the last seven rows of page two. This is the style that works best for desktop-fidelity applications using Swing.

Using Range Paging to Efficiently Scroll Through Large Result Sets

The Fusion web application allows you to limit very large query results by pre-determining how many ADF Business Components rows the query fetches. If it is very large, the end-user can be prompted to provide more filter criteria.

As a general rule, for highest performance, Oracle recommends building your application in a way that avoids giving the end user the opportunity to scroll through very large query results. To enforce this recommendation, call the `getEstimatedRowCount()` method on a view object to determine how many rows would be returned by the user's query *before* actually executing the query and allowing the user to proceed. If the estimated row count is unreasonably large, your application can demand that the end-user provide additional search criteria.

However, when you *must* work with very large result sets, typically over 100 rows, you can use the view object's access mode called "range paging" to improve performance. The feature allows your applications to page back and forth through data, a range of rows at a time, in a way that is more efficient for large data sets than the default "scrollable" access mode.

The range paging access mode is typically used for paging through read-only row sets, and often is used with read-only view objects. You allow the user to find the row they are looking for by paging through a large row set with range paging access mode, then you use the `Key` of that row to find the selected row in a different view object for editing.

Range paging for view objects supports a standard access mode and a variation of the standard access mode that combines the benefits of range paging and result set scrolling with a minimum number of visits to the database. These modes for the view object range paging feature include:

- `RANGE_PAGING`, standard access mode fetches the number of rows specified by a range size. In this mode, the number of rows that may be scrolled without requerying the database is determined by a range size that you set. The default is to fetch a single row, but it is expected that you will set a range size equal to the number of rows you want to be able to display to the user before they scroll to the next result set. The application requeries the database each time a row outside of the range is accessed by the end user. Thus, scrolling backward and forward through the row set will requery the database. For clarification about this database-centric paging strategy, see Understanding How Oracle Supports "TOP-N" Queries.
- `RANGE_PAGING_INCR`, incremental access mode gives the UI designer more flexibility for the number of rows to display at a time while keeping database queries to a minimum. In this mode, the UI incrementally displays the result set from the memory cache and thus supports scrolling within a single database query. The number of rows that the end user can scroll though in a single query is determined by the range size and a range paging cache factor that you set. For example, suppose that you set the range size to 4 and the cache factor to

5. Then, the maximum number of rows to cache in memory will be 4*5 = 20. For further explanation of the caching behavior, see What Happens When View Rows are Cached When Using Range Paging.

> **Note:**
>
> Additionally, the view object supports a `RANGE_PAGING_AUTO_POST` access mode to accommodate the inserting and deleting of rows from the row set. This mode behaves like the `RANGE_PAGING` mode, except that it eagerly calls `postChanges()` on the database transaction whenever any changes are made to the row set. However, this mode is typically not appropriate for use in Fusion web applications unless you can guarantee that the transaction will definitely be committed or rolled-back during the same HTTP request. Failure to heed this advice can lead to strange results in an environment where both application modules and database connections can be pooled and shared serially by multiple different clients.

Understanding How Oracle Supports "TOP-N" Queries

The Oracle database supports a feature called a "Top-N" query to efficiently return the first *n* ordered rows in a query. For example, if you have a query like:

```
SELECT EMPNO, ENAME,SAL FROM EMP ORDER BY SAL DESC
```

If you want to retrieve the top 5 employees by salary, you can write a query like:

```
SELECT * FROM (
   SELECT X.*,ROWNUM AS RN FROM (
      SELECT EMPNO,ENAME,SAL FROM EMP ORDER BY SAL DESC
   ) X
) WHERE RN <= 5
```

which gives you results like:

```
     EMPNO ENAME       SAL    RN
---------- -------- ------ ----
      7839 KING       5000     1
      7788 SCOTT      3000     2
      7902 FORD       3000     3
      7566 JONES      2975     4
      7698 BLAKE      2850     5
```

The feature is not only limited to retrieving the first *n* rows in order. By adjusting the criteria in the outermost `WHERE` clause you can efficiently retrieve any range of rows in the query's sorted order. For example, to retrieve rows `6` through `10` you could alter the query this way:

```
SELECT * FROM (
  SELECT X.*,ROWNUM AS RN FROM  (
     SELECT EMPNO,ENAME,SAL FROM EMP ORDER BY SAL DESC
  ) X
) WHERE RN BETWEEN 6 AND 10
```

Generalizing this idea, if you want to see page number *P* of the query results, where each page contains *R* rows, then you would write a query like:

```
SELECT * FROM (
  SELECT X.*,ROWNUM AS RN FROM (
    SELECT EMPNO,ENAME,SAL FROM EMP ORDER BY SAL DESC
  ) X
) WHERE RN BETWEEN ((:P - 1) * :R) + 1 AND (:P) * :R
```

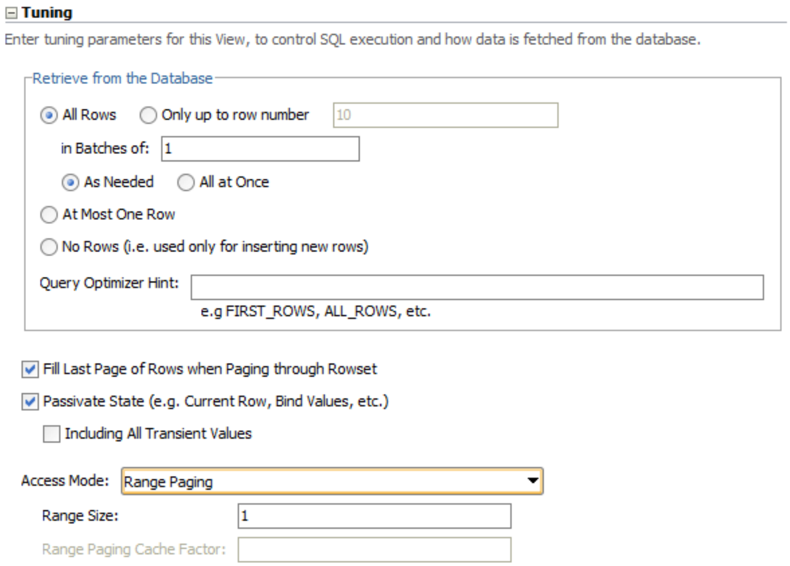
As the result set you consider grows larger and larger, it becomes more and more efficient to use this technique to page through the rows. Rather than retrieving hundreds or thousands of rows over the network from the database, only to display ten of them on the page, instead you can produce a clever query to retrieve only the `R` rows on page number `P` from the database. No more than a handful of rows at a time needs to be returned over the network when you adopt this strategy.

To implement this database-centric paging strategy in your application, you could handcraft the clever query yourself and write code to manage the appropriate values of the `:R` and `:P` bind variables. Alternatively, you can use the view object's range paging access mode, which implements it automatically for you.

How to Enable Range Paging for a View Object

You can use the Tuning panel of the overview editor for the view object to set the access mode to either standard range paging or incremental range paging. The **Range Paging Cache Factor** field is only editable when you select **Range Paging Incremental**. Figure 9-3 shows the view object's **Access Mode** set to **Range Paging** (standard mode) with the default range size of 1. To understand the row set caching behavior of both access modes, see What Happens When View Rows are Cached When Using Range Paging.

Figure 9-3 Access Mode in the Overview Editor for the View Object



⊟ Tuning

Enter tuning parameters for this View, to control SQL execution and how data is fetched from the database.

Retrieve from the Database

● All Rows ○ Only up to row number 10

in Batches of: 1

● As Needed ○ All at Once

○ At Most One Row

○ No Rows (i.e. used only for inserting new rows)

Query Optimizer Hint:

e.g FIRST_ROWS, ALL_ROWS, etc.

☑ Fill Last Page of Rows when Paging through Rowset

☑ Passivate State (e.g. Current Row, Bind Values, etc.)

□ Including All Transient Values

Access Mode: Range Paging

Range Size: 1

Range Paging Cache Factor:

To programmatically enable standard range paging for your view object, first call `setRangeSize()` to define the number of rows per page, then call the following method with the desired mode:

```
yourViewObject.setAccessMode(RowSet.RANGE_PAGING | RANGE_PAGING_INCR);
```

If you set RANGE_PAGING_INCR, then you must also call the following method to set the cache factor for your defined range size:

```
yourViewObject.setRangePagingCacheFactor(int f);
```

What Happens When You Enable Range Paging

When a view object's access mode is set to RANGE_PAGING, the view object takes its default query like:

```
SELECT EMPNO, ENAME, SAL FROM EMP ORDER BY SAL DESC
```

and automatically "wraps" it to produce a Top-N query.

For best performance, the statement uses a combination of greater than and less than conditions instead of the BETWEEN operator, but the logical outcome is the same as the Top-N wrapping query you saw above. The actual query produced to wrap a base query of:

```
SELECT EMPNO, ENAME, SAL FROM EMP ORDER BY SAL DESC
```

looks like this:

```
SELECT * FROM (
  SELECT /*+ FIRST_ROWS */ IQ.*, ROWNUM AS Z_R_N FROM (
    SELECT EMPNO, ENAME, SAL FROM EMP ORDER BY SAL DESC
  ) IQ  WHERE ROWNUM < :0)
WHERE Z_R_N > :1
```

The two bind variables are bound as follows:

- :1 index of the first row in the current page
- :0 is bound to the last row in the current page

What Happens When View Rows are Cached When Using Range Paging

When a view object operates in RANGE_PAGING access mode, it only keeps the current range (or "page") of rows in memory in the view row cache at a time. That is, if you are paging through results ten at a time, then on the first page, you'll have rows 1 through 10 in the view row cache. When you navigate to page two, you'll have rows 11 through 20 in the cache. This also can help make sure for large row sets that you don't end up with tons of rows cached just because you want to preserve the ability to scroll backwards and forwards.

When a view object operates in RANGE_PAGING_INCR access mode, the cache factor determines the number of rows to cache in memory for a specific range size. For example, suppose the range size is set to 4 and cache factor to 5. Then, the memory will keep at most 4*5 = 20 rows in its collection. In this example, when the range is refreshed for the first time, the memory will have just four rows even though the range paging query is bound to retrieve rows 0 to 19 (for a total of twenty rows). When the range is scrolled past the forth row, more rows will be read in from the current result set. This will continue until all twenty rows from the query result are read. If the user's action causes the next set of rows to be retrieve, the query will be reexecuted with the new row number bind values. The exact row number bind values are determined by the new range-start and the number of rows that can be retained from the cache. For example, suppose all twenty rows have been filled up and the user asks to move the range-start to 18 (0-based). This means that memory can retain row 18 and row 19 and will need two more rows to fill the range. The query is reexecuted for rows 20 and 21.

How to Scroll to a Given Page Number Using Range Paging

When a view object operates in `RANGE_PAGING` access mode, to scroll to page number *n* call its `scrollToRangePage()` method, passing *n* as the parameter value.

How to Estimate the Number of Pages in the Row Set Using Range Paging

When a view object operates in `RANGE_PAGING` access mode, you can access an estimate of the total number of pages the entire query result would produce using the `getEstimatedRangePageCount()` method.

Understanding the Tradeoffs of Using a Range Paging Mode

You might ask yourself, "Why wouldn't I *always* want to use `RANGE_PAGING` or `RANGE_PAGING_INCR` mode?" The answer is that using range paging potentially causes more overall queries to be executed as you are navigating forward and backward through your view object rows. You would want to avoid using `RANGE_PAGING` mode in these situations:

- You plan to read all the rows in the row set immediately (for example, to populate a dropdown list).

  In this case your range size would be set to `-1` and there really is only a single "page" of all rows, so range paging does not add value.

- You need to page back and forth through a small-sized row set.

  If you have 100 rows or fewer, and are paging through them 10 at a time, with `RANGE_PAGING` mode you will execute a query each time you go forward and backward to a new page. Otherwise, in the default scrollable mode, you will cache the view object rows as you read them in, and paging backwards through the previous pages will not reexecute queries to show those already-seen rows. Alternatively, you can use `RANGE_PAGING_INCR` mode to allow scrolling through in-memory results based on a row set cache factor that you determine.

In the case of a very large (or unpredictably large) row set, the trade off of potentially doing a few more queries — each of which only returns up to the `RangeSize` number of rows from the database — is more efficient then trying to cache all of the previously viewed rows. This is especially true if you allow the user to jump to an arbitrary page in the list of results. Doing so in default, scrollable mode requires fetching and caching all of the rows between the current page and the page the users jumps to. In `RANGE_PAGING` mode, it will ask the database just for the rows on that page. Then, if the user jumps back to a page of rows that they have already visited, in `RANGE_PAGING` mode, those rows get requeried again since only the current page of rows is held in memory in this mode. The incremental range paging access mode `RANGE_PAGING_INCR` combines aspects of both standard range paging and scrollable access mode since it allows the application to cache more rows in memory and permits the user to jump to any combination of those rows without needing to requery.

Using Forward Only Mode to Avoid Caching View Rows

You can set the `forward only` mode on the ADF view object if each row in the row set needs to be read only once and does not require reiteration. This technique avoids caching of retrieved rows.

Often you will write code that programmatically iterates through the results of a view object. A typical situation will be custom validation code that must process multiple rows of query results to determine whether an attribute or an entity is valid or not. In these cases, if you intend to read each row in the row set a single time and never require scrolling backward or reiterating the row set a subsequent time, then you can use "forward only" mode to avoid caching the retrieved rows. To enable forward only mode, call `setForwardOnly(true)` on the view object.

> **Note:**
>
> Using a read-only view object (with no entity usages) in forward-only mode with an appropriately tuned fetch size is the most efficient way to programmatically read data.

You can also use forward-only mode to avoid caching rows when inserting, updating, or deleting data as long as you never scroll backward through the row set and never call `reset()` to set the iterator back to the first row. Forward only mode only works with a range size of one (`1`).

Using Retain Row Set to Optimize View Link Accessor Access

You can force an ADF view link accessor row set to refresh its rows from the database. This is because each time you access a view link accessor row set, you access an object of this rowset without a database roundtrip.

Each time you retrieve a view link accessor row set, by default the view object creates a new `RowSet` object to allow you to work with the rows. This does *not* imply *re-executing* the query to produce the results each time, only creating a new instance of a `RowSet` object with its default iterator reset to the "slot" before the first row. To force the row set to refresh its rows from the database, you can call its `executeQuery()` method.

You can enable caching of the view link accessor row set when you do not want the application to incur the small amount of overhead associated with creating new detail row sets. For example, because **view accessor** row sets remain stable as long as the master row view accessor attribute remains unchanged, it would not be necessary to re-create a new row set for UI components, like the tree control, where data for each master node in a tree needs to retain its distinct set of detail rows. The view link accessor's detail row set can also be accessed programmatically. In this case, if your application makes numerous calls to the same view link accessor attributes, you can consider caching the view link accessor row set. This style of managing master-detail coordination differs from creating view link instances in the data model, as explained in What You May Need to Know About View Link Accessors Versus Data Model View Link Instances.

You can enable retention of the view link accessor row set using the overview editor for the view object that is the source for the view link accessor. Select **Retain Row Set** in the View Accessors page of the overview editor for the view object.

Alternatively, you can enable a custom Java class for your view object, override the `create()` method, and add a line after `super.create()` that calls the `setViewLinkAccessorRetained()` method passing `true` as the parameter. It affects all view link accessor attributes for that view object.

When this feature is enabled for a view object, since the view link accessor row set is not re-created each time, the current row of its default row set iterator is also retained as a side-effect. This means that your code will need to explicitly call the `reset()` method on the row set you retrieve from the view link accessor to reset the current row in its default row set iterator back to the "slot" before the first row.

Note, however, that with accessor retention enabled, your failure to call `reset()` each time before you iterate through the rows in the accessor row set can result in a subtle, hard-to-detect error in your application. For example, if you iterate over the rows in a view link accessor row set like this, for example to calculate some aggregate total:

```
RowSet rs = (RowSet)row.getAttribute("OrdersShippedToPurchaser");
while (rs.hasNext()) {
  Row r = rs.next();
  // Do something important with attributes in each row
}
```

The first time you work with the accessor row set the code will work. However, since the row set (and its default row set iterator) are retained, the second and subsequent times you access the row set the current row will already be at the end of the row set and the while loop will be skipped since `rs.hasNext()` will be `false`. Instead, with this feature enabled, write your accessor iteration code like this:

```
RowSet rs = (RowSet)row.getAttribute("OrdersShippedToPurchaser");
rs.reset(); // Reset default row set iterator to slot before first row!
while (rs.hasNext()) {
  Row r = rs.next();
  // Do something important with attributes in each row
}
```

Recall that if view link consistency is on, when the accessor is retained the new unposted rows will show up at the end of the row set. This is slightly different from when the accessor is not retained (the default), where new unposted rows will appear at the beginning of the accessor row set.

Using Dynamic Attributes On View Objects to Store UI State

You can add dynamic attributes which hold serializable objects as values to an ADF view object.

You can add one or more dynamic attributes to a view object at runtime using the `addDynamicAttribute()` method. Dynamic attributes can hold any serializable object as their value. Typically, you will consider using dynamic attributes when writing generic framework extension code that requires storing some additional per-row transient state to implement a feature you want to add to the framework in a global, generic way.

10

Defining Validation and Business Rules Declaratively

This chapter describes how to use ADF entity objects to write business rules that implement declarative validation in an Oracle ADF application.
This chapter includes the following sections:

About Declarative Validation

Use the overview editor of the ADF entity object to define declarative validation rules. These rules are stored in the entity object XML file. In addition to built-in declarative validation, you can write custom rules.

The easiest way to create and manage validation rules is through **declarative validation rules**. Declarative validation rules are defined using the overview editor, and once created, are stored in the **entity object** XML file. Declarative validation is different from programmatic validation (covered in Implementing Validation and Business Rules Programmatically), which is stored in an entity object's Java file.

Oracle ADF provides built-in declarative validation rules that satisfy many of your business needs. If you have custom validation rules you want to reuse, you can code them and add them to the IDE, so that the rules are available directly from JDeveloper. Custom validation rules are an advanced topic and covered in Implementing Custom Validation Rules. You can also base validation on a Groovy expression, as described in Using Groovy Expressions For Business Rules and Triggers.

When you add a validation rule, you supply an appropriate error message and can later translate it easily into other languages if needed. You can also define how validation is triggered and set the severity level.

Declarative Validation Use Cases and Examples

In a Fusion web application containing **ADF Business Components**, most of your validation code is defined in your entity objects. Encapsulating the business logic in these shared, reusable components ensures that your business information is validated consistently in every **view object** or client that accesses it, and it simplifies maintenance by centralizing where the validation is stored.

Another benefit of using declarative validation (versus writing your own validation) is that the validation framework takes care of the complexities of batching validation exceptions, which frees you to concentrate on your application's specific validation rule logic.

In the model layer, ADF Model validation rules can be set for the attributes of a collection. Many of the declarative validation features available for entity objects are also available at the model layer, should your application warrant the use of model-layer validation in addition to business-layer validation. For more information, see Using Validation in the ADF Model Layer .

When you use the ADF Business Components **application module data control**, you do not need to use model-layer validation. Consider defining all or most of your validation rules in the centralized, reusable, and easier to maintain entity objects of your business layer. With other types of **data controls**, **ADF Model** layer validation can be more useful.

Additional Functionality for Declarative Validation

You may find it helpful to understand other Oracle ADF features before you start using declarative validation. It is possible to go beyond the declarative behavior to implement more complex validation rules for your business domain layer when needed. Following are links to other functionality that may be of interest.

- Using Method Validators explains how to use the Method validator to invoke custom validation code.
- Implementing Custom Validation Rules details how to extend the basic set of declarative rules with custom rules of your own.

Determining Where to Implement Validation

Oracle ADF follows the Model View Controller (MVC) architectural design pattern that allows validation implementation at each layer. Depending on business needs, implement validations at particular layers.

As explained in Oracle ADF Architecture, the model-view-controller (MVC) architecture of a Fusion web application has multiple layers that provide a clean separation of business logic, page navigation, and user interface. These layers also provide multiple options for implementing validation, and there are benefits and disadvantages to using validation at each layer. The place where you choose to implement validation is dependent on the given business need or use case. Generally, the validations that occur closer to the data source are more centralized and therefore easier to maintain, while those that occur closer to the user interface provide more immediate feedback at runtime.

Data services layer -- Whether it be a database or a web service or some other source of data, the data services layer typically resides outside the context of the application itself and employs constraints that define the nature of each field of data. Data services constraints are typically defined for every piece of data that is stored, and are very important to maintaining the integrity of that data. However, it is neither effective nor efficient to rely solely on these constraints, because they are not enforced until the data reaches the data source, typically when the application attempts to commit a transaction. This requires a round trip from the client UI to the data source and back, and a single error on a single record can cause the rollback of entire transaction.

Business services layer -- Business rules that you implement at the business service layer in ADF Business Components (such as entity objects and view objects) provide more efficiency than data services constraints because they don't require a round trip to the data source. They are also easier to maintain than validators implemented at the view layer because a business rule defined on an attribute for a given entity object is enforced on every page that uses that entity object. In addition to attribute-level validators, you can also implement entity-level validators when the validity of the value of one attribute is dependent on the value of another. For example, `DateShipped` must be greater than `DateOrdered`. This kind of business rule is explained in this chapter.

Model layer -- Although not used as commonly as the other kinds of business rules, model layer validation can also be useful. You can define business rules on a binding to a business service. This feature can be useful, for example, if you can't implement validation on the business service itself because your application consumes an external web service. See Using Validation in the ADF Model Layer .

View layer -- From the user's perspective, view layer validations are the most efficient. They provide immediate feedback when the user enters data and subsequently leaves (tabs out of) a given field, without requiring even a trip to the application server. However, view layer validations are implemented on the page, so every page that provides access to a given data field must define the validation for that field. See the "Validating and Converting Input" chapter in *Developing Web User Interfaces with Oracle ADF Faces*.

Understanding the Validation Cycle

You can enforce validation on an ADF entity row, either when a persistent entity row attribute is modified, or an entity row is created.

Each entity row tracks whether or not its data is valid. When an existing entity row is retrieved from the database, the entity is assumed to be valid. When the first persistent attribute of an existing entity row is modified, or when a new entity row is created, the entity is marked invalid.

When an entity is in an invalid state, the declarative validation you have configured and the programmatic validation rules you have implemented are evaluated again before the entity can be considered valid again. You can determine whether a given entity row is valid at runtime by calling the `isValid()` method on it.

> **Note:**
>
> Because attributes can (by default) be left blank, validations are not triggered if the attribute contains no value. For example, if a user creates a new entity row and does not enter a value for a given attribute, the validation on that attribute is not run. To force the validation to execute in this situation, set the Mandatory flag on the attribute.

Types of Entity Object Validation Rules

Entity object validation rules fall into two basic categories: *attribute-level* and *entity-level*.

Attribute-Level Validation Rules

Attribute-level validation rules are triggered for a particular entity object attribute when either the end user or the program code attempts to modify the attribute's value. Since you cannot determine the order in which attributes will be set, attribute-level validation rules should be used only when the success or failure of the rule depends exclusively on the candidate value of that single attribute.

The following represent attribute-level validations:

- The value of the `OrderDate` of an order should not be a date in the past.
- The `ProductId` attribute of a product should represent an existing product.

Entity-Level Validation Rules

All other kinds of validation rules are entity-level validation rules. These are rules whose implementation requires considering two or more entity attributes, or possibly composed children entity rows, in order to determine the success or failure of the rule.

The following represent entity-level validations:

- The value of the `OrderShippedDate` should be a date that comes after the `OrderDate`.
- The `ProductId` attribute of an order should represent an existing product.

Entity-level validation rules are triggered by calling the `validate()` method on a `Row`. This occurs when:

- You call the method explicitly on the entity object
- You call the method explicitly on a view row with an entity row part that is invalid
- A view object's iterator calls the method on the current row in the view object before allowing the current row to change
- During transaction commit, processing validates an invalid entity (in the list of pending changes) before proceeding with posting the changes to the database

As part of transaction commit processing, entity-level validation rules can fire multiple times (up to a specified limit). For more information, see Avoiding Infinite Validation Cycles.

Understanding Commit Processing and Validation

Transaction commit processing happens in three basic phases:

1. Ensure that any invalid entity rows on the pending changes list are valid.
2. Post the pending changes to the database by performing appropriate DML operations.
3. Commit the transaction.

If you have business validation logic in your entity objects that executes queries or stored procedures that depend on seeing the posted changes in the `SELECT` statements they execute, they should be coded in the `beforeCommit()` method described in What You May Need to Know About Row Set Access with View Accessors. This method fires after all DML statements have been applied so queries or stored procedures invoked from that method can "see" all of the pending changes that have been saved, but not yet committed.

> **Caution:**
>
> Don't use the transaction-level `postChanges()` method in web applications unless you can guarantee that the transaction will definitely be committed or rolled-back during the same HTTP request. This method exists to force the transaction to post unvalidated changes without committing them. Failure to heed this advice can lead to strange results in an environment where both application modules and database connections can be pooled and shared serially by multiple different clients.

Understanding the Impact of Composition on Validation Order

Because a composed child entity row is considered an integral part of its composing parent entity object, any change to composed child entity rows causes the parent entity to be marked invalid. For example, if a line item on an order were to change, the entire order would now be considered to be changed, or invalid.

Therefore, when the composing entity is validated, it causes any currently invalid composed children entities to be validated first. This behavior is recursive, drilling into deeper levels of invalid composed children if they exist.

Avoiding Infinite Validation Cycles

If your validation rules contain code that updates attributes of the current entity or other entities, then the act of validating the entity can cause that or other entities to become invalid. As part of the transaction commit processing phase that attempts to validate all invalid entities in the pending changes list, the transaction performs multiple passes (up to a specified limit) on the pending changes list in an attempt to reach a state where all pending entity rows are valid.

The maximum number of validation passes is specified by the transaction-level validation threshold setting. The default value of this setting is 10. You can increase the threshold count if the entities involved contain the appropriate logic to validate themselves in the subsequent passes.

If after 10 passes, there are still invalid entities in the list, you will see the following exception:

```
JBO-28200: Validation threshold limit reached. Invalid Entities still in cache
```

This is a sign that you need to debug your validation rule code to avoid inadvertently invalidating entities in a cyclic fashion.

To change the validation threshold:

1. In the Applications window, double-click the desired application module.
2. In the overview editor, click the **Configurations** navigation tab, select the configuration hyperlink for which you want to adjust the validation threshold.
3. In the overview editor for application module configurations (on the `bc4j.xcfg` file), click the **Properties** tab, and click **Add Property**.
4. In the Add Property dialog, select the `jbo.validation.threshold` property and click **OK**.
5. In the overview editor, in the Properties list, enter a new value for the `jbo.validation.threshold` property and save your changes.

You can also change the validation threshold programmatically by using the `SetValidationThreshold()` method as shown below. In this example, the new threshold is 12.

```
oracle.jbo.server.DBTransaction::setValidationThreshold(12)
```

What Happens When Validations Fail

When an entity object's validation rules throw exceptions, the exceptions are bundled and returned to the client. If the validation failures are thrown by methods you've overridden to handle events during the transaction `postChanges` processing, then the validation failures cause the transaction to roll back any database `INSERT`, `UPDATE`, or `DELETE` statements that might have been performed already during the current `postChanges` cycle.

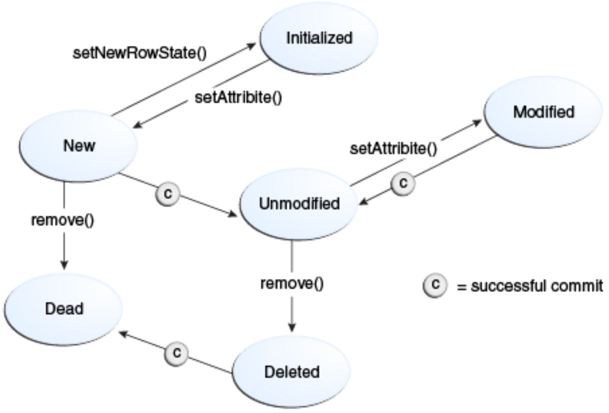
> **Note:**
>
> The bundling of exceptions is the default behavior for ADF Model-based web applications, but not for Oracle ADF Model Tester or Swing bindings. Additional configuration is required to bundle exceptions for the Oracle ADF Model Tester or Swing clients.

Understanding Entity Objects Row States

When an entity row is in memory, it has an entity state that reflects the logical state of the row. Figure 10-1 illustrates the different entity row states and how an entity row can transition from one state to another. When an entity row is first created, its status is `New`. You can use the `setNewRowState()` method to mark the entity as being `Initialized`, which removes it from the transaction's list of pending changes until the user sets at least one of its attributes, at which time it returns to the New state. This allows you to create more than one initialized row and post only those that the user modifies.

The Unmodified state reflects an entity that has been retrieved from the database and has not yet been modified. It is also the state that a New or Modified entity transitions to after the transaction successfully commits. During the transaction in which it is pending to be deleted, an Unmodified entity row transitions to the Deleted state. Finally, if a row that was New and then was removed before the transaction commits, or Unmodified and then successfully deleted, the row transitions to the Dead state.

Figure 10-1 Diagram of Entity Row States and Transitions



You can use the getEntityState() and getPostState() methods to access the current state of an entity row in your business logic code. The getEntityState() method returns the current state of an entity row with regard to the transaction, while the getPostState() method returns the current state of an entity row with regard to the database after using the postChanges() method to post pending changes without committing the transaction.

For example, if you start with a new row, both getEntityState() and getPostState() return STATUS_NEW. Then when you post the row (before commit or rollback), the row will have an entity state of STATUS_NEW and a post state of STATUS_UNMODIFIED. If you subsequently remove that row, the entity state will remain STATUS_NEW because for the transaction the row is still new. But the post state will be STATUS_DEAD.

Understanding Bundled Exception Mode

An **application module** provides a feature called bundled exception mode which allows web applications to easily present a maximal set of failed validation exceptions to the end user, instead of presenting only the *first* error that gets raised. By default, the ADF Business Components **application module pool** enables bundled exception mode for web applications.

You typically will not need to change this default setting. However it is important to understand that it is enabled by default since it effects how validation exceptions are thrown. Since the Java language and runtime only support throwing a single exception object, the way that bundled validation exceptions are implemented is by wrapping a set of exceptions as details of a new "parent" exception that contains them. For example, if multiple attributes in a single entity object fail attribute-level validation, then these multiple `ValidationException` objects will be wrapped in a `RowValException`. This wrapping exception contains the row key of the row that has failed validation. At transaction commit time, if multiple rows do not successfully pass the validation performed during commit, then all of the `RowValException` objects will get wrapped in an enclosing `TxnValException` object.

When writing custom error processing code, you can use the `getDetails()` method of the `JboException` base exception class to recursively process the bundled exceptions contained inside it.

> **Note:**
>
> All the exception classes mentioned here are in the `oracle.jbo` package.

Adding Validation Rules to Entity Objects and Attributes

ADF Business Components allows you to declaratively add validations rules to an entity object.

The process for adding a validation rule to an entity object is similar for most of the validation rules, and is done using the Add Validation Rule dialog. You can open this dialog from the overview editor by clicking the **Add** icon on the Business Rules page.

It is important to note that when you define a rule declaratively using the Add Validation Rule dialog, the rule definition you provide specifies the *valid* condition for the attribute or entity object. At runtime, the entry provided by the user is evaluated against the rule definition and an error or warning is raised if the entry fails to satisfy the specified criteria. For example, if you specify a Length validator on an attribute that requires it to be `Less Than or Equal To 12`, the validation fails if the entry is more than 12 characters, and the error or warning is raised.

How to Add a Validation Rule to an Entity or Attribute

To add a declarative validation rule to an entity object, use the Business Rules page of the overview editor.

Before you begin:

It may be helpful to have an understanding of the use of validation rules in entity objects and attributes. For more information, see Adding Validation Rules to Entity Objects and Attributes.

You may also find it helpful to understand additional functionality that can be added using other validation features. For more information, see Additional Functionality for Declarative Validation.

To add a validation rule:

1. In the Applications window, double-click the desired entity object.
2. In the overview editor, click the **Business Rules** navigation tab, select the object for which you want to add a validation rule, and then click the **Add** icon.
   - To add a validation rule at the entity object level, select **Entity**.

- To add a validation rule for an attribute, expand **Attributes** and select the desired attribute.

When you add a new validation rule, the Add Validation Rule dialog appears.

3. In the Add Validation Rule dialog, select the desired type of validation rule from the **Rule Type** dropdown list.
4. Use the dialog settings to configure the new rule.

   The controls will change depending on the kind of validation rule you select.

   For more information about the different validation rules, see Using the Built-in Declarative Validation Rules.
5. Optionally, click the **Validation Execution** tab to enter criteria for the execution of the rule, such as dependent attributes and a precondition expression.

   For more information, see Triggering Validation Execution.

   > **Note:**
   >
   > For Key Exists and Method entity validators, you can also use the **Validation Execution** tab to specify the validation level.
6. Click the **Failure Handling** tab and enter or select the error message that will be shown to the user if the validation rule fails.

   For more information, see Creating Validation Error Messages.
7. Click **OK**.

How to View and Edit a Validation Rule on an Entity Object or Attribute

The Business Rules page of the overview editor for entity objects displays the validation rules for an entity and its attributes in a tree control. To see the validation rules that apply to the entity as a whole, expand in the **Entity** node. To see the validation rules that apply to an attribute, expand the **Attributes** node and then expand the attribute.

The validation rules that are shown on the Business Rules page of the overview editor include those that you have defined as well as database constraints, such as mandatory or precision. To open a validation rule for editing, double-click the rule or select the rule and click the **Edit** icon.

What Happens When You Add a Validation Rule

When you add a validation rule to an entity object, JDeveloper updates the entity object's XML document to include an entry describing what rule you've used and what rule properties you've entered. For example, if you add a range validation rule to the `DiscountAmount` attribute, this results in a `RangeValidationBean` entry in the XML file, as shown in the following example.

```
<Attribute
  Name="DiscountAmount"
  IsNotNull="true"
  ColumnName="DISCOUNT_AMOUNT"
  . . .
```

```
    <validation:RangeValidationBean
      xmlns="http://xmlns.oracle.com/adfm/validation"
      Name="DiscountAmount_Rule_0"
      ResId="DiscountAmount_RangeError_0"
      OnAttribute="DiscountAmount"
      OperandType="LITERAL"
      Inverse="false"
      MinValue="0"
      MaxValue="40"/>
    . . .
  </Attribute>
```

At runtime, the rule is enforced by the entity object based on this declarative information.

What You May Need to Know About Entity and Attribute Validation Rules

Declarative validation enforces both entity-level and attribute-level validation, depending on where you place the rules. Entity-level validation rules are enforced when a user tries to commit pending changes or navigates between rows. Attribute-level validation rules are enforced when the user changes the value of the related attribute.

The Unique Key validator (described in How to Ensure That Key Values Are Unique) can be used only at the entity level. Internally the Unique Key validator behaves like an attribute-level validator. This means that users see the validation error when they tab out of the key attribute for the key that the validator is validating. This is done because the internal cache of entities can never contain a duplicate, so it is not allowed for an attribute value to be set that would violate that. This check needs to be performed when the attribute value is being set because the cache consistency check is done during the setting of the attribute value.

> **Best Practice:**
>
> If the validity of one attribute is dependent on one or more other attributes, enforce this rule using entity validation, not attribute validation. Examples of when you would want to do this include the following:
>
> - You have a Compare validator that compares one attribute to another.
> - You have an attribute with an expression validator that examines the value in another attribute to control branching in the expression to validate the attribute differently depending on the value in this other attribute.
> - You make use of conditional execution, and your precondition expression involves an attribute other than the one that you are validating.

Entity object validators are triggered whenever the entity, as a whole, is dirty. To improve performance, you can indicate which attributes play a role in your rule and thus the rule should be triggered only if one or more of these attributes are dirty. For more information on triggering attributes, see, Triggering Validation Execution.

What You May Need to Know About List of Values and Attribute Validation Rules

Developers may define a **List of Values (LOV)** attribute on a view object to support displaying choice lists in the user interface. The view object's LOV attribute in turn relies on a data source view object to provide the values for display. Because the LOV feature assumes that the queried data source contains only valid values, any validation rules defined on data source view object attributes will be suppressed before the choice list displays in the user interface. Therefore, the developer who defines the LOV must ensure that the list of values returned by the data source view object contains only valid values, as describe in Working with List of Values (LOV) in View Object Attributes.

Using the Built-in Declarative Validation Rules

ADF Business Components provides numerous built-in declarative validation rules to achieve numerous validation use cases.

The built-in declarative validation rules can satisfy many, if not all, of your business needs. These rules are easy to implement because you don't write any code. You use the user-interface tools to choose the type of validation and how it is used.

Built-in declarative validation rules can be used to:

- Ensure that key values are unique (primary key or other unique keys)
- Determine the existence of a key value
- Make a comparison between an attribute and anything from a literal value to a SQL query
- Validate against a list of values that might be a literal list, a SQL query, or a view attribute
- Make sure that a value falls within a certain range, or that it is limited by a certain number of bytes or characters
- Validate using a regular expression or evaluate a Groovy expression
- Make sure that a value satisfies a relationship defined by an aggregate on a child entity available through an accessor
- Validate using a validation condition defined in a Java method on the entity

How to Ensure That Key Values Are Unique

The Unique Key validator ensures that primary key values for an entity object are always unique. The Unique Key validator can also be used for a non-primary-key attribute, as long as the attribute is defined as an alternate key. For information on how to define alternate keys, see How to Define Alternate Key Values.

Whenever any of the key attribute values change, this rule validates that the new key does not belong to any other entity object instance of this entity object class. (It is the business-logic tier equivalent of a unique constraint in the database.) If the key is found in one of the entity objects, a `TooManyObjectsException` is thrown. The validation check is done both in the entity cache and in the database.

There is a slight possibility that unique key validation might not be sufficient to prevent duplicate rows in the database. It is possible for two application module sessions to simultaneously attempt to create records with the same key. To prevent this from happening, create a unique index in the database for any unique constraint that you want to enforce.

Before you begin:

It may be helpful to have a general understanding of the built-in validation rules. For more information, see Using the Built-in Declarative Validation Rules.

You may also find it helpful to understand additional functionality that can be added using other validation features. For more information, see Additional Functionality for Declarative Validation.

To ensure that a key value is unique:

1. In the Applications window, double-click the desired entity object.
2. In the overview editor, click the **Business Rules** navigation tab, select the **Entity** node, and click the **Add** icon.
3. In the Add Validation Rule dialog, in the **Rule Type** dropdown list, select **UniqueKey**.
4. In the **Keys** box, select the primary or alternate key.
5. Optionally, click the **Validation Execution** tab to enter criteria for the execution of the rule, such as dependent attributes and a precondition expression.

   For more information, see Triggering Validation Execution.

   > **Best Practice:**
   >
   > While it is possible to add a precondition for a Unique Key validator, it is not a best practice. If a Unique Key validator fails to fire, for whatever reason, the cache consistency check is still performed and an error will be returned. It is generally better to add the validator and a meaningful error message.

6. Click the **Failure Handling** tab and enter or select the error message that will be shown to the user if the validation rule fails.

   For more information, see Creating Validation Error Messages.
7. Click **OK**.

What Happens When You Use a Unique Key Validator

When you use a Unique Key validator, a `<UniqueKeyValidationBean>` tag is added to the entity object's XML file. The following example shows the XML for a Unique Key validator.

```
<validation:UniqueKeyValidationBean
  Name="CustomerEORule0"
  KeyName="SCustomerIdPk">
  <validation:OnAttributes>
    <validation:Item
      Value="Id"/>
  </validation:OnAttributes>
</validation:UniqueKeyValidationBean>
```

How to Validate Based on a Comparison

The Compare validator performs a logical comparison between an entity attribute and a value. When you add a Compare validator, you specify an operator and something to compare with. You can compare the following:

- Literal value

  When you use a Compare validator with a literal value, the value in the attribute is compared against the specified literal value. When using this kind of comparison, it is important to consider data types and formats. The literal value must conform to the format specified by the data type of the entity attribute to which you are applying the rule. In all cases, the type corresponds to the type mapping for the entity attribute.

  For example, an attribute of column type DATE maps to the `oracle.jbo.domain.Date` class, which accepts dates and times in the same format accepted by `java.sql.TimeStamp` and `java.sql.Date`. You can use format masks to ensure that the format of the value in the attribute matches that of the specified literal. For information about entity object attribute type mappings, see How to Set Database and Java Data Types for an Entity Object Attribute. For information about the expected format for a particular type, refer to the Javadoc for the type class.

- Query result

  When you use this type of validator, the SQL query is executed each time the validator is executed. The validator retrieves the first row from the query result, and it uses the value of the first column in the query (of that first row) as the value to compare. Because this query cannot have any bind variables in it, this feature should be used only when selecting one column of one row of data that does not depend on the values in the current row.

- View object attribute

  When you use this type of validator, the view object's SQL query is executed each time the validator is executed. The validator retrieves the first row from the query result, and it uses the value of the selected view object attribute from that row as the value to compare. Because you cannot associate values with the view object's named bind variables, those variables can only take on their default values. Therefore this feature should be used only for selecting an attribute of one row of data that does not depend on the values in the current row.

- **View accessor** attribute

  When defining the view accessor, you can assign row-specific values to the validation view object's bind variables.

- Expression

  For information on the expression option, see Using Groovy Expressions For Business Rules and Triggers.

- Entity attribute

  The entity attribute option is available only for entity-level Compare validators.

Before you begin:

It may be helpful to have a general understanding of the built-in validation rules. For more information, see Using the Built-in Declarative Validation Rules.

You may also find it helpful to understand additional functionality that can be added using other validation features. For more information, see Additional Functionality for Declarative Validation.

To validate based on a comparison:

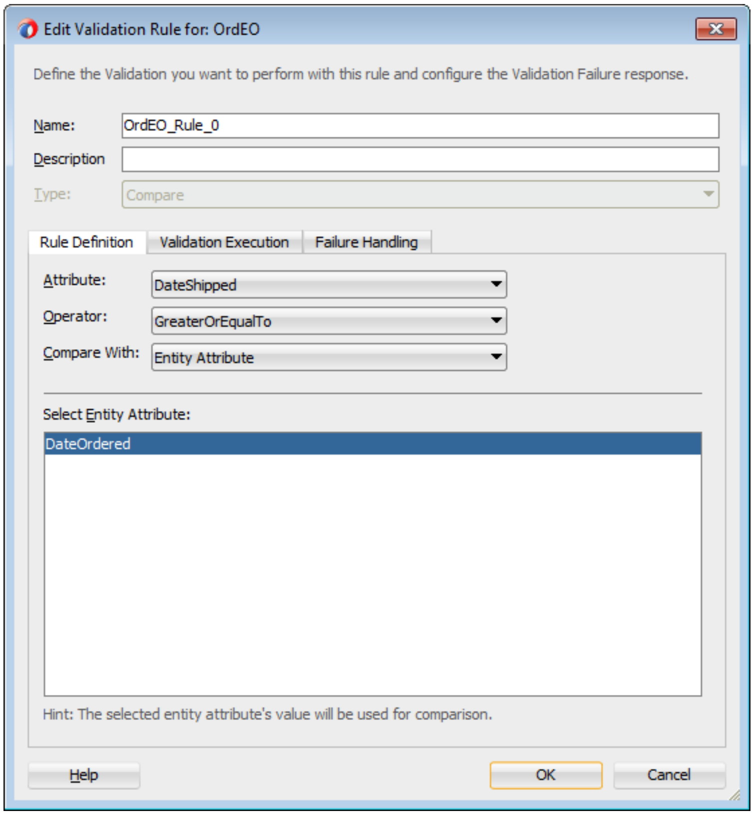
1. In the Applications window, double-click the desired entity object.
2. In the overview editor, click the **Business Rules** navigation tab, select where you want to add the validator, and click the **Add** icon.
   - To add an entity-level validator, select the **Entity** node.
   - To add an attribute-level validator, expand the **Attributes** node and select the appropriate attribute.
3. In the Add Validation Rule dialog, in the **Rule Type** dropdown list, select **Compare**. Note that the subordinate fields change depending on your choices.
4. Select the appropriate operator.
5. Select an item in the **Compare With** list, and based on your selection provide the appropriate comparison value.
6. Optionally, click the **Validation Execution** tab to enter criteria for the execution of the rule, such as dependent attributes and a precondition expression.

   For more information, see Triggering Validation Execution.
7. Click the **Failure Handling** tab and enter or select the error message that will be shown to the user if the validation rule fails.

   For more information, see Creating Validation Error Messages.
8. Click **OK**.

Figure 10-2 shows what the dialog looks like when you use an entity-level Compare validator with an entity object attribute.

Figure 10-2 Compare Validator Using an Entity Object Attribute



Edit Validation Rule for: OrdEO

Define the Validation you want to perform with this rule and configure the Validation Failure response.

Name: OrdEO_Rule_0

Description:

Type: Compare

Attribute: DateShipped

Operator: GreaterOrEqualTo

Compare With: Entity Attribute

Select Entity Attribute: DateOrdered

Hint: The selected entity attribute's value will be used for comparison.

What Happens When You Validate Based on a Comparison

When you create a Compare validator, a `<CompareValidationBean>` tag is added to an entity object's XML file. The following example shows the XML code for an entity level validator in the `OrdEO` entity object.

```
<validation:CompareValidationBean
    Name="OrdEO_Rule_0"
    ResId="oracle.summit.model.entities.OrdEO_Rule_0"
    OnAttribute="DateShipped"
    OperandType="ATTR"
    Inverse="false"
    CompareType="GREATERTHANEQUALTO"
    CompareValue="DateOrdered">
    <validation:OnAttributes>
      <validation:Item
        Value="DateShipped"/>
      <validation:Item
        Value="DateOrdered"/>
    </validation:OnAttributes>
  </validation:CompareValidationBean>
```

How to Validate Using a List of Values

The List validator compares an attribute against a list of values (LOV). When you add a List validator, you specify the type of list to choose from:

- Literal values - The validator ensures that the entity attribute is in (or not in, if specified) the list of values.
- Query result - The validator ensures that the entity attribute is in (or not in, if specified) the first column of the query's result set. The SQL query validator cannot use a bind variable, so it should be used only on a fixed, small list that you have to query from a table. All rows of the query are retrieved into memory.
- View object attribute - The validator ensures that the entity attribute is in (or not in, if specified) the view attribute. The View attribute validator cannot use a bind variable, so it should be used only on a fixed, small list that you have to query from a table. All rows of the query are retrieved into memory.
- View accessor attribute - The validator ensures that the entity attribute is in (or not in) the view accessor attribute. The view accessor is probably the most useful option, because it can take bind variables and after you've created the LOV on the user interface, a view accessor is required.

> **Best Practice:**
>
> When using a List validator, the view accessor is typically the most useful choice because you can define a **view criteria** on the view accessor to filter the view data when applicable; and when defining an LOV on a view attribute, you typically use a view accessor with a view criteria.

Before you begin:

It may be helpful to have a general understanding of the built-in validation rules. For more information, see Using the Built-in Declarative Validation Rules.

You may also find it helpful to understand additional functionality that can be added using other validation features. For more information, see Additional Functionality for Declarative Validation.

To validate using a list of values:

1. In the Applications window, double-click the desired entity object.
2. In the overview editor, click the **Business Rules** navigation tab, select where you want to add the validator, and click the **Add** icon.
   - To add an entity-level validator, select the **Entity** node.
   - To add an attribute-level validator, expand the **Attributes** node and select the appropriate attribute.
3. In the Add Validation Rule dialog, in the **Rule Type** dropdown list, select **List**.
4. In the **Attribute** list, choose the appropriate attribute.
5. In the **Operator** field, select **In** or **NotIn**, depending on whether you want an inclusive or exclusive list.
6. In the **List Type** field, select the appropriate type of list.
7. Depending on the type of list you selected, you can either enter a list of values (each value on a new line) or an SQL query, or select a view object attribute or view accessor attribute.

8. Optionally, click the **Validation Execution** tab to enter criteria for the execution of the rule, such as dependent attributes and a precondition expression.

   For more information, see Triggering Validation Execution.

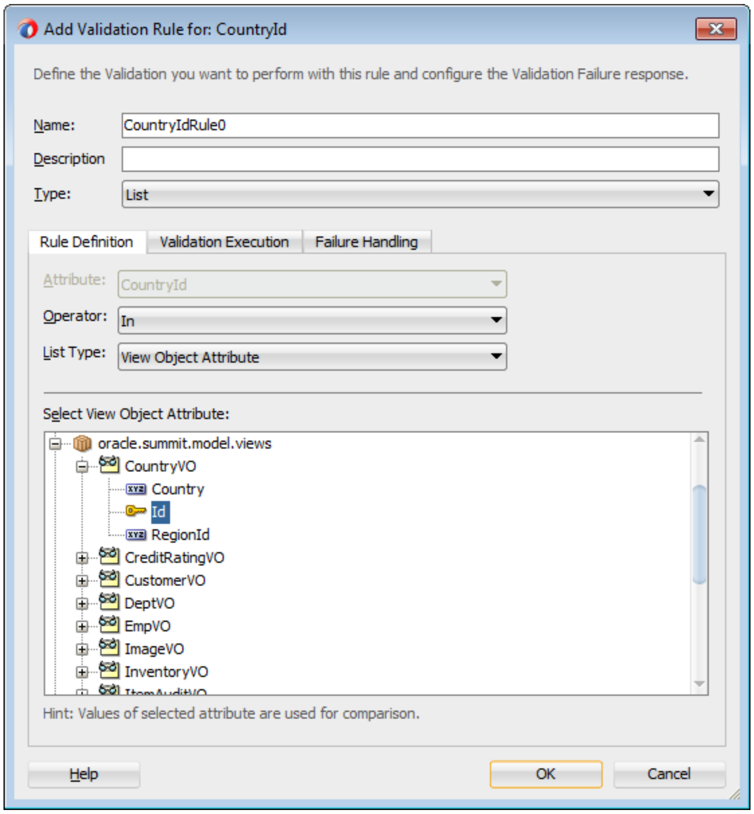
9. Click the **Failure Handling** tab and enter or select the error message that will be shown to the user if the validation rule fails.

   For more information, see Creating Validation Error Messages.

10. Click **OK**.

Figure 10-3 shows what the dialog looks like when you use a List validator with a view object attribute.

Figure 10-3 List Validator Using a View Object Attribute



Add Validation Rule for: CountryId

Define the Validation you want to perform with this rule and configure the Validation Failure response.

Name: CountryIdRule0

Description:

Type: List

Attribute: CountryId

Operator: In

List Type: View Object Attribute

Select View Object Attribute:

```
oracle.summit.model.views
    CountryVO
        Country
        Id
        RegionId
    CreditRatingVO
    CustomerVO
    DeptVO
    EmpVO
    ImageVO
    InventoryVO
    ItemAuditVO
```

Hint: Values of selected attribute are used for comparison.

What Happens When You Validate Using a List of Values

When you validate using a list of values, a `<ListValidationBean>` tag is added to an entity object's XML file. The following example shows the `CustomerEO.CountryId` attribute, which uses a view object attribute for the List validator.

```
<validation:ListValidationBean
      Name="CountryIdRule0"
      OnAttribute="CountryId"
      OperandType="JBO"
      Inverse="false"
      ListValue="oracle.summit.model.views.CountryVO.Id"/>
```

What You May Need to Know About the List Validator

The List validator is designed for validating an attribute against a relatively small set of values. If you select the **Query Result** or **View Object Attribute** type of list validation, keep in mind that the validator retrieves all of the rows from the query before performing an in-memory scan to validate whether the attribute value in question matches an attribute in the list. The query performed by the validator's SQL or view object query does not reference the value being validated in the `WHERE` clause of the query.

It is inefficient to use a validation rule when you need to determine whether a user-entered product code exists in a table of a large number of products. Instead, Using View Objects for Validation explains the technique you can use to efficiently perform SQL-based validations by using a view object to perform a targeted validation query against the database. See also Using Validators to Validate Attribute Values.

Also, if the attribute you're comparing to is a key, the Key Exists validator is more efficient than validating a list of values; and if these choices need to be translatable, you should use a static view object instead of the literal choice.

How to Make Sure a Value Falls Within a Certain Range

The Range validator performs a logical comparison between an entity attribute and a range of values. When you add a Range validator, you specify minimum and maximum literal values. The Range validator verifies that the value of the entity attribute falls within the range (or outside the range, if specified).

If you need to dynamically calculate the minimum and maximum values, or need to reference other attributes on the entity, use the Script Expression validator and provide a Groovy expression. For more information, see What You May Need to Know About Referencing Business Components in Groovy Expressions and What You May Need to Know About Manipulating Business Component Attribute Values in Groovy Expressions.

Before you begin:

It may be helpful to have a general understanding of the built-in validation rules. For more information, see Using the Built-in Declarative Validation Rules.

You may also find it useful to understand additional functionality that can be added using other validation features. For more information, see Additional Functionality for Declarative Validation.

To validate within a certain range:

1. In the Applications window, double-click the desired entity object.
2. In the overview editor, click the **Business Rules** navigation tab, select where you want to add the validator, and click the **Add** icon.
   - To add an entity-level validator, select the **Entity** node.
   - To add an attribute-level validator, expand the **Attributes** node and select the appropriate attribute.
3. In the Add Validation Rule dialog, in the **Rule Type** dropdown list, select **Range**.
4. In the **Attribute** list, select the appropriate attribute.

5. In the **Operator** field, select **Between** or **NotBetween**.
6. In the **Minimum** and **Maximum** fields, enter appropriate values.
7. Optionally, click the **Validation Execution** tab to enter criteria for the execution of the rule, such as dependent attributes and a precondition expression.

   For more information, see Triggering Validation Execution.
8. Click the **Failure Handling** tab and enter or select the error message that will be shown to the user if the validation rule fails.

   For more information, see Creating Validation Error Messages.
9. Click **OK**.

What Happens When You Use a Range Validator

When you validate against a range, a `<RangeValidationBean>` tag is added to the entity object's XML file. The following example shows the `ItemEO.Quantity` attribute with a minimum quantity of one and a maximum of 99.

```
<validation:RangeValidationBean
      Name="QuantityRule0"
      OnAttribute="Quantity"
      OperandType="LITERAL"
      Inverse="false"
      MinValue="1"
      MaxValue="99"/>
```

How to Validate Against a Number of Bytes or Characters

The Length validator validates whether the string length (in characters or bytes) of an attribute's value is less than, equal to, or greater than a specified number, or whether it lies between a pair of numbers.

Before you begin:

It may be helpful to have a general understanding of the built-in validation rules. For more information, see Using the Built-in Declarative Validation Rules.

You may also find it helpful to understand additional functionality that can be added using other validation features. For more information, see Additional Functionality for Declarative Validation.

To validate against a number of bytes or characters:

1. In the Applications window, double-click the desired entity object.
2. In the overview editor, click the **Business Rules** navigation tab, select where you want to add the validator, and click the **Add** icon.
   - To add an entity-level validator, select the **Entity** node.
   - To add an attribute-level validator, expand the **Attributes** node and select the appropriate attribute.
3. In the Add Validation Rule dialog, in the **Rule Type** dropdown list, select **Length**.
4. In the **Attribute** list, select the appropriate attribute.
5. In the **Operator** field, select how to evaluate the value.

6. In the **Comparison Type** field, select **Byte** or **Character** and enter a length.
7. Optionally, click the **Validation Execution** tab to enter criteria for the execution of the rule, such as dependent attributes and a precondition expression. For more information, see Triggering Validation Execution.
8. Click the **Failure Handling** tab and enter or select the error message that will be shown to the user if the validation rule fails. For more information, see Creating Validation Error Messages.
9. Click **OK**.

What Happens When You Validate Against a Number of Bytes or Characters

When you validate using length, a `<LengthValidationBean>` tag is added to the entity object's XML file, as shown in the following example. For example, you might have a field where the user enters a password or PIN and the application wants to validate that it is at least 6 characters long, but not longer than 10. You would use the Length validator with the Between operator and set the minimum and maximum values accordingly.

```
<validation:LengthValidationBean
  OnAttribute="pin"
  CompareType="BETWEEN"
  DataType="CHARACTER"
  MinValue="6"
  MaxValue="10"
  Inverse="false"/>
```

How to Validate Using a Regular Expression

The Regular Expression validator compares attribute values against a mask specified by a Java regular expression.

If you want to create expressions that can be personalized in metadata, you can use the Script Expression validator. For more information, see Using Groovy Expressions For Business Rules and Triggers.

Before you begin:

It may be helpful to have a general understanding of the built-in validation rules. For more information, see Using the Built-in Declarative Validation Rules.

You may also find it useful to understand additional functionality that can be added using other validation features. For more information, see Additional Functionality for Declarative Validation.

To validate using a regular expression

1. In the Applications window, double-click the desired entity object.
2. In the overview editor, click the **Business Rules** navigation tab, select where you want to add the validator, and click the **Add** icon.
   - To add an entity-level validator, select the **Entity** node.
   - To add an attribute-level validator, expand the **Attributes** node and select the appropriate attribute.

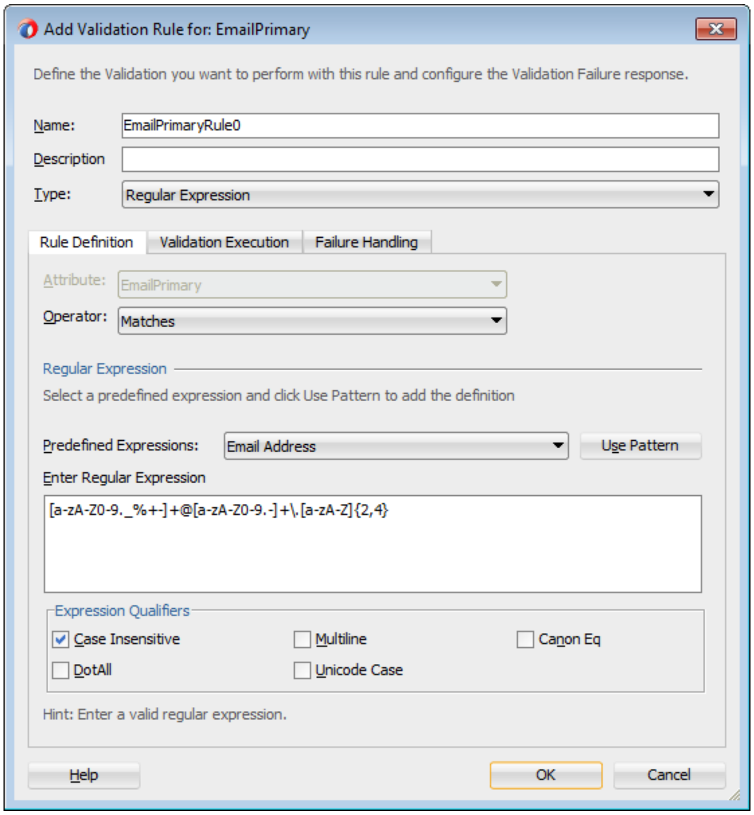
3. In the Add Validation Rule dialog, in the **Rule Type** dropdown list, select **Regular Expression**.
4. In the **Operator** field, select **Matches** or **Not Matches**.
5. To use a predefined expression (if available), you can select one from the dropdown list and click **Use Pattern**. Otherwise, write your own regular expression in the field provided.

> **Note:**
>
> You can add your own expressions to the list of predefined expressions. To add a predefined expression, add an entry in the `PredefinedRegExp.properties` file in the BC4J subdirectory of the JDeveloper system directory (for example, `C:\Documents and Settings\`*`username`*`\Application Data\JDeveloper\`*`system##`*`\o.BC4J\PredefinedRegExp.properties`).

6. Optionally, click the **Validation Execution** tab to enter criteria for the execution of the rule, such as dependent attributes and a precondition expression. For more information, see Triggering Validation Execution.
7. Click the **Failure Handling** tab and enter or select the error message that will be shown to the user if the validation rule fails. For more information, see Creating Validation Error Messages.
8. Click **OK**.

Figure 10-4 shows what the dialog looks like when you select a Regular Expression validator and validate that the `Email` attribute matches a predefined **Email Address** expression.

Figure 10-4   Regular Expression Validator Matching Email Address



Add Validation Rule for: EmailPrimary
Define the Validation you want to perform with this rule and configure the Validation Failure response.
Description
Rule Definition
Validation Execution
Failure Handling
Regular Expression
Select a predefined expression and click Use Pattern to add the definition
Use Pattern
Enter Regular Expression
Expression Qualifiers
Hint: Enter a valid regular expression.
Help
OK
Cancel

Name: EmailPrimaryRule0

Type: Regular Expression

Attribute: EmailPrimary

Operator: Matches

Predefined Expressions: Email Address

```
[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,4}
```

☑ Case Insensitive　☐ Multiline　☐ Canon Eq
☐ DotAll　☐ Unicode Case

What Happens When You Validate Using a Regular Expression

When you validate using a regular expression, a `<RegExpValidationBean>` tag is added to the entity object's XML file. The following example shows an `Email` attribute that must match a regular expression.

```
<validation:RegExpValidationBean
    Name="EmailPrimaryRule0"
    OnAttribute="EmailPrimary"
    Pattern="[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,4}"
    Flags="CaseInsensitive"
    Inverse="false"/>
```

How to Use the Average, Count, or Sum to Validate a Collection

You can use collection validation on the average, count, sum, min, or max of a collection. This validator is available only at the entity level. It is useful for validating the aggregate calculation over a collection of associated entities by way of an entity accessor to a child entity (on the many end of the association). You must select the association accessor to define the Collection validator.

Before you begin:

It may be helpful to have a general understanding of the built-in validation rules. For more information, see Using the Built-in Declarative Validation Rules.

You may also find it helpful to understand additional functionality that can be added using other validation features. For more information, see Additional Functionality for Declarative Validation.

To validate using an aggregate calculation:

1. In the Applications window, double-click the desired entity object.
2. In the overview editor, click the **Business Rules** navigation tab, select the **Entity** node, and click the **Add** icon.
3. In the Add Validation Rule dialog, in the **Rule Type** dropdown list, select **Collection**.
4. In the **Operation** field, specify the operation (sum, average, count, min, or max) to perform on the collection for comparison.
5. Select the appropriate accessor and attribute for the validation.

   The accessor you choose must be a composition association accessor. Only accessors of this type are displayed in the dropdown list.
6. Specify the operator and the comparison type and value.
7. Optionally, click the **Validation Execution** tab to enter criteria for the execution of the rule, such as dependent attributes and a precondition expression. For more information, see Triggering Validation Execution.
8. Click the **Failure Handling** tab and enter or select the error message that will be shown to the user if the validation rule fails. For more information, see Creating Validation Error Messages.
9. Click **OK**.

What Happens When You Use Collection Validation

When you validate using a Collection validator, a `<CollectionValidationBean>` tag is added to the entity object's XML file, as in the following example.

```
<validation:CollectionValidationBean
    Name="OrdEORule01"
    Accessor="ItemEO"
    CollAttribute="Quantity"
    OperandType="LITERAL"
    Inverse="false"
    CompareType="GREATERTHAN"
    CompareValue="5"
    Operation="min"/>
```

How to Determine Whether a Key Exists

The Key Exists validator is used to determine whether a key value (primary, foreign, or alternate key) exists.

There are a couple of benefits to using the Key Exists validator:

- The Key Exists validator has better performance because it first checks the cache and only goes to the database if necessary.
- Since the Key Exists validator uses the cache, it will find a key value that has been added in the current transaction, but not yet committed to the database. For

example, you add a new `Department` and then you want to link an `Employee` to that new department.

Before you begin:

It may be helpful to have a general understanding of the built-in validation rules. For more information, see Using the Built-in Declarative Validation Rules.

You may also find it useful to understand additional functionality that can be added using other validation features. For more information, see Additional Functionality for Declarative Validation.

To determine whether a value exists:

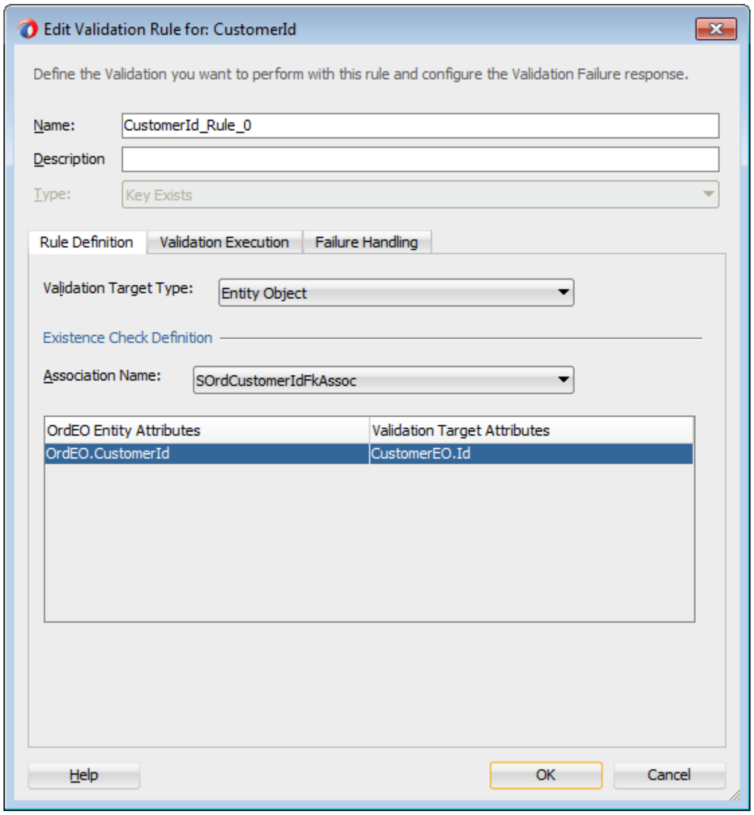
1. In the Applications window, double-click the desired entity object.
2. In the overview editor, click the **Business Rules** navigation tab, select where you want to add the validator, and click the **Add** icon.
   - To add an entity-level validator, select the **Entity** node.
   - To add an attribute-level validator, expand the **Attributes** node and select the appropriate attribute.
3. In the Add Validation Rule dialog, in the **Rule Type** dropdown list, select **Key Exists**.
4. Select the type of validation target (**Entity Object**, **View Object**, or **View Accessor**).

   If you want the Key Exists validator to be used for all view objects that use this entity attribute, select **Entity Object**.
5. Depending on the validation target, you can choose either an association or a key value.

   If you are searching for an attribute that does not exist in the **Validation Target Attributes** list, it is probably not defined as a key value. To create alternate keys, see How to Define Alternate Key Values.
6. Optionally, click the **Validation Execution** tab to enter criteria for the execution of the rule, such as dependent attributes and the validation level (entity or transaction). For more information, see Triggering Validation Execution.
7. Click the **Failure Handling** tab and enter or select the error message that will be shown to the user if the validation rule fails. For more information, see Creating Validation Error Messages.
8. Click **OK**.

Figure 10-5 shows a Key Exists validator that validates whether the `MembershipId` entered in the `PersonEO` entity object exists in the `MembershipBaseEO` entity object.

Figure 10-5   Key Exists Validator on an Entity Attribute



Edit Validation Rule for: CustomerId

Define the Validation you want to perform with this rule and configure the Validation Failure response.

Name: CustomerId_Rule_0

Description:

Type: Key Exists

Validation Target Type: Entity Object

Association Name: SOrdCustomerIdFkAssoc

OrdEO Entity Attributes	Validation Target Attributes
OrdEO.CustomerId	CustomerEO.Id

What Happens When You Use a Key Exists Validator

When you use a Key Exists validator, an `<ExistsValidationBean>` tag is created in the XML file for the entity object, as in the following example.

```
<validation:ExistsValidationBean
      Name="CustomerId_Rule_0"
      ResId="oracle.summit.model.entities.OrdEO.CustomerId_Rule_0"
      OperandType="EO"
      AssocName="oracle.summit.model.entities.assoc.SOrdCustomerIdFkAssoc"/>
```

What You May Need to Know About Declarative Validators and View Accessors

When using declarative validators you must consider how your validation will interact with expected input. The combination of declarative validators and view accessors provides a simple yet powerful alternative to coding. But, as powerful as the combination is, you still need to consider how data composition can impact performance.

Consider a scenario where you have the following:

- A `ServiceRequestEO` entity object with `Product` and `RequestType` attributes, and a view accessor that allows it to access the `RequestTypeVO` view object

- A RequestTypeVO view object with a query specifying the Product attribute as a bind parameter

The valid list of RequestTypes varies by Product. So, to validate the RequestType attribute, you use a List validator using the view accessor.

Now lets add a set of new service requests. For the first service request (row), the List validator binds the value of the Product attribute to the view accessor and executes it. For each subsequent service request the List validator compares the new value of the Product attribute to the currently bound value.

- If the value of Product matches, the current RowSet object is retained.
- If the value of Product has changed, the new value is bound and the view accessor re-executed.

Now consider the expected composition of input data. For example, the same products could appear in the input multiple times. If you simply validate the data in the order received, you might end up with the following:

1. Dryer (initial query)
2. Washing Machine (re-execute view accessor)
3. Dish Washer (re-execute view accessor)
4. Washing Machine (re-execute view accessor)
5. Dryer (re-execute view accessor)

In this case, the validator will execute 5 queries to get 3 distinct row sets. As an alternative, you can add an ORDER BY clause to the RequestTypeVO to sort it by Product. In this case, the validator would execute the query only once each for Washing Machine and Dryer.

1. Dish Washer (initial query)
2. Dryer (re-execute view accessor)
3. Dryer
4. Washing Machine (re-execute view accessor)
5. Washing Machine

A small difference on a data set this size, but multiplied over larger data sets and many users this could easily become an issue. An ORDER BY clause is not a solution to every issue, but this example illustrates how data composition can impact performance.

Using Entity-Level Triggers

ADF Business Components allows you to apply triggers in an entity object's lifecycle. Triggers execute at these set lifecycle events such as before a delete, after it, or before an insert operation.

Entity-level triggers enable you to implement expressions that are executed in response to trigger points in the entity object life cycle, such as before an insert operation or after a delete.

For example, you can evaluate or assert the value of an attribute prior to inserting an entity object or after updating an entity object. For a list of the available trigger points, see the online help.

How to Implement Business Rules Using Entity-Level Triggers

You can use Groovy script to implement business rules that are executed in response to entity-level triggers.

Before you begin:
It may be helpful to have an understanding of entity-level triggers. See Using Entity-Level Triggers.

You may also find it helpful to understand additional functionality that can be added using other validation features. See Additional Functionality for Declarative Validation.

To add a trigger:

1. In the Applications window, double-click the entity object you want to add a trigger to.
2. In the overview editor, click the **Business Rules** navigation tab.
3. On the Business Rules page, select the Entity-Level Triggers node, and click the **Create New Trigger** icon.
4. Select the appropriate trigger point from the **Type** dropdown list.
5. Enter a Groovy expression.
6. Click the **Failure Handling** tab and enter or select the error message that will be shown to the user if the expression fails.

See Creating Validation Error Messages to read more on creating validation error messages.
When a Groovy expression is evaluated at runtime, it is assumed to be untrusted. For more information, see What You May Need to Know About Untrusted Groovy Expressions.

What Happens When You Create an Entity-Level Trigger

When you create a trigger, a `<Trigger>` tag is added to the entity object's XML file, as shown in the following example.

```
<Trigger
   Name="BeforeUpdate">
   <validation:ExpressionValidationBean
     Name="BeforeUpdateExpression0"
     OperandType="EXPR"
     Inverse="false">
     <validation:TransientExpression
       Name="ValidationRuleScript"
       trustMode="untrusted"
       hasReturn="false"
       CodeSourceName="ItemEORow"/>
   </validation:ExpressionValidationBean>
</Trigger>
```

Using Groovy Expressions For Business Rules and Triggers

You can use Groovy expressions to build business rules and triggers. These expressions are stored as part of the ADF entity object's XML definition as well as the `.bcs` file associated with the entity object.

Groovy expressions are Java-like scripting code stored in the XML definition of an entity object and the `.bcs` file associated with the entity object. Because Groovy scripts are stored in the `.bcs` file, you can change the expression values in this file.

For more information about using Groovy script in your entity object business logic, see Using Groovy Scripting Language with Business Components.

How to Reference Entity Object Methods in Groovy Expressions

You can call methods on the current entity instance using the `source` property of the current object. The `source` property allows you to access to the entity instance.

If the method is a non-boolean type and the method name is `getXyzAbc()` with no arguments, then you access its value as if it were a property named `XyzAbc`. For a boolean-valued property, the same holds true but the JavaBean naming pattern for the getter method changes to recognize `isXyzAbc()` instead of `getXyzAbc()`. If the method on your entity object does not match the JavaBean getter method naming pattern, or if it takes one or more arguments, then you must call it like a method using its complete name.

For example, say you have an entity object with the four methods shown in the following example.

```
public boolean isNewRow() {
 System.out.println("## isNewRow() accessed ##");
 return true;
}

public boolean isNewRow(int n) {
 System.out.println("## isNewRow(int n) accessed ##");
 return true;
}

public boolean testWhetherRowIsNew() {
 System.out.println("## testWhetherRowIsNew() accessed ##");
 return true;
}

public boolean testWhetherRowIsNew(int n) {
 System.out.println("## testWhetherRowIsNew(int n) accessed ##");
 return true;
}
```

Then the following Groovy validation condition would execute them all, one of them being executed twice, as shown in this example.

```
newRow && source.newRow && source.isNewRow(5) && source.testWhetherRowIsNew() &&
source.testWhetherRowIsNew(5)
```

By running this example and forcing entity validation to occur, you would see the following diagnostic output in the log window:

```
## isNewRow() accessed ##
## isNewRow() accessed ##
## isNewRow(int n) accessed ##
## testWhetherRowIsNew() accessed ##
## testWhetherRowIsNew(int n) accessed ##
```

Notice the slightly different syntax for the reference to a method whose name matches the JavaBeans property getter method naming pattern. Both `newRow` and `source.newRow` work to access the boolean-valued, JavaBeans getter-style method that has no arguments. But because the `testWhetherRowIsNew()` method does not match the JavaBeans getter method naming pattern, and the third `isNewRow()` method takes an argument, then you must call them like methods using their complete name.

How to Validate Using a True/False Expression

You can use a Groovy expression to return a true/false statement. The Script Expression validator *requires* that the expression either return `true` or `false`, or that it calls the `adf.error.raise/warn()` method. A common use of this feature would be to validate an attribute value, for example, to make sure that an account number is valid.

> **Note:**
>
> Using the `adf.error.raise/warn()` method (rather than simply returning `true` or `false`) allows you to define the message text to show to the user, and to associate an entity-level validator with a specific attribute. For more information, see How to Conditionally Raise Error Messages Using Groovy.

Before you begin:

It may be helpful to have an understanding of the use of Groovy in validation rules. For more information, see Using Groovy Expressions For Business Rules and Triggers.

You may also find it useful to understand additional functionality that can be added using other validation features. For more information, see Additional Functionality for Declarative Validation.

To validate using a true/false expression:

1. In the Applications window, double-click the desired entity object.
2. In the overview editor, click the **Business Rules** navigation tab, select where you want to add the validator, and click the dropdown of the plus icon to select **New Validator**.
   - To add an entity-level validator, select the **Entity** node.
   - To add an attribute-level validator, expand the **Attributes** node and select the appropriate attribute.
3. In the Add Validation Rule dialog, in the **Rule Type** dropdown list, select **Expression**.
4. Enter a validation expression in the **Expression** field.
5. Optionally, click the **Validation Execution** tab to enter criteria for the execution of the rule, such as dependent attributes and a precondition expression in the **Expression** field. For more information, see Triggering Validation Execution.

6. Click the **Failure Handling** tab and enter or select the error message that will be shown to the user if the validation rule fails. For more information, see Creating Validation Error Messages.
7. Click **OK**.

The following Groovy script example validates account numbers based on the Luhn algorithm, a checksum formula in widespread use.

```
@ValidatorExpression(name="PaymentOptionIdRule0",
attributeName="PaymentOptionId")
def PaymentOptionId_PaymentOptionIdRule0_ValidationRuleScript_ValidationRule()
{
String acctnumber = newValue;
 sumofdigits = 0;
 digit = 0;
 addend = 0;
 timesTwo = false;
 range = acctnumber.length()-1..0
  range.each {i ->
  digit = Integer.parseInt (acctnumber.substring (i, i + 1));
  if (timesTwo) {
   addend = digit * 2;
   if (addend > 9) {
    addend -= 9;
   }
  }
  else {
   addend = digit;
  }
  sumofdigits += addend;
  timesTwo = !timesTwo;
 }
 modulus = sumofdigits % 10;
 return modulus == 0;
}
```

What Happens When You Add a Groovy Expression

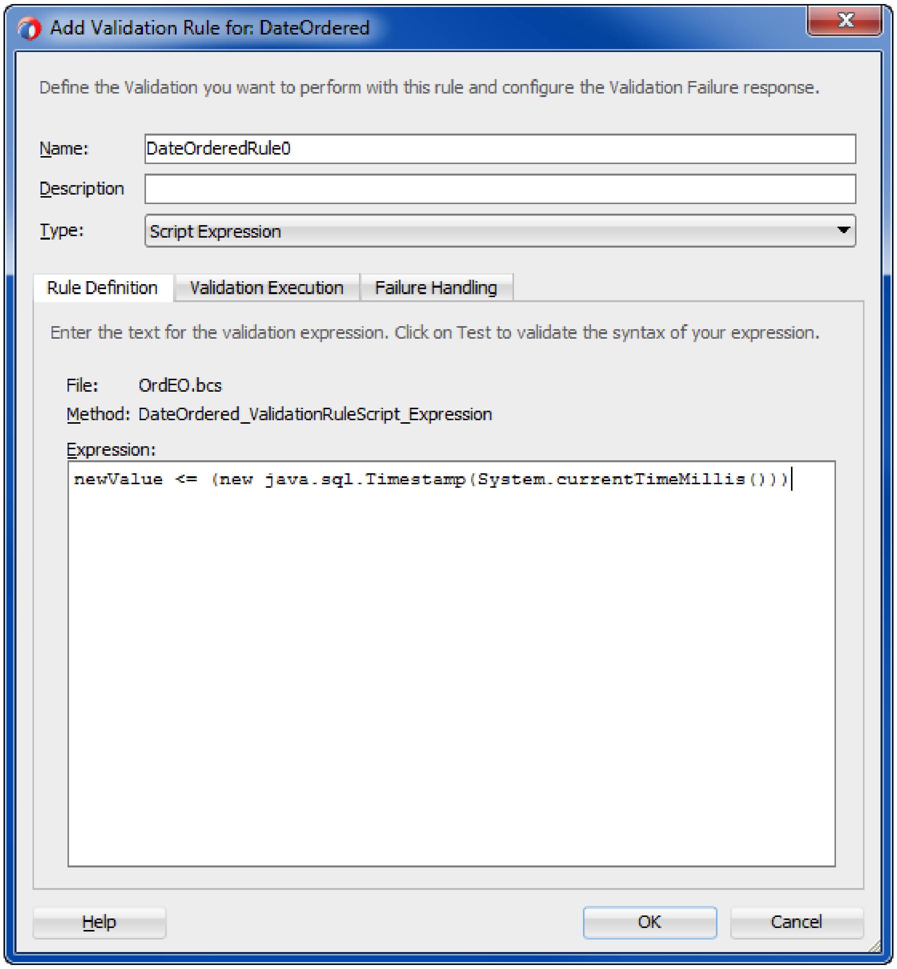
When you create a Groovy expression, it is saved in the entity object's XML component. The following example shows a `DateOrdered` attribute in an entity object. The Groovy expression is wrapped by a `<validation:ExpressionValidationBean>` tag. The Groovy script is enclosed in `OrdEO.bcs` file as indicated by `CodeSourceName="OrdEORow”` in the XML definition.

```
<Attribute
  Name="DateOrdered"
  ColumnName="DATE_ORDERED"
  SQLType="TIMESTAMP"
  Type="java.sql.Date"
  ColumnType="DATE"
  TableName="S_ORD">
  <DesignTime>
    <Attr Name="_DisplaySize" Value="7"/>
  </DesignTime>
  <validation:ExpressionValidationBean
    Name="DateOrderedRule0"
    OperandType="EXPR"
    Inverse="false">
    <validation:TransientExpression
```

```
      Name="ValidationRuleScript"
      trustMode="untrusted"
      CodeSourceName="OrdEORow"/>
  </validation:ExpressionValidationBean>
</Attribute>
```

Figure 10-6 shows the validation expression in the Edit Validation Rule dialog.

Figure 10-6 Validation Expression for Date Ordered Attribute



Define the Validation you want to perform with this rule and configure the Validation Failure response.

Name: DateOrderedRule0

Description

Type: Script Expression

Enter the text for the validation expression. Click on Test to validate the syntax of your expression.

File: OrdEO.bcs

Method: DateOrdered_ValidationRuleScript_Expression

Expression:

```
newValue <= (new java.sql.Timestamp(System.currentTimeMillis()))
```

Triggering Validation Execution

ADF Business Components allows you to define attribute level validations, as well as entity-level validations.

JDeveloper allows you to select the attributes that trigger validation, so that validation execution happens only when one of the triggering attributes is dirty. In previous releases of JDeveloper, an entity-level validator would fire on an attribute whenever the entity as a whole was dirty. This feature is described in How to Specify Which Attributes Fire Validation.

JDeveloper also allows you to specify a precondition for the execution of a validator (as described in How to Set Preconditions for Validation) and set transaction-level validation (described in How to Set Transaction-Level Validation).

How to Specify Which Attributes Fire Validation

When defining a validator at the entity level, you have the option of selecting one or more attributes of the entity object that, when changed, trigger execution of the validator.

> **Note:**
>
> When the validity of one attribute is dependent on the value in another attribute, the validation should be performed as entity validation, not attribute validation. You can set validation execution order on the entity level or attribute level.

If you do not specify one or more dependent attributes, the validator will fire whenever the entity is dirty. Firing execution only when required makes your application more performant.

Before you begin:

It may be helpful to have an understanding of how validation rules are triggered. For more information, see Triggering Validation Execution.

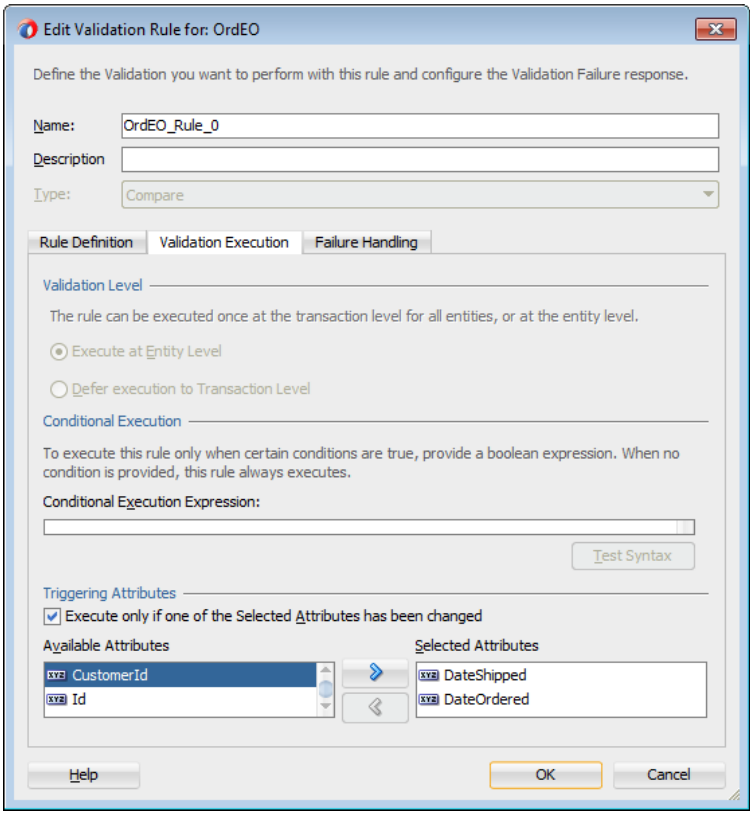
You may also find it useful to understand additional functionality that can be added using other validation features. For more information, see Additional Functionality for Declarative Validation.

To specify which attributes fire validation:

1. In the Applications window, double-click the desired entity object.
2. In the overview editor, click the **Business Rules** navigation tab, select the **Entity** node, and click the **Edit** icon.
3. In the Edit Validation Rule dialog, click the **Validation Execution** tab.
4. Select the attributes that will fire validation.
5. Click **OK**.

For example, in the `SummitADF` application workspace, the `OrdEO` entity object has an entity-level validator that requires the `DateShipped` attribute to be later than the `DateOrdered` attribute. As shown in Figure 10-7, this validator is set to be executed on the entity object only when either the `DateShipped` attribute or the `DateOrdered` attribute has been changed.

Figure 10-7 Triggering attributes on the Validation Execution tab of the Edit Validation Rule dialog



Edit Validation Rule for: OrdEO

Define the Validation you want to perform with this rule and configure the Validation Failure response.

Name: OrdEO_Rule_0

Description

Type: Compare

Validation Level

The rule can be executed once at the transaction level for all entities, or at the entity level.

● Execute at Entity Level

○ Defer execution to Transaction Level

Conditional Execution

To execute this rule only when certain conditions are true, provide a boolean expression. When no condition is provided, this rule always executes.

Conditional Execution Expression:

Triggering Attributes

☑ Execute only if one of the Selected Attributes has been changed

Available Attributes: CustomerId, Id

Selected Attributes: DateShipped, DateOrdered

What Happens When You Constrain Validation Execution with Triggering Attributes

When you specify triggering attributes on the **Validation Execution** tab of the Edit Validation Rule dialog, JDeveloper adds a `<validation:OnAttributes>` tag to the validator definition in the entity object's XML file. The following example shows the XML code for the entity-level validator for the `OrdEO` entity object in the core Summit ADF sample application.

```
<validation:CompareValidationBean
   Name="OrdEO_Rule_0"
   ResId="oracle.summit.model.entities.OrdEO_Rule_0"
   OnAttribute="DateShipped"
   OperandType="ATTR"
   Inverse="false"
   CompareType="GREATERTHANEQUALTO"
   CompareValue="DateOrdered">
   <validation:OnAttributes>
     <validation:Item
       Value="DateShipped"/>
     <validation:Item
       Value="DateOrdered"/>
   </validation:OnAttributes>
</validation:CompareValidationBean>
```

How to Set Preconditions for Validation

The **Validation Execution** tab (on the Add/Edit Validation Rule dialog) allows you to add a Groovy expression that serves as a precondition. If you enter an expression in the **Conditional Execution Expression** box, the validator is executed only if the condition evaluates `True`.

> **Best Practice:**
>
> While it is possible to add a precondition for a Unique Key validator, it is not a best practice. If a Unique Key validator fails to fire, for whatever reason, the cache consistency check is still performed and an error will be returned. It is generally better to add the validator and a meaningful error message.

How to Set Transaction-Level Validation

Performing a validation during the transaction level (rather than entity level) means that the validation will be performed after all entity-level validation is performed. For this reason, it may be useful if you want to ensure that a validator is performed at the end of the process.

In addition, the Key Exists validator is more performant with bulk transactions if it is run as a transaction level validator since it will be run only once for all entities in the transaction (of the same type), rather than once per entity. This will result in improved performance if the validator has to go to the database.

> **Note:**
>
> Transaction-level validation is only applicable to Key Exists and Method entity validators.

Before you begin:

It may be helpful to have an understanding of how validation rules are triggered. For more information, see Triggering Validation Execution.

You may also find it useful to understand additional functionality that can be added using other validation features. For more information, see Additional Functionality for Declarative Validation.

To specify entity-level or transaction-level validation:

1. In the Applications window, double-click the desired entity object.
2. In the overview editor, click the **Business Rules** navigation tab, select an entity-level validation rule, and click the **Edit** icon.
3. In the Edit Validation Rule dialog, click the **Validation Execution** tab.
4. Select **Execute at Entity Level** or **Defer Execution to Transaction Level**.

5. Click **OK**.

What You May Need to Know About the Order of Validation Execution

You cannot control the order in which attributes are validated – they are always validated in the order they appear in the entity definition. You can order validations for a given attribute (or for the entity), but you cannot reorder the attributes themselves.

Creating Validation Error Messages

ADF Business Components allows you to create validation error messages by entering text that provides self-explanatory information to the end-user on the problem and troubleshooting tips.

Validation error messages provide important information for the user: the message should convey what went wrong and how to fix it.

How to Create Error Messages for Validators and Triggers

When you create or edit a validation rule or trigger, you can enter text to help the user determine what caused the error.

Before you begin:

It may be helpful to have an understanding of error messages in validation rules. For more information, see Creating Validation Error Messages.

You may also find it useful to understand additional functionality that can be added using other validation features. For more information, see Additional Functionality for Declarative Validation.

To create error messages for validators or triggers:

1. In the Applications window, double-click the desired entity object.
2. In the overview editor, click the **Business Rules** navigation tab, select a validation rule or trigger, and click the **Edit** icon.
3. In the Edit dialog, click the **Failure Handling** tab.
4. In the **Message Text** field, enter your error message.

   You can also define error messages in a message bundle file. To select a previously defined error message or to define a new one in a message bundle file, click the **Select Message** icon.

   > **Note:**
   >
   > Triggers and Script Expression validators allow you to enter more than one error message. This is useful if the validation script conditionally returns different error or warning messages. For more information, see How to Conditionally Raise Error Messages Using Groovy.

5. You can optionally include message tokens in the body of the message, and define them in the **Token Message Expressions** list.

Figure 10-8 shows the failure message for a validation rule in the OrdEO entity object that contains message tokens. For more information on this feature, see How to Embed a Groovy Expression in an Error Message.

6. Click **OK**.

How to Localize Validation Messages

The error message is a translatable string and is managed in the same way as translatable UI **control hints** in an entity object message bundle class. To view the error message for the defined rule in the message bundle class, locate the String key in the message bundle that corresponds to the ResId property in the XML document entry for the validator. For example, The following example shows a message bundle where the NAME_CANNOT_BEGIN_WITH_U key appears with the error message for the default locale.

Resource bundles can be created for your applications either as a list resource bundle (as shown in the sample below), as a properties bundle, or as an XLIFF resource bundle. For more information about using translatable strings in a resource bundle, see Working with Resource Bundles.

```
package devguide.advanced.customerrors;
import java.util.ListResourceBundle;

public class CustomMessageBundle extends ListResourceBundle {
  private static final Object[][] sMessageStrings = new String[][] {
  // other strings here
  {"NAME_CANNOT_BEGIN_WITH_U", "The name cannot begin with the letter u!"},
  // other strings here
  };
  // etc.
}
```

How to Conditionally Raise Error Messages Using Groovy

You can use the adf.error.raise() and adf.error.warn() methods to conditionally raise one error message or another depending upon branching in the Groovy expression. For example, if an attribute value is *x*, then validate as follows, and if the validation fails, raise error messageA; whereas if the attribute value is *y*, then instead validate a different way and if validation fails, raise error messageB.

If the expression returns false (versus raising a specific error message using the raise() method), the validator calls the first error message associated with the validator.

The syntax of the raise() method takes one required parameter (the msgId to use from the message bundle), and optionally can take the attrName parameter. If you pass in the AttrName, the error is associated with that attribute even if the validation is assigned to the entity.

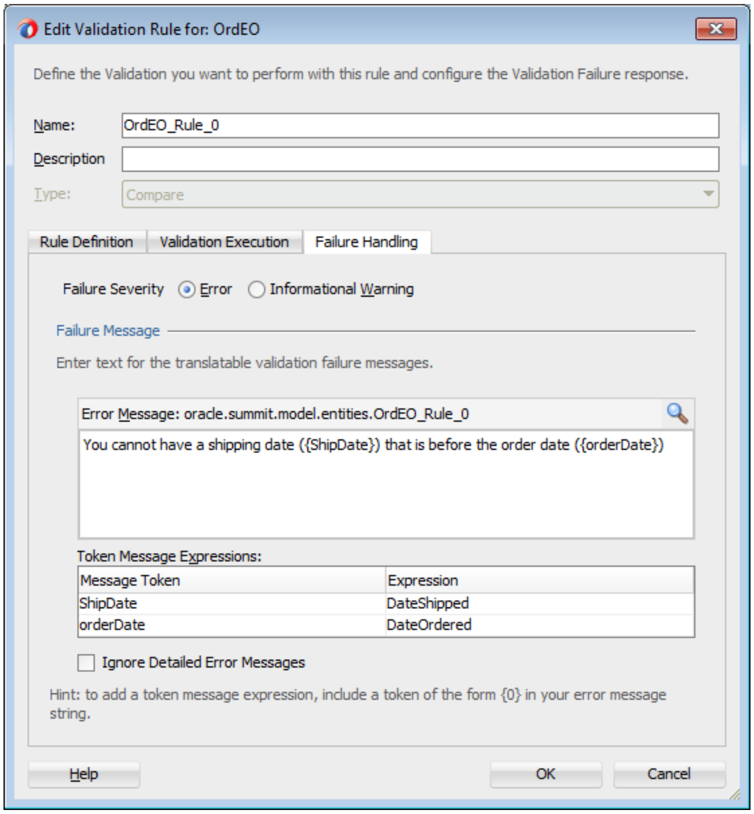
You can use either adf.error.raise() or adf.error.warn() methods, depending on whether you want to throw an exception, or whether you want processing to continue, as described in Setting the Severity Level for Validation Exceptions.

How to Embed a Groovy Expression in an Error Message

A validator's error message can contain embedded expressions that are resolved by the server at runtime. To access this feature, simply enter a named token delimited by curly braces (for example, `{2}` or `{errorParam}`) in the error message text where you want the result of the Groovy expression to appear.

After entering the token into the text of the error message (on the **Failure Handling** tab of the Edit Validation Rule dialog), the **Token Message Expressions** table at the bottom of the dialog displays a row that allows you to enter a Groovy expression for the token. Figure 10-8 shows the failure message for a validation rule in the `OrdEO` entity object that contains message tokens.

Figure 10-8 Using Message Tokens in a Failure Message



Define the Validation you want to perform with this rule and configure the Validation Failure response.

Name: OrdEO_Rule_0

Description:

Type: Compare

Failure Severity ● Error ○ Informational Warning

Failure Message

Enter text for the translatable validation failure messages.

Error Message: oracle.summit.model.entities.OrdEO_Rule_0

You cannot have a shipping date ({ShipDate}) that is before the order date ({orderDate})

Token Message Expressions:

Message Token	Expression
ShipDate	DateShipped
orderDate	DateOrdered

☐ Ignore Detailed Error Messages

Hint: to add a token message expression, include a token of the form {0} in your error message string.

The expressions shown in Figure 10-8 are Groovy expressions that return the labels of the specified fields. You can also use Groovy expressions to access attribute values and other business components objects. For example, you can use the Groovy expression `newValue` to return the entered value.

The Groovy syntax to retrieve a value from a view accessor is *`accessorName`*`.currentRow.`*`AttributeName`*. For example, the Groovy expression `MyEmpAccessor.currentRow.Job` returns the value of the `Job` attribute in the current row of the `MyEmpAccessor` view accessor.

The Groovy expression can also be more complex, as in the following example, which shows an expression in the error message for the List validation rule for the `PaymentTypeId` attribute in the `OrdEO` entity object.

```
cr = CustomerEO.CreditRatingId
ratings = [1, 2]
if (cr  in ratings)
{return true}
else
{return false}
```

For more information about accessing business components objects using Groovy, see Using Groovy Scripting Language with Business Components.

Setting the Severity Level for Validation Exceptions

ADF Business Components allows you to set severity levels for validations exceptions as Informational Warning or Error. The latter level if set does not allow the end-user to proceed without fixing the issue.

You can set the severity level for validation exceptions to two levels, Informational Warning and Error. If you set the severity level to Informational Warning, an error message will display, but processing will continue. If you set the validation level to Error, the user will not be able to proceed until you have fixed the error.

Under most circumstances you will use the Error level for validation exceptions, so this is the default setting. However, you might want to implement a Informational Warning message if the user has a certain security clearance. For example, a store manager may want to be able to make changes that would surface as an error if a clerk tried to do the same thing.

When the severity level is set to **Informational Warning**, the error message is presented with a warning icon rather than an error icon and after acknowledging the warning, you can proceed to save the changes.

To set the severity level for validation exceptions, use the **Failure Handling** tab of the Add Validation Rule dialog.

To set the severity level of a validation exception:

1. In the Applications window, double-click the desired entity object.
2. In the overview editor, click the **Business Rules** navigation tab.
3. On the Business Rules page, select an existing validation rule and click the **Edit** icon, or click the **Add** icon to create a new rule.
4. In the Edit/Add Validation Rule dialog, click the **Failure Handling** tab and select the option for either **Error** or **Informational Warning**.
5. Click **OK**.

Bulk Validation in SQL

ADF Business Components provides bulk validation facility to improve performance of batch-load applications. The validation is applied on primary keys, alternative keys and foreign keys.

To improve the performance of batch-load applications, such as data synchronization programs, Oracle ADF employs bulk validation for primary keys (including alternate keys) and foreign keys.

When the Key Exists validator is configured to defer validation until the transaction commits, or when the rows are being updated or inserted through the `process`*`XXX`* methods of the ADF Business Components service layer, the validation cache is preloaded. This behavior uses the normal row-by-row derivation and validation logic, but uses validation logic that checks a memory cache before making queries to the database. Performance is improved by preloading the memory cache using bulk SQL operations based on the inbound data.

11 Working Programmatically with View Objects

This chapter describes programmatic uses of the ADF Business Components API that you can use while designing and working with ADF view objects in an Oracle ADF application.
This chapter includes the following sections:

Generating Custom Java Classes for a View Object

Use custom Java code in a custom Java class associated with an ADF Business Components view object to add custom logic and validation methods.

As you've seen, all of the basic querying functionality of a view object can be achieved without using custom Java code. Clients can retrieve and iterate through the data of any SQL query without resorting to any custom code on the **view object** developer's part. In short, for many view objects, once you have defined the SQL statement, you're done. However, it's important to understand how to enable custom Java generation for a view object when your needs might require it. For example, reasons you might write code in a custom Java class include:

- To add validation methods (although Groovy Script expressions can provide this support without needing Java)
- To add custom logic
- To augment built-in behavior

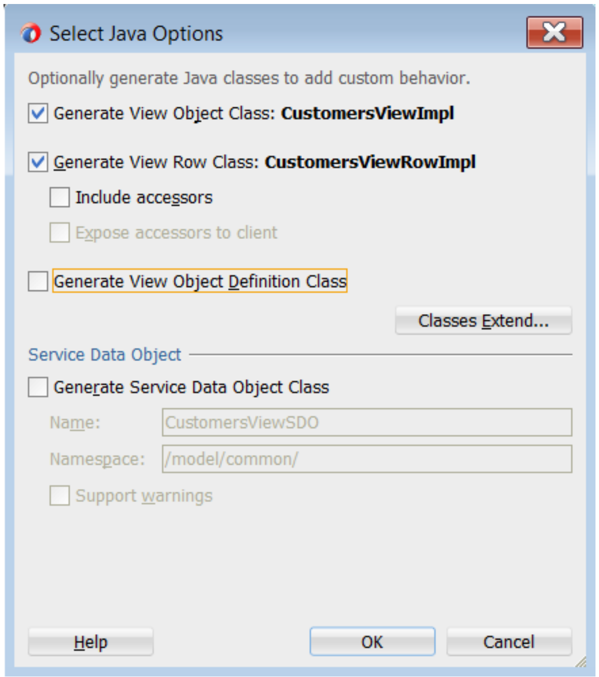
Most Commonly Used ADF Business Components Methods provides a quick reference to the most common code that you will typically write, use, and override in your custom view object and view row classes.

How To Generate Custom Classes

To enable the generation of custom Java classes for a view object, use the **Java** page of the view object overview editor. As shown in the figure below, there are three optional Java classes that can be related to a view object. The first two in the list are the most commonly used:

- **View object class**, which represents the component that performs the query and controls the execution lifecycle
- **View row class**, which represents each row in the query result

Figure 11-1 View Object Custom Java Generation Options



Select Java Options

Optionally generate Java classes to add custom behavior.

☑ Generate View Object Class: **CustomersViewImpl**

☑ Generate View Row Class: **CustomersViewRowImpl**

□ Include accessors

□ Expose accessors to client

□ Generate View Object Definition Class

Service Data Object

□ Generate Service Data Object Class

Name: CustomersViewSDO

Namespace: /model/common/

□ Support warnings

Generating View Row Attribute Accessors

When you enable the generation of a custom view row class, if you also select the **Include Accessors** checkbox, as shown in the Select Java Options dialog, then JDeveloper generates getter and setter methods for each attribute in the view row. For example, for the `ProductView` view object, the corresponding custom `ProductViewRowImpl.java` class might have methods like this generated in it:

```
public String getName() {...}
public void setName(String value) {...}
public String getShortDesc() {...}
public void setShortDesc(String value) {...}
public String getImageId() {...}
public void setImageId(Integer value) {...}
public String getSuggestedWhlslPrice() {...}
public void setSuggestedWhlslPrice(BigDecimal value) {...}
```

These methods allow you to work with the row data with compile-time checking of the correct datatype usage. That is, instead of writing a line like this one that gets the value of the Name attribute:

```
String name = row.getAttribute("Name");
```

you can write the code like:

```
String name = row.getName();
```

You can see that with the latter approach, the Java compiler would catch a typographical error had you accidentally typed FullName instead of Name:

```
// spelling name wrong gives compile error
String name = row.getFullName();
```

Without the generated view row accessor methods, an incorrect line of code like the following cannot be caught by the compiler:

```
// Both attribute name and type cast are wrong, but compiler cannot catch it
Integer name = (Integer)row.getAttribute("FullName");
```

It contains both an incorrectly spelled attribute name, as well as an incorrectly typed cast of the getAttribute() return value. Using the generic APIs on the Row interface, errors of this kind will raise exceptions at runtime instead of being caught at compile time.

Exposing View Row Accessors to Clients

When enabling the generation of a custom view row class, if you would like to generate a view row attribute accessor, select the **Expose Accessor to the Client** checkbox. This causes an additional custom row interface to be generated which application clients can use to access custom methods on the row without depending directly on the implementation class.

> **Best Practice:**
>
> When you create client code for business components, you should use business service interfaces rather than concrete classes. Using the interface instead of the implementation class, ensures that client code does not need to change when your server-side implementation does. For more details working with client code, see in What You May Need to Know About Custom Interface Support.

For example, in the case of the ProductView view object, exposing the accessors to the client will generate a custom row interface named ProductViewRow. This interface is created in the common subpackage of the package in which the view object resides. Having the row interface allows clients to write code that accesses the attributes of query results in a strongly typed manner. The following example shows a TestClient sample client program that casts the results of the productByName.first() method to the ProductViewRow interface so that it can make method calls like printAllAttributes() and testSomethingOnProductRows().

```
package oracle.summit.model.extend;
```

```
import oracle.jbo.*;
import oracle.jbo.client.Configuration;

import oracle.summit.model.extend.common.ProductView;
import oracle.summit.model.extend.common.ProductViewEx;
import oracle.summit.model.extend.common.ProductViewExRow;
import oracle.summit.model.extend.common.ProductViewRow;

public class TestClient {
  public static void main(String[] args) {
    String amDef = "oracle.summit.model.extend.AppModule";
    String config = "AppModuleLocal";
    ApplicationModule am =
                  Configuration.createRootApplicationModule(amDef,config);
    ProductView products = (ProductView)am.findViewObject("ProductView1");
    products.executeQuery();
    ProductViewRow product = (ProductViewRow)products.first();
    printAllAttributes(products,product);
    testSomethingOnProductsRow(product);
    products = (ProductView)am.findViewObject("ProductViewEx1");
    ProductViewEx productsByName = (ProductViewEx)products;
    productsByName.setbv_ProductName("bunny");
    productsByName.executeQuery();
    product = (ProductViewRow)productsByName.first();
    printAllAttributes(productsByName,product);
    testSomethingOnProductsRow(product);
    am.getTransaction().rollback();
    Configuration.releaseRootApplicationModule(am,true);
  }
  private static void testSomethingOnProductsRow(ProductViewRow product) {
    try {
      if (product instanceof ProductViewExRow) {
        ProductViewExRow productByName = (ProductViewExRow)product;
        productByName.someExtraFeature("Test");
      }
      product.setName("Q");
      System.out.println("Setting the Name attribute to 'Q' succeeded.");
    }
    catch (ValidationException v) {
      System.out.println(v.getLocalizedMessage());
    }
  }
  private static void printAllAttributes(ViewObject vo, Row r) {
    String viewObjName = vo.getName();
    System.out.println("Printing attribute for a row in VO '"+ viewObjName+"'");
    StructureDef def = r.getStructureDef();
    StringBuilder sb = new StringBuilder();
    int numAttrs = def.getAttributeCount();
    AttributeDef[] attrDefs = def.getAttributeDefs();
    for (int z = 0; z < numAttrs; z++) {
      Object value = r.getAttribute(z);
      sb.append(z > 0 ? "  " : "")
        .append(attrDefs[z].getName())
        .append("=")
        .append(value == null ? "<null>" : value)
        .append(z < numAttrs - 1 ? "\n" : "");
    }
    System.out.println(sb.toString());
  }
}
```

Configuring Default Java Generation Preferences

You've seen how to generate custom Java classes for your view objects when you need to customize their runtime behavior, or if you simply prefer to have strongly typed access to bind variables or view row attributes.

To change the default settings that control how JDeveloper generates Java classes, choose **Tools | Preferences** and open the **ADF Business Components** page. The settings you choose will apply to all future business components you create.

Oracle recommends that developers getting started with ADF Business Components set their preference to generate no custom Java classes by default. As you run into specific needs, you can enable just the bit of custom Java you need for that one component. Over time, you'll discover which set of defaults works best for you.

What Happens When You Generate Custom Classes

When you choose to generate one or more custom Java classes, JDeveloper creates the Java file(s) you've indicated.

For example, in the case of a view object named `oracle.summit.model.extend.ProductView`, the default names for its custom Java files will be `ProductViewImpl.java` for the view object class and `ProductViewRowImpl.java` for the view row class. Both files get created in the same `./model/extend` directory as the component's XML document file.

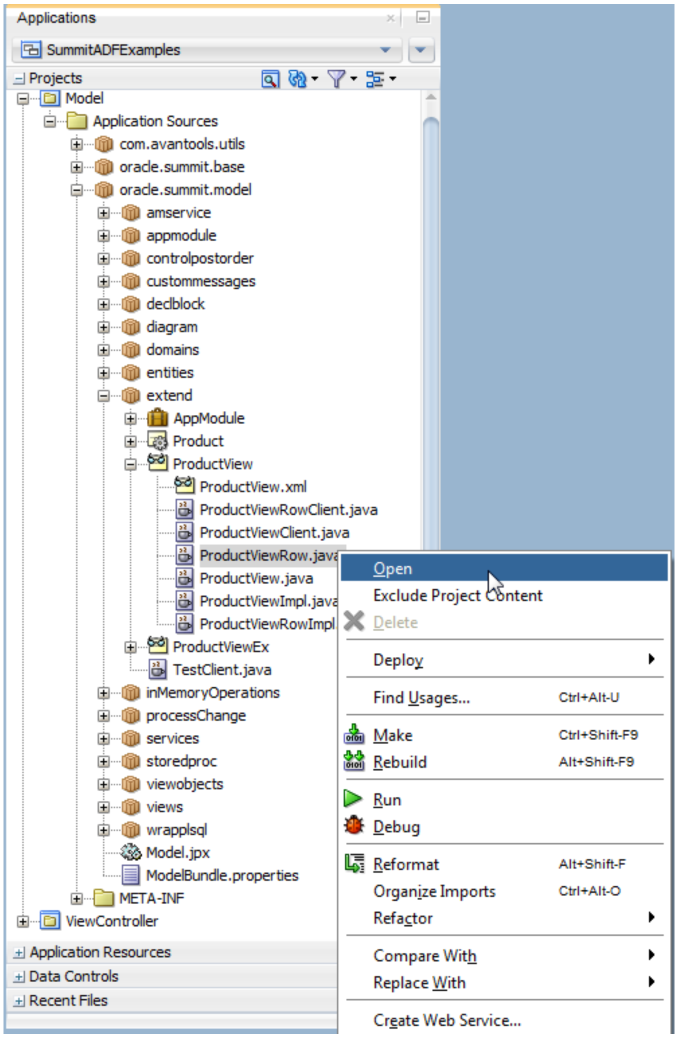
The Java generation options for the view object continue to be reflected on the Java page on subsequent visits to the view object overview editor. Just as with the XML definition file, JDeveloper keeps the generated code in your custom Java classes up to date with any changes you make in the editor. If later you decide you didn't require a custom Java file for any reason, unchecking the relevant options in the Java page causes the custom Java files to be removed.

Viewing and Navigating to Custom Java Files

As shown in Figure 11-2, when you've enabled generation of custom Java classes, they also appear under the node for the view object. When you need to see or work with the source code for a custom Java file, there are two ways to open the file in the source editor:

- Choose **Open** in the context menu, as shown in Figure 11-2
- With the Java file node selected in the Applications window, double-click a node in the Structure window

Figure 11-2 Viewing and Navigating to Custom Java Classes for a View Object



What You May Need to Know About Custom Classes

This section provides additional information to help you use custom Java classes.

About the Framework Base Classes for a View Object

When you use an "XML-only" view object, at runtime its functionality is provided by the default ADF Business Components implementation classes. Each custom Java class that gets generated will automatically extend the appropriate ADF Business Components base class so that your code inherits the default behavior and can easily add or customize it. A view object class will extend `ViewObjectImpl`, while the view row class will extend `ViewRowImpl` (both in the `oracle.jbo.server` package).

You Can Safely Add Code to the Custom Component File

Based perhaps on previous negative experiences, some developers are hesitant to add their own code to generated Java source files. Each custom Java source code file that JDeveloper creates and maintains for you includes the following comment at the top of the file to clarify that it is safe to add your own custom code to this file:

```
// ---------------------------------------------------------------------
// ---    File generated by ADF Business Components Design Time.
// ---    Custom code may be added to this class.
// ---    Warning: Do not modify method signatures of generated methods.
// ---------------------------------------------------------------------
```

JDeveloper does not blindly regenerate the file when you click the **OK** or **Apply** button in the component dialogs. Instead, it performs a smart update to the methods that it needs to maintain, leaving your own custom code intact.

Attribute Indexes and InvokeAccessor Generated Code

The view object is designed to function either in an XML-only mode or using a combination of an XML document and a custom Java class. Since attribute values are not stored in private member fields of a view row class, such a class is not present in the XML-only situation. Instead, attributes are defined as an AttributesEnum type, which specifies attribute names (and accessors for each attribute) based on the view object's XML document, in sequential order of the <ViewAttribute> tag, the association-related <ViewLinkAccessor> tag, and the <ViewAccessor> tag in that file. At runtime, the attribute values in an view row are stored in a structure that is managed by the base ViewRowImpl class, indexed by the attribute's numerical position in the view object's attribute list.

For the most part this private implementation detail is unimportant. However, when you enable a custom Java class for your view row, this implementation detail is related to some of the generated code that JDeveloper automatically maintains in your view row class, and you may want to understand what that code is used for. For example, in the custom Java class for the Users view row, the following example shows that each attribute, **view link accessor** attribute, or **view accessor** attribute has a corresponding generated AttributesEnum enum. JDeveloper defines enums instead of constants in order to prevent merge conflicts that could result when multiple developers add new attributes to the XML document.

```
public class RowImpl extends SummitViewRowImpl implements ProductViewRow {
/**
 * AttributesEnum: generated enum for identifying attributes and accessors.
 * Do not modify.
 */
public enum AttributesEnum {...}
  public static final int ID = AttributesEnum.Id.index();
  public static final int NAME = AttributesEnum.Name.index();
  public static final int SHORTDESC = AttributesEnum.ShortDesc.index();
  public static final int LONGTEXTID = AttributesEnum.LongtextId.index();
  public static final int IMAGEID = AttributesEnum.ImageId.index();
  public static final int SUGGESTEDWHLSLPRICE =
                             AttributesEnum.SuggestedWhlslPrice.index();
  public static final int WHLSLUNITS = AttributesEnum.WhlslUnits.index();
  public static final int SOMEVALUE = AttributesEnum.SomeValue.index();
  public static final int SFADSF = AttributesEnum.sfadsf.index();
  ...
```

You'll also notice that the automatically maintained, strongly typed getter and setter methods in the view row class use these attribute constants like this:

```
public Integer getId() {
  return (Integer) getAttributeInternal(ID); // <-- Attribute constant
}
public void setId(Integer value) {
  setAttributeInternal(ID, value);// <-- Attribute constant
}
```

The last two aspects of the automatically maintained code related to view row attribute constants defined by the `AttributesEnum` type are the `getAttrInvokeAccessor()` and `setAttrInvokeAccessor()` methods. These methods optimize the performance of attribute access by numerical index, which is how generic code in the `ViewRowImpl` base class typically accesses attribute values. An example of the `getAttrInvokeAccessor()` method looks like the following from the `ProductViewRowImpl.java` class. The companion `setAttrInvokeAccessor()` method looks similar.

```
protected Object getAttrInvokeAccessor(int index, AttributeDefImpl attrDef) throws Exception {
    if ((index >= AttributesEnum.firstIndex()) && (index < AttributesEnum.count())) {
        return AttributesEnum.staticValues()[index - AttributesEnum.firstIndex()].get(this);
    }
    return super.getAttrInvokeAccessor(index, attrDef);
}
```

The rules of thumb to remember about this generated attribute-related code are the following.

The Do's

- Add custom code if needed inside the strongly typed attribute getter and setter methods
- Use the view object overview editor to change the order or type of view object attributes

  JDeveloper will change the Java signature of getter and setter methods, as well as the related XML document for you.

The Don'ts

- Don't modify the list of enums in the generated `AttributesEnum` enum
- Don't modify the `getAttrInvokeAccessor()` and `setAttrInvokeAccessor()` methods

Avoid Creating Dependencies on Parent Application Module Types

Custom methods that you implement in the view object class or view object row class must not be dependent on the return type of an **application module**. At runtime, in specific cases, methods that execute with such a dependency may throw a `ClassCastException` because the returned application module does not match the expected type. It is therefore recommended that custom methods that you implement should not have code to get a specific application module implementation or view object implementation as shown below.

```
((MyAM)getRootApplicationModule()).getMyVO
```

Specifically, the above code fails with a `ClassCastException` in the following scenarios:

- When the ADF Business Components framework instantiates the view object in a different container application module than its defining container. While view objects are typically instantiated in the container application module that declares their view usage (XML definition), the ADF Business Components runtime does not guarantee that the containers associated with each application module will remain fixed. Thus, if you implement methods with dependencies on the parent application module type, your methods may not execute consistently.

- When you manually nest an application module under a **root application module**. In this case, the nested application modules share the same `Transaction` object and there is no guarantee that the expected application module type is returned with the above code.
- When the ADF Business Components framework implementation changes with releases. For example, in previous releases, the framework created an internal root application module in order to control declarative transactions that the application defined using **ADF task flows**.

Working Programmatically with Multiple Named View Criteria

ADF Business Components allows you to define multiple view criteria and apply combinations of them to a view object at runtime.

You can define *multiple* named **view criteria** in the overview editor for a view object and then selectively apply any combination of them to your view object at runtime as needed. For information about working with named view criteria at design time, see How to Create Named View Criteria Declaratively.

> **Note:**
>
> The example in this section refers to the `oracle.summit.model.multinamedviewcriteria` package in the `SummitADF_Examples` application workspace.

Applying One or More Named View Criteria

To apply one or more named view criteria, use the `setApplyViewCriteriaNames()` method. This method accepts a String array of the names of the criteria you want to apply. If you apply more than one named criteria, they are `AND`-ed together in the WHERE clause produced at runtime. New view criteria that you apply with the `setApplyViewCriteriaNames()` method will overwrite all previously applied view criteria. Alternatively, you can use the `setApplyViewCriteriaName()` method when you want to append a single view criteria to those that were previously applied.

When you need to apply more than one named view criteria, you can expose custom methods on the client interface of the view object to encapsulate applying combinations of the named view criteria. For example, the following example shows custom methods `showCustomersForSalesRep()`, `showCustomersForCreditRating()`, and `showCustomersForSalesRepForCreditRating()`, each of which uses the `setApplyViewCriteriaNames()` method to apply an appropriate combination of named view criteria. Once these methods are exposed on the view object's client interface, at runtime clients can invoke these methods as needed to change the information displayed by the view object.

```
// In CustomerViewImpl.java
   public void showCustomersForSalesRep() {
   // reset the view criteria
      setApplyViewCriteriaNames(null);
   // set the view criteria to show all the customers for the given sales rep
      setApplyViewCriteriaName("SalesRepIsCriteria");
      executeQuery();
   }

   public void showCustomersForCreditRating() {
      setApplyViewCriteriaNames(null);
      setApplyViewCriteriaName("CreditRatingIsCriteria");
      executeQuery();
   }

   public void showCustomersForSalesRepForCreditRating() {
   // reset the view criteria
      setApplyViewCriteriaNames(null);
   // apply both view criteria to show all the customers that match both
      setApplyViewCriteriaNames(new String[]{"SalesRepIsCriteria",
                                          "CreditRatingIsCriteria"});
      executeQuery();
   }
```

What You May Need to Know about Applying One or More Named View Criteria

You can also use the `appendViewCriteriaName()` method instead of `setApplyViewCriteriaNames()` to apply one or more named view criteria. This method works the same way as `setApplyViewCriteriaNames()` that accepts a String array of the names of the criteria you want to apply. If you apply more than one named criteria, they are `AND`-ed together in the WHERE clause produced at runtime. New view criteria that you apply with the `appendViewCriteriaName()` method will append the new view criteria to the already applied view criteria list of the view object. Alternatively, you can use the `appendViewCriteriaName()` method when you want to append a single view criteria to those that were previously applied.

The following code examples show the different new APIs that you can add to a view object.

- To append a given view criteria name to an existing list of applied view criteria.

  ```
   public void appendViewCriteriaName(String criteria);
  ```

- To append a list of given view criteria name to an existing list of view criteria.

  ```
  public void appendViewCriteriaName(String[] criterias);
  ```

- To append a given view criteria object to an existing list of criteria.

  ```
  public void appendViewCriteriaObject(ViewCriteria criteria);
  ```

- To append a list of given view criteria object to an existing list of applied view criteria.

  ```
  public void appendViewCriteriaObject(ViewCriteria[] criterias);
  ```

- To append a given view criteria expression to an existing list of applied view criteria.

  ```
     public void appendViewCriteriaExpression(String expr);
  ```

- To reset all the view criteria objects to their definition state.

  ```
   public void resetAllViewCriteria();
  ```

  This also removes any view criteria created at runtime.

- To reset a given view criteria object to its definition state.

  ```
  public void resetViewCriteriaName(String viewcriteria);
  ```

Removing All Applied Named View Criteria

To remove any currently applied named view criteria, use `setApplyViewCriteriaNames(null)`. For example, you could add the `showAllCustomers()` method in the following example to the `CustomersView` view object and expose it on the client interface. This would allow clients to return to an unfiltered view of the data when needed.

Do not remove any design time view criteria because the row level bind variable values may already be applied on the row set. To help ensure this, named view criteria that get defined for a view accessor in the design time, will be applied as "required" view criteria on the view object instance so that it does not get removed by the view criteria's life cycle methods.

```
// In CustomerViewImpl.java
public void showAllCustomers() {
  setApplyViewCriteriaNames(null);
  executeQuery();
}
```

> **Note:**
>
> The `setApplyViewCriterias(null)` removes all applied view criteria, but allows you to later reapply any combination of them. In contrast, the `clearViewCriterias()` method *deletes* all named view criteria. After calling `clearViewCriterias()` you would have to use `putViewCriteria()` again to define new named criteria before you could apply them.

Using the Named Criteria at Runtime

At runtime, your application can invoke different client methods on a single view object interface to return different filtered sets of data. The following example shows the interesting lines of a test client class that works with the `CustomerView` view object described above. The `showResults()` method is a helper method that iterates over the rows in the view object to display some attributes.

```
// In MultiNamedViewCriteriaTestClient.java
   CustomerView vo = (CustomerView) am.findViewObject("CustomerView");

// Show list of all rows in the CustomerView, without applying any view criteria.
   showResults(vo,"All Customers");

// Use  type safe set method to set the value of the bind variable used by the
// view criteria
   vo.setbv_SalesRepId(12);
   vo.showCustomersForSalesRep();
   showResults(vo, "All Customers with SalesRepId = 12");

// Set the sales rep id to 11 and show the results.
   vo.setbv_SalesRepId(11);
   vo.showCustomersForSalesRep();
   showResults(vo, "All Customers with SalesRepId = 11");

// use type safe method to set value of bind variable used by the view criteria
   vo.setbv_CreditRatingId(2);

// This method applies both view criteria to the VO so the results match
// both criteria, but we do not set the bind variable for sales rep as it
// was already set. The results will show all the customers with a sales
// rep of 11 and a credit rating of 2.
   vo.showCustomersForSalesRepForCreditRating();
   showResults(vo, "Customers with SalesRepId = 11 and CreditRating = 2");

// Now show all the customers again. The showAllCustomers method resets
// the view criteria.
   vo.showAllCustomers();
   showResults(vo, "All Customers");
```

Running `MultiNamedViewCriteriaTestClient.java` produces output as follows:

```
---All Customers ---

Unisports Sao Paulo [12, 1]
Simms Athletics Osaka [14, 4]
Delhi Sports New Delhi [14, 2]
...
---All Customers with SalesRepId = 12 ---

Unisports Sao Paulo [12, 1]
Futbol Sonora Nogales [12, 1]
Stuffz Sporting Goods Madison [12, 3]
...
---All Customers with SalesRepId = 11 ---

Womansport Seattle [11, 3]
Beisbol Si! San Pedro de Macon's [11, 1]
Big John's Sports Emporium San Francisco [11, 1]
```

```
Ojibway Retail Buffalo [11, 3]
...
---Customers with SalesRepId = 11 and CreditRating = 2 ---

Max Gear New York [11, 2]
MoreAndMoreStuffz Dallas [11, 2]
Schindler's Sports St Louis [11, 2]
...
---All Customers ---

Unisports Sao Paulo [12, 1]
Simms Athletics Osaka [14, 4]
Delhi Sports New Delhi [14, 2]
...
```

Performing In-Memory Sorting and Filtering of Row Sets

ADF Business Components allows you to configure view objects to search for data in-memory as opposed to from the database. In-memory searching and sorting is transaction-efficient and cost-effective.

View Object rows can be sorted by using the following APIs:

- `setSortBy`
- `setOrderByClause`
- `setOrderByorSortBy`

The `SortBy()` method is used to deal with in-memory sorting and the `OrderBy()` method is used to sort rows in the database layer.

By default a view object performs its query against the database to retrieve the rows in its resulting row set. However, you can also use view objects to perform in-memory searches and sorting to avoid unnecessary trips to the database. If transient attributes are passed to `setSortBy` API, then the rows will be sorted only in memory. This type of operation is ideal for sorting and filtering the new, as yet unposted rows of the view object row set; otherwise, unposted rows are added to the top of the row set. If any persistent attributes are passed to `setSortBy` API, then only for those persistent attributes a database order by clause is generated, so that rows are first sorted in database layer. If both transient and persistent attributes are passed , then for the persistent attributes a database order by clause is generated, database sorted rows are fetched into memory and then ADF BC will sort the fetched rows using both the transient and persistent attributes in the order specified.

> **Note:**
>
> The example in this section refers to to the `oracle.summit.model.inMemoryOperations` package in the `SummitADF_Examples` application workspace.

Understanding the View Object's SQL Mode

The view object's SQL mode controls the source used to retrieve rows to populate its row set. The `setQueryMode()` allows you to control which mode, or combination of modes, are used:

- `ViewObject.QUERY_MODE_SCAN_DATABASE_TABLES`

  This is the default mode that retrieves results from the database.
- `ViewObject.QUERY_MODE_SCAN_VIEW_ROWS`

  This mode uses rows already in the row set as the source, allowing you to progressively refine the row set's contents through in-memory filtering.
- `ViewObject.QUERY_MODE_SCAN_ENTITY_ROWS`

  This mode, valid only for entity-based view objects, uses the entity rows presently in the entity cache as the source to produce results based on the contents of the cache.

You can use the modes individually, or combine them using Java's logical `OR` operator (*X* `|` *Y*). For example, to create a view object that queries the entity cache for unposted new entity rows, as well as the database for existing rows, you could write code like:

```
setQueryMode(ViewObject.QUERY_MODE_SCAN_DATABASE_TABLES |
             ViewObject.QUERY_MODE_SCAN_ENTITY_ROWS)
```

If you combine the SQL modes, the view object automatically handles skipping of duplicate rows. In addition, there is an implied order to the results that are found:

1. Scan view rows (if specified)
2. Scan entity cache (if specified)
3. Scan database tables (if specified) by issuing a SQL query

If you call the `setQueryMode()` method to change the SQL mode, your new setting takes effect the next time you call the `executeQuery()` method.

Sorting View Object Rows In Memory

To sort the rows in a view object at runtime, use the `setSortBy()` method to control the sort order or you can set the view object property `PROP_ALWAYS_USE_SORT` to `true` to allow the system to handle in-memory sorting by default.

When you want to control the sort order, you pass a sort expression that looks like a SQL ORDER BY clause. However, instead of referencing the column names of the table, you use the view object's attribute names. For example, for a view object containing attributes named `CreditRatingId` and `ZipCode`, you could sort the view object first by `Customer` descending, then by `DaysOpen` by calling:

```
setSortBy("CreditRatingId desc, ZipCode");
```

Alternatively, you can use the *zero-based* attribute index position in the sorting clause like this:

```
setSortBy("3 desc, 2");
```

After calling the `setSortBy()` method, the rows will be sorted the next time you call the `executeQuery()` method. The view object translates this sorting clause into an appropriate format to use for ordering the rows depending on the SQL mode of the view object. If you use the default SQL mode, the `SortBy` clause is translated into an appropriate `ORDER BY` clause and used as part of the SQL statement sent to the database. If you use either of the in-memory SQL modes, then the `SortBy` by clause is translated into one or more `SortCriteria` objects for use in performing the in-memory sort.

> **Note:**
>
> While SQL ORDER BY expressions treat column names in a case-insensitive way, the attribute names in a `SortBy` expression are case-*sensitive*.

Combining setSortBy and setQueryMode for In-Memory Sorting

You can perform an in-memory sort on the rows produced by a read-only view object using the `setSortBy()` and `setQueryMode()` methods. The following example shows the interesting lines of code from the `TestClientSetSortBy` class that uses `setSortBy()` and `setQueryMode()` to perform an in-memory sort on the rows produced by a read-only view object `CustomersInTx`.

```
// In TestClientSetSortBy.java
am.getTransaction().executeCommand("ALTER SESSION SET SQL_TRACE TRUE");
ViewObject vo = am.findViewObject("CustomersInTx");
vo.executeQuery();
showRows(vo,"Initial database results");
vo.setSortBy("Name");
vo.setQueryMode(ViewObject.QUERY_MODE_SCAN_VIEW_ROWS);
vo.executeQuery();
showRows(vo,"After in-memory sorting by Customer Name ");
vo.setSortBy("CreditRatingId desc, ZipCode");
vo.executeQuery();
showRows(vo,"After in-memory sorting by CreditRating desc, and ZipCode");
```

Running the example produces the results:

```
--- Initial database results ---

36,Great Gear,1500 Barton Springs Rd,78704, 2, TX
37,Acme Outfitters,3500 Guadalupe St,78705, 3, TX
38,Athena's Closet,1209 Red River St,78701, 1, TX
39,BuyMyJunk,5610 E Mockingbird Ln,75206, 3, TX
...

--- After in-memory sorting by Customer Name  ---

37,Acme Outfitters,3500 Guadalupe St,78705, 3, TX
38,Athena's Closet,1209 Red River St,78701, 1, TX
43,Big Bad Sports,2920 Hillcroft St,77057, 1, TX
39,BuyMyJunk,5610 E Mockingbird Ln,75206, 3, TX
...

--- After in-memory sorting by CreditRating desc, and ZipCode ---
```

```
42,Field Importers,2111 Norfolk St,77098, 4, TX
39,BuyMyJunk,5610 E Mockingbird Ln,75206, 3, TX
37,Acme Outfitters,3500 Guadalupe St,78705, 3, TX
41,MoreAndMoreStuffz,3501 McKinney Ave,75204, 2, TX
...
```

The first line in the above test client containing the `executeCommand()` call issues the `ALTER SESSION SET SQL TRACE` command to enable SQL tracing for the current database session. This causes the Oracle database to log every SQL statement performed to a server-side trace file. It records information about the text of each SQL statement, including how many times the database parsed the statement and how many round-trips the client made to fetch batches of rows while retrieving the query result.

> **Note:**
>
> You might need a DBA to grant permission to the `Summit` account to perform the `ALTER SESSION` command to do the tracing of SQL output.

Once you've produced a trace file, you can use the `TKPROF` utility that comes with the database to format the file:

```
tkprof xe_ora_3916.trc trace.prf
```

For details about working with the `TKPROF` utility, see sections "Understanding SQL Trace and TKPROF" and "Using the SQL Trace Facility and TKPROF" in the "Performing Application Tracing" chapter of the *Oracle Database SQL Tuning Guide*.

This will produces a `trace.prf` file containing the interesting information shown in the following example about the SQL statement performed by the `CustomersInTx` view object. You can see that after initially querying six rows of data in a single execute and fetch from the database, the two subsequent sorts of those results did not cause any further executions. Since the code set the SQL mode to `ViewObject.QUERY_MODE_SCAN_VIEW_ROWS` the `setSortBy()` followed by the `executeQuery()` performed the sort in memory.

```
SELECT
    S_CUSTOMER.ID ID,
    S_CUSTOMER.NAME NAME,
    S_CUSTOMER.PHONE PHONE,
    S_CUSTOMER.ADDRESS ADDRESS,
    S_CUSTOMER.CITY CITY,
    S_CUSTOMER.STATE STATE,
    S_CUSTOMER.ZIP_CODE ZIP_CODE,
    S_CUSTOMER.CREDIT_RATING_ID CREDIT_RATING_ID
FROM
    S_CUSTOMER
WHERE
    S_CUSTOMER.STATE = 'TX'

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.02          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        9      0.00       0.00          0         14          0           8
```

```
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total       11      0.00       0.02          0         14          0           8
```

Simplified In-Memory Sorting

Should you not require control over the way that rows are sorted in memory, the view object interface provides the PROP_ALWAYS_USE_SORT property that enforces sorting when you change the view object query mode to use in-memory rows as the source. You can use this property when a view object has a small row set and the client needs to have in-memory sorting performed. Set the view object property to specify default in-memory sorting after you set the query mode:

```
vo.setQueryMode(ViewObject.QUERY_MODE_SCAN_ENTITY_ROWS);
vo.setProperty(ViewObject.PROP_ALWAYS_USE_SORT, "true");
```

Extensibility Points for In-Memory Sorting

Should you need to customize the way that rows are sorted in memory, you have the following means at your disposal:

- You can override the method:

  ```
  public void sortRows(Row[] rows)
  ```

  This method performs the actual in-memory sorting of rows. By overriding this method you can plug in an alternative sorting approach if needed.
- You can override the method:

  ```
  public Comparator getRowComparator()
  ```

  The default implementation of this method returns an oracle.jbo.RowComparator. RowComparator invokes the compareTo() method to compare two data values. These methods/objects can be overridden to provide custom compare routines.

Performing In-Memory Filtering with View Criteria

To filter the contents of a row set using ViewCriteria, you can call:

- applyViewCriteria() or setApplyViewCriteriaNames() followed by executeQuery() to produce a new, filtered row set.
- findByViewCriteria() to retrieve a new row set to process programmatically without changing the contents of the original row set.

Both of these approaches can be used against the database or to perform in-memory filtering, or both, depending on the view criteria mode. You set the criteria mode using the setCriteriaMode() method on the ViewCriteria object, to which you can pass either of the following integer flags, or the logical OR of both:

- ViewCriteria.CRITERIA_MODE_QUERY
- ViewCriteria.CRITERIA_MODE_CACHE

When used for in-memory filtering with view criteria, the operators supported are shown in Table 11-1. You can group subexpressions with parenthesis and use the AND and OR operators between subexpressions.

Table 11-1   SQL Operators Supported By In-Memory Filtering with View Criteria

Operator	Operation
=, >, <, <=, >=, <>, LIKE, BETWEEN, IN	Comparison
NOT	Logical negation
AND	Conjunction
OR	Disjunction

The following example shows the interesting lines from a TestClientFindByViewCriteria class that uses the two features described above both against the database and in-memory. It uses a CustomerList view object instance and performs the following basic steps:

1. Queries customers from the database with a last name starting with a 'C', producing the output:

   ```
   --- Initial database results with applied view criteria ---
   John Chen
   Emerson Clabe
   Karen Colmenares
   ```

2. Subsets the results from step 1 in memory to only those with a first name starting with 'J'. It does this by adding a second view criteria row to the view criteria and setting the conjunction to use "AND". This produces the output:

   ```
   --- After augmenting view criteria and applying in-memory ---
   John Chen
   ```

3. Sets the conjunction back to OR and reapplies the criteria to the database to query customers with last name like 'J%' or first name like 'C%'. This produces the output:

   ```
   --- After changing view criteria and applying to database again ---
   John Chen
   Jose Manuel Urman
   Emerson Clabe
   Karen Colmenares
   Jennifer Whalen
   ```

4. Defines a new criteria to find customers in-memory with first or last name that contain a letter 'o'

5. Uses findByViewCriteria() to produce new a row set (instead of subsetting), producing the output:

   ```
   --- Rows returned from in-memory findByViewCriteria ---
   John Chen
   Jose Manuel Urman
   Emerson Clabe
   Karen Colmenares
   ```

6. Shows that original row set hasn't changed when findByViewCriteria() was used, producing the output:

   ```
   --- Note findByViewCriteria didn't change rows in the view ---
   John Chen
   Jose Manuel Urman
   Emerson Clabe
   ```

```
    Karen Colmenares
    Jennifer Whalen
```

```
// In TestClientFindByViewCriteria.java
ViewObject vo = am.findViewObject("CustomerList");
// 1. Show customers with a last name starting with a 'M'
ViewCriteria vc = vo.createViewCriteria();
ViewCriteriaRow vcr1 = vc.createViewCriteriaRow();
vcr1.setAttribute("LastName","LIKE 'M%'");
vo.applyViewCriteria(vc);
vo.executeQuery();
vc.add(vcr1);
vo.executeQuery();
showRows(vo, "Initial database results with applied view criteria");
// 2. Subset results in memory to those with first name starting with 'S'
vo.setQueryMode(ViewObject.QUERY_MODE_SCAN_VIEW_ROWS);
ViewCriteriaRow vcr2 = vc.createViewCriteriaRow();
vcr2.setAttribute("FirstName","LIKE 'S%'");
vcr2.setConjunction(ViewCriteriaRow.VCROW_CONJ_AND);
vc.setCriteriaMode(ViewCriteria.CRITERIA_MODE_CACHE);
vc.add(vcr2);
vo.executeQuery();
showRows(vo,"After augmenting view criteria and applying in-memory");
// 3. Set conjuction back to OR and reapply to database query to find
// customers with last name like 'H%' or first name like 'S%'
vc.setCriteriaMode(ViewCriteria.CRITERIA_MODE_QUERY);
vo.setQueryMode(ViewObject.QUERY_MODE_SCAN_DATABASE_TABLES);
vcr2.setConjunction(ViewCriteriaRow.VCROW_CONJ_OR);
vo.executeQuery();
showRows(vo,"After changing view criteria and applying to database again");
// 4. Define new critera to find customers with first or last name like '%o%'
ViewCriteria nameContainsO = vo.createViewCriteria();
ViewCriteriaRow lastContainsO = nameContainsO.createViewCriteriaRow();
lastContainsO.setAttribute("LastName","LIKE '%o%'");
ViewCriteriaRow firstContainsO = nameContainsO.createViewCriteriaRow();
firstContainsO.setAttribute("FirstName","LIKE '%o%'");
nameContainsO.add(firstContainsO);
nameContainsO.add(lastContainsO);
// 5. Use findByViewCriteria() to produce new rowset instead of subsetting
nameContainsO.setCriteriaMode(ViewCriteria.CRITERIA_MODE_CACHE);
RowSet rs = (RowSet)vo.findByViewCriteria(nameContainsO,
                         -1,ViewObject.QUERY_MODE_SCAN_VIEW_ROWS);
showRows(rs,"Rows returned from in-memory findByViewCriteria");
// 6. Show that original rowset hasn't changed
showRows(vo,"Note findByViewCriteria didn't change rows in the view");
```

Performing In-Memory Filtering with RowMatch

The `RowMatch` object provides an even more convenient way to express in-memory filtering conditions. You create a `RowMatch` object by passing a query predicate expression to the constructor like this:

```
RowMatch rm =
 new RowMatch("LastName = 'Popp' or (FirstName like 'A%' and LastName like 'K%')"
);
```

As you do with the `SortBy` clause, you phrase the `RowMatch` expression in terms of the view object attribute names, using the supported operators shown in Table 11-2. You can group subexpressions with parenthesis and use the `AND` and `OR` operators between subexpressions.

Table 11-2 SQL Operators Supported By In-Memory Filtering with RowMatch

Operator	Operation
=, >, <, <=, >=, <>, LIKE, BETWEEN, IN	Comparison
NOT	Logical negation Note that logical negation operations NOT IN are not supported by the RowMatch expression. To negate the IN operator, use this construction instead (note the use of brackets): NOT (EmpID IN ('VP','PU'))
AND	Conjunction
OR	Disjunction

You can also use a limited set of SQL functions in the RowMatch expression, as shown in Table 11-3.

Table 11-3 SQL Functions Supported By In-Memory Filtering with RowMatch

Operator	Operation
UPPER	Converts all letters in a string to uppercase.
TO_CHAR	Converts a number or date to a string.
TO_DATE	Converts a character string to a date format.
TO_TIMESTAMP	Converts a string to timestamp.

> **Note:**
>
> While SQL query predicates treat column names in a case-insensitive way, the attribute names in a RowMatch expression are case-*sensitive*.

Applying a RowMatch to a View Object

To apply a RowMatch to your view object, call the setRowMatch() method. In contrast to a ViewCriteria, the RowMatch is only used for *in-memory* filtering, so there is no "match mode" to set. You can use a RowMatch on view objects in any supported SQL mode, and you will see the results of applying it the next time you call the executeQuery() method.

When you apply a RowMatch to a view object, the RowMatch expression can reference the view object's named bind variables using the same :VarName notation that you would use in a SQL statement. For example, if a view object had a named bind variable named StatusCode, you could apply a RowMatch to it with an expression like:

```
Status = :StatusCode or :StatusCode = '%'
```

The following example shows the interesting lines of a `TestClientRowMatch` class that illustrate the `RowMatch` in action. The CustomerList view object used in the example has a transient `Boolean` attribute named `Selected`. The code performs the following basic steps:

1. Queries the full customer list, producing the output:

   ```
   --- Initial database results ---
   Neena Kochhar [null]
   Lex De Haan [null]
   Nancy Greenberg [null]
   :
   ```

2. Marks odd-numbered rows selected by setting the `Selected` attribute of odd rows to `Boolean.TRUE`, producing the output:

   ```
   --- After marking odd rows selected ---
   Neena Kochhar [null]
   Lex De Haan [true]
   Nancy Greenberg [null]
   Daniel Faviet [true]
   John Chen [null]
   Ismael Sciarra [true]
   :
   ```

3. Uses a `RowMatch` to subset the row set to contain only the select rows, that is, those with `Selected = true`. This produces the output:

   ```
   --- After in-memory filtering on only selected rows ---
   Lex De Haan [true]
   Daniel Faviet [true]
   Ismael Sciarra [true]
   Luis Popp [true]
   :
   ```

4. Further subsets the row set using a more complicated `RowMatch` expression, producing the output:

   ```
   --- After in-memory filtering with more complex expression ---
   Lex De Haan [true]
   Luis Popp [true]
   ```

```
// In TestClientRowMatch.java
// 1. Query the full customer list
ViewObject vo = am.findViewObject("CustomerList");
vo.executeQuery();
showRows(vo,"Initial database results");
// 2. Mark odd-numbered rows selected by setting Selected = Boolean.TRUE
markOddRowsAsSelected(vo);
showRows(vo,"After marking odd rows selected");
// 3. Use a RowMatch to subset row set to only those with Selected = true
RowMatch rm = new RowMatch("Selected = true");
vo.setRowMatch(rm);
// Note: Only need to set SQL mode when not defined at design time
vo.setQueryMode(ViewObject.QUERY_MODE_SCAN_VIEW_ROWS);
vo.executeQuery();
showRows(vo, "After in-memory filtering on only selected rows");
// 5. Further subset rowset using more complicated RowMatch expression
rm = new RowMatch("LastName = 'Popp' "+
                  "or (FirstName like 'A%' and LastName like 'K%')");
vo.setRowMatch(rm);
vo.executeQuery();
showRows(vo,"After in-memory filtering with more complex expression");
```

```
// 5. Remove RowMatch, set query mode back to database, requery to see full list
vo.setRowMatch(null);
vo.setQueryMode(ViewObject.QUERY_MODE_SCAN_DATABASE_TABLES);
vo.executeQuery();
showRows(vo,"After requerying to see a full list again");
```

Using RowMatch to Test an Individual Row

In addition to using a RowMatch to filter a row set, you can also use its rowQualifies() method to test whether any individual row matches the criteria it encapsulates. For example:

```
RowMatch rowMatch = new RowMatch("CountryId = 'US'");
if (rowMatch.rowQualifies(row)) {
  System.out.println("Customer is from the United States ");
}
```

How a RowMatch Affects Rows Fetched from the Database

Once you apply a RowMatch, if the view object's SQL mode is set to retrieve rows from the database, when you call executeQuery() the RowMatch is applied to rows as they are fetched. If a fetched row does not qualify, it is not added to the row set.

Unlike a SQL WHERE clause, a RowMatch can evaluate expressions involving transient view object attributes and not-yet-posted attribute values. This can be useful to filter queried rows based on RowMatch expressions involving transient view row attributes whose values are calculated in Java. This interesting aspect should be used with care, however, if your application needs to process a large row set. Oracle recommends using database-level filtering to retrieve the smallest-possible row set first, and then using RowMatch as appropriate to subset that list in memory.

Reading and Writing XML

View objects in ADF Business Components support reading and writing of data in XML documents. This allows view objects to perform CRUD operations based on XML documents.

The Extensible Markup Language (XML) standard from the Worldwide Web Consortium (W3C) defines a language-neutral approach for electronic data exchange. Its rigorous set of rules enables the structure inherent in data to be easily encoded and unambiguously interpreted using human-readable text documents.

View objects support the ability to write these XML documents based on their queried data. View objects also support the ability to read XML documents in order to apply changes to data including inserts, updates, and deletes. When you've introduced a **view link**, this XML capability supports reading and writing multi-level nested information for master-detail hierarchies of any complexity. While the XML produced and consumed by view objects follows a canonical format, you can combine the view object's XML features with XML Stylesheet Language Transformations (XSLT) to easily convert between this canonical XML format and any format you need to work with.

Note:

The example in this section refers to the `oracle.summit.model.readandwrite` package in the `SummitADF_Examples` application workspace.

How to Produce XML for Queried Data

To produce XML from a view object, use the `writeXML()` method. If offers two ways to control the XML produced:

1. For precise control over the XML produced, you can specify a view object attribute map indicating which attributes should appear, including which view link accessor attributes should be accessed for nested, detail information:

   ```
   Node writeXML(long options, HashMap voAttrMap)
   ```

2. To producing XML that includes all attributes, you can simply specify a depth level that indicates how many levels of view link accessor attributes should be traversed to produce the result:

   ```
   Node writeXML(int depthCount, long options)
   ```

The `options` parameter is an integer flag field that can be set to one of the following bit flags:

- `XMLInterface.XML_OPT_ALL_ROWS`

  Includes all rows in the view object's row set in the XML.

- `XMLInterface.XML_OPT_LIMIT_RANGE`

  Includes only the rows in the current range in the XML.

Using the logical OR operation, you can combine either of the above flags with the `XMLInterface.XML_OPT_ASSOC_CONSISTENT` flag when you want to include new, unposted rows in the current transaction in the XML output.

Both versions of the `writeXML()` method accept an optional third argument which is an XSLT stylesheet that, if supplied, is used to transform the XML output before returning it.

Additionally, both versions of the `writeXML()` method allow you to set the argument `depthCount=ignore` to indicate to ignore the depth count and just render what is in the data model based on the specified `options` parameter flags.

What Happens When You Produce XML

When you produce XML using `writeXML()`, the view object begins by creating a wrapping XML element whose default name matches the name of the view object definition. For example, for a `CustomersView` view object, the XML produced will be wrapped in an outermost CustomersView tag.

Then, it converts the attribute data for the appropriate rows into XML elements. By default, each row's data is wrapped in an row element whose name is the name of the view object with the `Row` suffix. For example, each row of data from a view object named `CustomersView` is wrapped in an `CustomersViewRow` element. The elements representing the attribute data for each row appear as nested children inside this row element.

If any of the attributes is a view link accessor attribute, and if the parameters passed to `writeXML()` enable it, the view object will include the data for the detail row set returned by the view link accessor. This nested data is wrapped by an element whose name is determined by the name of the view link accessor attribute. The return value of the `writeXML()` method is an object that implements the standard W3C `Node` interface, representing the root element of the generated XML.

> **Note:**
>
> The `writeXML()` method uses view link accessor attributes to programmatically access detail collections. It does not require adding view link instances in the data model.

For example, to produce an XML element for all rows of a `CustomersView` view object instance, and following view link accessors as many levels deep as exists, the following example shows the code required.

```
ViewObject vo = am.findViewObject("CustomersView");
printXML(vo.writeXML(-1,XMLInterface.XML_OPT_ALL_ROWS));
```

The `CustomersView` view object is linked to a `Orders` view object showing the orders created by that person. In turn, the `Orders` view object is linked to a `OrderItems` view object providing details on the items ordered by customers. Running the code in the following example produces the XML shown in the following example, reflecting the nested structure defined by the view links.

```
 ...
   <CustomersViewRow>
      <Id>211</Id>
      <Name>Kuhn's Sports</Name>
      <Phone>42-111292</Phone>
      <Address>7 Modrany</Address>
      <City>Prague</City>
      <CountryId>11</CountryId>
      <CreditRatingId>2</CreditRatingId>
      <SalesRepId>15</SalesRepId>
      <OrdersView>
         <OrdersViewRow>
            <Id>107</Id>
            <CustomerId>211</CustomerId>
            <DateOrdered>2012-12-13</DateOrdered>
            <DateShipped>2012-12-14</DateShipped>
            <SalesRepId>15</SalesRepId>
            <Total>142171</Total>
            <PaymentTypeId>2</PaymentTypeId>
            <PaymentOptionId>1082</PaymentOptionId>
            <OrderFilled>Y</OrderFilled>
         </OrdersViewRow>
         <OrdersViewRow>
            <Id>183</Id>
            <CustomerId>211</CustomerId>
            <DateOrdered>2013-02-24</DateOrdered>
            <DateShipped>2013-02-27</DateShipped>
            <SalesRepId>13</SalesRepId>
            <Total>3742</Total>
            <PaymentTypeId>1</PaymentTypeId>
            <OrderFilled>Y</OrderFilled>
         </OrdersViewRow>
         <OrdersViewRow>
            <Id>184</Id>
            <CustomerId>211</CustomerId>
            <DateOrdered>2012-11-19</DateOrdered>
            <DateShipped>2012-11-22</DateShipped>
            <SalesRepId>12</SalesRepId>
            <Total>987</Total>
            <PaymentTypeId>1</PaymentTypeId>
            <OrderFilled>Y</OrderFilled>
         </OrdersViewRow>
         <OrdersViewRow>
            <Id>185</Id>
            <CustomerId>211</CustomerId>
            <DateOrdered>2012-08-08</DateOrdered>
            <DateShipped>2012-08-12</DateShipped>
            <SalesRepId>12</SalesRepId>
            <Total>2525.25</Total>
            <PaymentTypeId>1</PaymentTypeId>
            <OrderFilled>Y</OrderFilled>
         </OrdersViewRow>
         <OrdersViewRow>
            <Id>186</Id>
            <CustomerId>211</CustomerId>
            <DateOrdered>2012-04-26</DateOrdered>
            <DateShipped>2012-04-30</DateShipped>
            <SalesRepId>12</SalesRepId>
            <Total>307.25</Total>
            <PaymentTypeId>1</PaymentTypeId>
            <OrderFilled>Y</OrderFilled>
         </OrdersViewRow>
         <OrdersViewRow>
            <Id>187</Id>
            <CustomerId>211</CustomerId>
            <DateOrdered>2012-12-23</DateOrdered>
            <DateShipped>2012-12-30</DateShipped>
            <SalesRepId>13</SalesRepId>
            <Total>3872.9</Total>
            <PaymentTypeId>1</PaymentTypeId>
            <OrderFilled>Y</OrderFilled>
         </OrdersViewRow>
         <OrdersViewRow>
            <Id>188</Id>
            <CustomerId>211</CustomerId>
            <DateOrdered>2013-01-14</DateOrdered>
            <DateShipped>2013-01-15</DateShipped>
            <SalesRepId>12</SalesRepId>
            <Total>12492</Total>
            <PaymentTypeId>1</PaymentTypeId>
            <OrderFilled>Y</OrderFilled>
         </OrdersViewRow>
         ...
      </OrdersView>
   </CustomersViewRow>
...
```

What You May Need to Know About Reading and Writing XML

This section provides additional information to help you work with XML.

Controlling XML Element Names

You can use the Properties window to change the default XML element names used in the view object's canonical XML format by setting several properties. To accomplish this, open the overview editor for the view object, then:

- Select the attribute on the Attributes page and in the Properties window, select the Custom Properties navigation tab and set the custom *attribute*-level property named **Xml Element** to a value *SomeOtherName* to change the XML element name used for that attribute to <SomeOtherName>

  For example, the Email attribute in the CustomersView view object defines this property to change the XML element described in What Happens When You Produce XML to be <EmailAddress> instead of <Email>.

- Select the General navigation tab in the Properties window and set the custom *view object*-level property named **Xml Row Element** to a value *SomeOtherRowName* to change the XML element name used for that view object to <SomeOtherRowName>.

  For example, the CustomersView view object defines this property to change the XML element name for the rows described in What Happens When You Produce XML to be <Customer> instead of <CustomersViewRow>.

- To change the name of the element names that wrapper nested row set data from view link attribute accessors, use the View Link Properties dialog. To open the dialog, in the view link overview editor, click the **Edit accessors** icon in the Accessors section of the Relationship page. Enter the desired name of the view link accessor attribute in the **Accessor Name** field.

Controlling Element Suppression for Null-Valued Attributes

By default, if a view row attribute is null, then its corresponding element is omitted from the generated XML. Select the attribute on the Attributes page of the overview editor and in the Properties window, select the Custom Properties navigation tab and set the custom *attribute*-level property named **Xml Explicit Null** to any value (e.g. "true" or "yes") to cause an element to be included for the attribute if its value is null. For example, if an attribute named AssignedDate has this property set, then a row containing a null assigned date will contain a corresponding AssignedDate null="true"/ element. If you want this behavior for all attributes of a view object, you can define the **Xml Explicit Null** custom property at the view object level as a shortcut for defining it on each attribute.

Printing or Searching the Generated XML Using XPath

Two of the most common things you might want to do with the XML Node object returned from writeXML() are:

1. Printing the node to its serialized text representation — to send across the network or save in a file, for example
2. Searching the generated XML using W3C XPath expressions

Unfortunately, the standard W3C Document Object Model (DOM) API does not include methods for doing *either* of these useful operations. But there is hope. Since ADF Business Components uses the Oracle XML parser's implementation of the DOM, you can cast the Node return value from writeXML() to the Oracle specific classes XMLNode or XMLElement (in the oracle.xml.parser.v2 package) to access additional useful functionality like:

- Printing the XML element to its serialized text format using the print() method
- Searching the XML element in memory with XPath expressions using the selectNodes() method
- Finding the value of an XPath expression related to the XML element using the valueOf() method.

The following example shows the printXML() method in the TestClientWriteXML. It casts the Node parameter to an XMLNode and calls the print() method to dump the XML to the console.

```
// In TestClientWriteXML.java
private static void printXML(Node n) throws IOException {
  ((XMLNode)n).print(System.out);
}
```

Using the Attribute Map For Fine Control Over Generated XML

When you need fine control over which attributes appear in the generated XML, use the version of the writeXML() method that accepts a HashMap. The following example shows the interesting lines from a TestClientWriteXML class that use this technique. After creating the HashMap, you put String[]-valued entries into it containing the names of the attributes you want to include in the XML, keyed by the fully qualified name of the view definition those attributes belong to. The example includes the Name, City, and OrdersView attributes from the CustomersView view object, and the Id, DateOrdered, and Total attributes from the OrdersView view object.

Note:

For upward compatibility reasons with earlier versions of ADF Business Components the HashMap expected by the writeXML() method is the one in the com.sun.java.util.collections package.

While processing the view rows for a given view object instance:

- If an entry exists in the attribute map with a key matching the fully qualified view definition name for that view object, then only the attributes named in the corresponding String array are included in the XML.

  Furthermore, if the string array includes the name of a view link accessor attribute, then the nested contents of its detail row set are included in the XML. If a view link accessor attribute name does not appear in the string array, then the contents of its detail row set are not included.
- If no such entry exists in the map, then *all* attributes for that row are included in the XML.

```
HashMap viewDefMap = new HashMap();
viewDefMap.put("oracle.summit.model.readandwritexml.queries.CustomersView",
        new String[]{"Name","City",
                     "OrdersView" /* View link accessor attribute */
                     });
viewDefMap.put("oracle.summit.model.readandwritexml.queries.OrdersView",
        new String[]{"Id","DateOrdered","Total"});
printXML(vo.writeXML(XMLInterface.XML_OPT_ALL_ROWS,viewDefMap));
```

Running the above example produces the following XML, including only the exact attributes and view link accessors indicated by the supplied attribute map.

```
   <CustomersViewRow>
      <Id>211</Id>
      <Name>Kuhn's Sports</Name>
      <City>Prague</City>
      <OrdersView>
         <OrdersViewRow>
            <Id>107</Id>
            <DateOrdered>2012-12-13</DateOrdered>
            <Total>142171</Total>
         </OrdersViewRow>
         <OrdersViewRow>
            <Id>183</Id>
            <DateOrdered>2013-02-24</DateOrdered>
            <Total>3742</Total>
         </OrdersViewRow>
         <OrdersViewRow>
            <Id>184</Id>
            <DateOrdered>2012-11-19</DateOrdered>
            <Total>987</Total>
         </OrdersViewRow>
         <OrdersViewRow>
            <Id>185</Id>
            <DateOrdered>2012-08-08</DateOrdered>
            <Total>2525.25</Total>
         </OrdersViewRow>
         <OrdersViewRow>
            <Id>186</Id>
            <DateOrdered>2012-04-26</DateOrdered>
            <Total>307.25</Total>
         </OrdersViewRow>
         <OrdersViewRow>
            <Id>187</Id>
            <DateOrdered>2012-12-23</DateOrdered>
            <Total>3872.9</Total>
         </OrdersViewRow>
         <OrdersViewRow>
            <Id>188</Id>
            <DateOrdered>2013-01-14</DateOrdered>
            <Total>12492</Total>
         </OrdersViewRow>
         ...
      </OrdersView>
   </CustomersViewRow>
...
```

Use the Attribute Map Approach with Bi-Directional View Links

If your view objects are related through a view link that you have configured to be bi-directional, then you must use the `writeXML()` approach that uses the attribute map. If you were to use the `writeXML()` approach in the presence of bi-directional view links and were to supply a maximum depth of `-1` to include all levels of view links that exist, the `writeXML()` method will go into an infinite loop as it follows the bi-directional view links back and forth, generating deeply nested XML containing duplicate data until it runs out of memory. Use `writeXML()` with an attribute map instead in this situation. Only by using this approach can you control which view link accessors are included in the XML and which are not to avoid infinite recursion while generating the XML.

Transforming Generated XML Using an XSLT Stylesheet

When the canonical XML format produced by `writeXML()` does not meet your needs, you can supply an XSLT stylesheet as an optional argument. It will produce the XML as it would normally, but then transform that result using the supplied stylesheet before returning the final XML to the caller.

Consider the XSLT stylesheet shown in the following example. It is a simple transformation with a single template that matches the root element of the generated XML described in Using the Attribute Map For Fine Control Over Generated XML to create a new `CustomerNames` element in the result. The template uses the `xsl:for-each` instruction to process all `CustomersView` elements. For each `CustomersView` element that qualifies, it creates a `CustomerNames` element in the result whose `Contact` attribute is populated from the value of the `Name` child element of the `CustomersView`.

```
<?xml version="1.0" encoding="windows-1252" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <CustomerNames>
      <xsl:for-each
           select="/CustomersView/CustomersViewRow)">
        <xsl:sort select="Name"/>
        <Customer Contact="{Name}"/>
      </xsl:for-each>
    </CustomerNames>
  </xsl:template>
</xsl:stylesheet>
```

The following example shows the interesting lines from a `TestClientWriteXML` class that put this XSLT stylesheet into action when calling `writeXML()`.

```
// In TestClientWriteXML.java
XSLStylesheet xsl = getXSLStylesheet();
printXML(vo.writeXML(XMLInterface.XML_OPT_ALL_ROWS,viewDefMap,xsl));
```

Running the code in the above example produces the transformed XML shown here:

```
<CustomerNames>
   <Customer Contact="ABC Company"/>
   <Customer Contact="Acme Outfitters"/>
   <Customer Contact="Acme Sporting Goods"/>
   <Customer Contact="All Baseball"/>
   ...
   <Customer Contact="The Sports Emporium"/>
   <Customer Contact="Unisports"/>
   <Customer Contact="Value Valley"/>
   <Customer Contact="Wally's Mart"/>
   <Customer Contact="Wally's Weights"/>
   <Customer Contact="Westwind Sports"/>
   <Customer Contact="Womansport"/>
   <Customer Contact="Your Choice Sporting Goods"/>
   <Customer Contact="Z-Mart"/>
   <Customer Contact="Zibbers"/>
   <Customer Contact="Zip City"/>
</CustomerNames>
```

The getXSLStylesheet() helper method shown in the following example is also interesting to study since it illustrates how to read a resource like an XSLT stylesheet from the classpath at runtime. The code expects the Example.xsl stylesheet to be in the same directory as the TestClientWriteXML class. By referencing the Class object for the TestClientWriteXML class using the .class operator, the code uses the getResource() method to get a URL to the resource. Then, it passes the URL to the newXSLStylesheet() method of the XSLProcessor class to create a new XSLStylesheet object to return. That object represents the compiled version of the XSLT stylesheet read in from the *.xslfile.

```
private static XSLStylesheet getXSLStylesheet()
        throws XMLParseException, SAXException,IOException,XSLException {
  String xslurl = "Example.xsl";
  URL xslURL = TestClientWriteXML.class.getResource(xslurl);
  XSLProcessor xslProc = new XSLProcessor();
  return xslProc.newXSLStylesheet(xslURL);
}
```

Note:

When working with resources like XSLT stylesheets that you want to be included in the output directory along with your compiled Java classes and XML metadata, you can use the **Compiler** page of the **Project Properties** dialog to update the **Copy File Types to Output Directory** field to include .xsl in the semicolon-separated list.

Generating XML for a Single Row

In addition to calling writeXML() on a view object, you can call the same method with the same parameters and options on any Row as well. If the Row object on which you call writeXML() is a entity row, you can bitwise-OR the additional XMLInterface.XML_OPT_CHANGES_ONLY flag if you only want the changed entity attributes to appear in the XML.

How to Consume XML Documents to Apply Changes

To have a view object consume an XML document to process inserts, updates, and deletes, use the readXML() method:

```
void readXML(Element elem, int depthcount)
```

The canonical format expected by readXML() is the same as what would be produced by a call to the writeXML() method on the same view object. If the XML document to process does not correspond to this canonical format, you can supply an XSLT stylesheet as an optional third argument to `readXML()` to transform the incoming XML document *into* the canonical format before it is read for processing.

What Happens When You Consume XML Documents

When a view object consumes an XML document in canonical format, it processes the document to recognize row elements, their attribute element children, and any nested elements representing view link accessor attributes. It processes the document recursively to a maximum level indicated by the `depthcount` parameter. Passing `-1` for the `depthcount` to request that it process all levels of the XML document.

How ViewObject.readXML() Processes an XML Document

For each row element it recognizes, the `readXML()` method does the following:

- Identifies the related view object to process the row.
- Reads the children attribute elements to get the values of the primary key attributes for the row.
- Performs a `findByKey()` using the primary key attributes to detect whether the row already exists or not.
- If the row exists:
  - If the row element contains the marker attribute `bc4j-action="remove"`, then the existing row is deleted.
  - Otherwise, the row's attributes are updated using the values in any attribute element children of the current row element in the XML
- If the row does not exist, then a new row is created, inserted into the view object's row set. Its attributes are populated using the values in any attribute element children of the current row element in the XML.

Using readXML() to Processes XML for a Single Row

The same `readXML()` method is also supported on any `Row` object. The canonical XML format it expects is the same format produced by a call to `writeXML()` on the same row. You can invoke `readXML()` method on a row to:

- Update its attribute values from XML
- Remove the row, if the `bc4j-action="remove"` marker attribute is present on the corresponding row element.
- Insert, update, or delete any nested rows via view link accessors

Consider the XML document shown in the following example. It is in the canonical format expected by a single row in the `CustomersView` view object. Nested inside the root `CustomersViewRow` element, the `City` attribute represents the customer's city. The nested `OrdersView` element corresponds to the `OrdersView` view link accessor attribute and contains three `OrdersViewRow` elements. Each of these includes `Id` elements representing the primary key of a `OrdersView` row.

```
<CustomersViewRow>
   <Id>16</Id>
   <!-- This will update Customer's ConfirmedEmail attribute -->
   <City>Houston</City>
   <OrdersView>
      <!-- This will be an update since it does exist -->
      <OrdersViewRow>
         <Id>1018</Id>
         <DateShipped>2013-03-13</DateShipped>
      </OrdersViewRow>
      <!-- This will be an insert since it doesn't exist -->
      <OrdersViewRow>
         <Id>9999</Id>
         <CustomerId>16</CustomerId>
         <Total>9999.00</Total>
      </OrdersViewRow>
      <!-- This will be deleted -->
      <OrdersViewRow bc4j-action="remove">
         <Id>1008</Id>
      </OrdersViewRow>
   </OrdersView>
</CustomersViewRow>
```

The following example shows the interesting lines of code from a `TestClientReadXML` class that applies this XML datagram to a particular row in the `CustomersView` view object. `TestClientReadXML` class performs the following basic steps:

1. Finds a target row by key (e.g. for customer "The Sports Emporium").
2. Shows the XML produced for the row before changes are applied.
3. Obtains the parsed XML document with changes to apply using a helper method.
4. Reads the XML document to apply changes to the row.
5. Shows the XML with the pending changes applied.

   `TestClientReadXML` class is using the `XMLInterface.XML_OPT_ASSOC_CONSISTENT` flag described in How to Produce XML for Queried Data to ensure that new, unposted rows are included in the XML.

```
ViewObject vo = am.findViewObject("CustomersView");
Key k = new Key(new Object[] { 16 });
// 1. Find a target row by key (e.g. for customer "The Sports Emporium")
Row sports = vo.findByKey(k, 1)[0];
// 2. Show the XML produced for the row before changes are applied
printXML(sports.writeXML(-1, XMLInterface.XML_OPT_ALL_ROWS));
// 3. Obtain parsed XML document with changes to apply using helper method
Element xmlToRead = getInsertUpdateDeleteXMLGram();
printXML(xmlToRead);
// 4. Read the XML document to apply changes to the row
sports.readXML(getInsertUpdateDeleteXMLGram(), -1);
// 5. Show the XML with the pending changes applied
printXML(sports.writeXML(-1, XMLInterface.XML_OPT_ALL_ROWS |
                              XMLInterface.XML_OPT_ASSOC_CONSISTENT));
```

Running the code in the above example initially displays the "before" version of The Sports Emporium information. Notice that:

- The `City` attribute has a value of "Lapeer"
- There is no `ShippedDate` attribute for order ID `1018`
- There is an orders row for order ID `1008`, and
- There is no orders row related to order ID `9999`.

```
<CustomersViewRow>
   <Id>16</Id>
   <Name>The Sports Emporium</Name>
   <Address>222 E Nepessing St</Address>
   <City>Lapeer</City>
   <State>MI</State>
   <CountryId>4</CountryId>
   <ZipCode>48446</ZipCode>
   <CreditRatingId>1</CreditRatingId>
   <SalesRepId>13</SalesRepId>
   <OrdersView>
      <OrdersViewRow>
         <Id>1018</Id>
         <CustomerId>16</CustomerId>
         <DateOrdered>2013-02-25</DateOrdered>
         <SalesRepId>11</SalesRepId>
         <Total>195.99</Total>
         <PaymentTypeId>2</PaymentTypeId>
         <PaymentOptionId>1009</PaymentOptionId>
         <OrderFilled>Y</OrderFilled>
      </OrdersViewRow>
      <OrdersViewRow>
         <Id>1008</Id>
         <CustomerId>16</CustomerId>
         <DateOrdered>2013-02-26</DateOrdered>
         <SalesRepId>14</SalesRepId>
         <Total>100.97</Total>
         <PaymentTypeId>2</PaymentTypeId>
         <PaymentOptionId>1009</PaymentOptionId>
         <OrderFilled>N</OrderFilled>
      </OrdersViewRow>
      <OrdersViewRow>
         <Id>1023</Id>
         <CustomerId>16</CustomerId>
         <DateOrdered>2013-01-11</DateOrdered>
         <DateShipped>2013-01-15</DateShipped>
         <SalesRepId>11</SalesRepId>
         <Total>2451.97</Total>
         <PaymentTypeId>2</PaymentTypeId>
         <PaymentOptionId>1009</PaymentOptionId>
         <OrderFilled>Y</OrderFilled>
      </OrdersViewRow>
   </OrdersView>
</CustomersViewRow>
```

After applying the changes from the XML document using `readXML()` to the row and printing its XML again using `writeXML()` you see that:

- The `City` attribute is now Houston.
- A new orders row for order `9999` got created.
- The date shipped for orders row `1018` has been set to 2013-03-13, and
- The orders row for order `1008` is removed

```
<CustomersViewRow>
   <Id>16</Id>
   <Name>The Sports Emporium</Name>
   <Address>222 E Nepessing St</Address>
   <City>Houston</City>
   <State>MI</State>
   <CountryId>4</CountryId>
   <ZipCode>48446</ZipCode>
   <CreditRatingId>1</CreditRatingId>
   <SalesRepId>13</SalesRepId>
   <OrdersView>
      <OrdersViewRow>
         <Id>9999</Id>
         <CustomerId>16</CustomerId>
         <Total>9999.00</Total>
      </OrdersViewRow>
      <OrdersViewRow>
         <Id>1018</Id>
         <CustomerId>16</CustomerId>
         <DateOrdered>2013-02-25</DateOrdered>
         <DateShipped>2013-03-13</DateShipped>
         <SalesRepId>11</SalesRepId>
         <Total>195.99</Total>
         <PaymentTypeId>2</PaymentTypeId>
         <PaymentOptionId>1009</PaymentOptionId>
         <OrderFilled>Y</OrderFilled>
      </OrdersViewRow>
      <OrdersViewRow>
         <Id>1023</Id>
         <CustomerId>16</CustomerId>
         <DateOrdered>2013-01-11</DateOrdered>
         <DateShipped>2013-01-15</DateShipped>
         <SalesRepId>11</SalesRepId>
         <Total>2451.97</Total>
         <PaymentTypeId>2</PaymentTypeId>
         <PaymentOptionId>1009</PaymentOptionId>
         <OrderFilled>Y</OrderFilled>
      </OrdersViewRow>
   </OrdersView>
</CustomersViewRow>
```

> **Note:**
>
> The example illustrated using `readXML()` to apply changes to a single row. If the XML document contained a wrapping `CustomersView` row, including the primary key attribute in each of its one or more nested `CustomersViewRow` elements, then that document could be processed using the `readXML()` method on the `CustomersView` view object for handling operations for multiple `CustomersView` rows.

Working Programmatically with Custom Data Sources and the Framework Base Class ProgrammaticViewObjectImpl

ADF Business Components supports programmatic view objects that you can create to interact with custom data sources.

The framework base classes `oracle.jbo.server.ProgrammaticViewObjectImpl` and `oracle.jbo.serverProgrammaticViewRowImpl` let you take control of interacting with a custom data source and returning the collection of data. ADF will internally take care of converting the data object into a `ViewRow`.

These framework base implementation classes provide specific hook points for the application developer to interact with a custom data source and perform operations. Using the framework base class, ADF supports features like activation/passivation for the given collection. Unlike earlier, developers do not need to code a wide variety of methods that require understanding of the various stages in the view object lifecycle to populate the row programmatically.

> **Note:**
>
> Prior to the `ProgrammaticViewObjectImpl` base class, applications extended from the base class `oracle.jbo.server.ViewObjectImpl` and overrode a number of lifecycle methods for custom behavior. Using programmatic view objects means integration with a custom data source is accomplished by the application developer without needing to understand the various stages of the view object lifecycle.

To work with programmatic view objects in the ADF Model project, you select the Programmatic option in the Create View Object wizard. Then View Object java classes that you generate in the wizard will by default extend from the framework base classes `oracle.jbo.server.ProgrammaticViewObjectImpl` and `oracle.jbo.server.ProgrammaticViewRowImpl` instead of extending from the classic style framework classes `oracle.jbo.server.ViewObjectImpl` and `oracle.jbo.server.ViewRowImpl`.

> **Note:**
>
> In order to ensure this behavior when creating a programmatic view object, you must de-select the option in the ADF Business Components Preferences dialog that enables extending from the classic style base class `oracle.jbo.server.ViewObjectImpl`. The option in the ADF Business Components-View Objects page of the Tools-Preferences dialog is **Classic Programmatic View** {de-select to enable extending from `ProgrammaticViewObjectImpl`).

How to Create a View Object Class Extending ProgrammaticViewObjectImpl

To create a programmatic view object that extends `oracle.jbo.server.ProgrammaticViewObjectImpl` or `oracle.jbo.server.ProgrammaticViewRowImp`, you use the Create View Object wizard and select the **Programmatic** data source option.

Before you begin:

You must disable classic style programmatic view object generation. In the Tools-Preferences dialog, de-select the option **Classic Programmatic View** on the ADF Business Components-View Objects page. If you leave this option selected (default), JDeveloper will generate view object Java classes that extend `oracle.jbo.server.ViewObjectImpl` and `oracle.jbo.server.ViewRowImpl`.

For details about classic mode, see Using Classic Style Programmatic View Objects for Alternative Data Sources.

To create programmatic view objects extending ProgrammaticViewObjectImpl:

1. In the Applications window, right-click the project in which you want to create the view object and choose **New**.
2. In the New Gallery, expand **Business Tier**, select **ADF Business Components** and then **View Object**, and click **OK**.
3. In the Create View Object wizard, in the Name page, provide a name and package for the view object. For the data source, select **Programmatic**.
4. In the Attributes page, click **New** one or more times to define the view object attributes your programmatic view object requires.
5. In the Attribute Settings page, adjust any setting you may need to for the attributes you defined.
6. In the Java page, select **Generate View Object Class** to enable a custom view object class (`ProgrammaticViewObjectImpl`) to contain your code.
7. Click **Finish** to create the view object.

   In your view object's custom Java class, override the methods described in section Key Framework Methods to Override for ProgrammaticViewObjectImpl Based View Objects to implement your custom data retrieval strategy.

Key Framework Methods to Override for ProgrammaticViewObjectImpl Based View Objects

You will override one of two methods, depending on the how the view object's **Access Mode** property has been set.

The property **Access Mode** determines how many rows will be returned by the view object query. The property’s default setting `SCROLLABLE` will cache the view object rows as you read them in. The `RANGEPAGING` setting fetches the number of rows specified by a range size. For more details about access mode, see section Using Range Paging to Efficiently Scroll Through Large Result Sets.

SCROLLABLE

If access mode is `SCROLLABLE`, then the `getScrollableData()` API needs to be overridden to provide a concrete implementation.

- `protected Collection<Object> getScrollableData(ScrollableDataFilter dataFilter)`

  For a scrollable programmatic view object the framework expects the extender to provide the complete `dataset` as the result of the following API call. The framework will internally manage the API calls when the client asks for next or previous rows in the collection.

RANGEPAGING

If access mode is `RANGEPAGING`, then the application need to override following two API methods together.

- `/Apps should return the collection within a current Range.`

  `protected Collection<Object> getRangePagingData(RangePagingDataFilter dataFilter)`

- `protected Collection<Object> retrieveDataByKey(Key key , int size).`

For a range paging view object the framework expects the extender to only provide the collection of rows with a current range. When the framework needs a different range of rows, a followup API call will be made. The current Range's start index and size information can be obtained from `RangePagingDataFilter` object passed as an argument.

What You May Need to Know About Programmatic View Object Triggers

You can use triggers defined in the Groovy scripting language in place of overridden methods of `ProgrammaticViewObjectImpl.java` to perform data retrieval operations on the custom datasource. Triggers can be called from the various programmatic view object lifecycle points.

The application allows you to call triggers that are defined in Groovy script in place of Java hook points. These triggers interact with a custom data source and perform data retrieval operations. They can be called from the various programmatic view object lifecycle points. Although the option of subclassing the framework class, namely `ProgrammaticViewObjectImpl` and overriding certain methods to define implementations specific to a custom datasource remains, in order for the trigger to execute, the Java hook points should not be defined. Alternatively, if Java hook points are defined, then the triggers shouldn't exist for the Java hook points to execute. Note that if Java hook points as well as triggers are defined, then the triggers will not execute. The application allows you to utilize Groovy scripts in the overridden methods to interact with the custom data source. These triggers can be called from various programmatic view object lifecycle points.

Specifically, the programmatic view object triggers are:

`FetchData`

This trigger is used to retrieve a data collection from the custom datasource. This trigger is invoked in place of the `getScrollableData()` and `getRangePagingData()` methods of `ProgrammaticViewObjectImpl.java` depending on whether the access mode of the view object is scroll or range paging. The `returnList` data collection is populated with the records (objects/maps) retrieved from the custom data source.

The following sample code illustrates usage of this trigger. The following context specific keywords is used in the sample code.

- `dataFilter` – Returns the DataFilter object which is a ScrollableDataFilter or RangePagingDataFilter depending on the mode of the view object.
- `dataFilter.fetchSize` – Returns the range size of the data to be fetched.
- `dataFilter.fetchStart` — Returns the range start value of the data to be fetched.
- `dataFilter.searchCriteria` - Returns a list of ViewCriteria applied on the view object.
- `dataFilter.sortCriteria` - Returns a list of SortCriteria applied on the view object.

- `dataFilter.bindParameterValues` - Returns a map containing the bind parameter names and values corresponding to the rowset that is currently being executed.
- `dataFilter.masterData` - When used in the context of a detail view object, it returns the dataprovider object of the master row.
- `returnList` - The collection that is to be populated with the records (objects/maps) from the custom data source.

```
@TriggerExpression(triggerType="FetchData", name="FetchData_Rule_0")
def FetchData_Rule_0_ValidationRuleScript_Trigger()
{
   def empData = model.data.DataCenter.getInstance().getEmpData()
   def empFilteredData = new ArrayList()
   model.data.DeptObject master = (model.data.DeptObject)dataFilter.masterData

   for (emp in empData)
   {
      if (master == null || emp.deptno.equals(master.deptno))
      {
         empFilteredData.add(emp)
      }
   }

   def fetchStart = dataFilter.fetchStart
   def fetchSize = dataFilter.fetchSize
   if(fetchStart == -1)
   {
     fetchStart = 0;
   }
   for(int i = fetchStart; i < fetchSize + fetchStart && i <
empFilteredData.size() ; i++)
   {
     returnList.add(empFilteredData.get(i))
@TriggerExpression(triggerType="RetrieveDataByKey",
name="RetrieveDataByKey_Rule_0")
def RetrieveDataByKey_Rule_0_ValidationRuleScript_Trigger()
{
   def empData = model.data.DataCenter.getInstance().getEmpData()

   def keyValue = dataFilter.key.keyValues[0]

   def iter = empData.iterator()
   while (iter.hasNext())
   {
      def emp = iter.next();
      if (emp.empno.equals(keyValue))
      {
         returnList.add(emp);
      }
   }
   }
}
```

`RetrieveDataByKey Trigger`

This trigger is used to retrieve a data collection matching a key value from the custom datasource. It is invoked in place of the `retrieveDataByKey` method of `ProgrammaticViewObjectImpl.java`. This trigger is invoked when the required data matching the key is not present in the current range of rows retrieved by the view object. This trigger retrieves the collection of objects/maps matching the key value.

The following sample code illustrate the usage of this trigger. The following context specific keywords is used in the sample code.

- `dmlFilter.key` - Returns an oracle.jbo.Key containing the key values of the data/object to be retrieved.
- `dmlFilter.fetchSize` - Returns the fetch size.
- `returnList` - The collection that is to be populated with the records (objects/maps) from the custom data source that match the key value.

```
@TriggerExpression(triggerType="RetrieveDataByKey",
name="RetrieveDataByKey_Rule_0")
def RetrieveDataByKey_Rule_0_ValidationRuleScript_Trigger()
{
   def empData = model.data.DataCenter.getInstance().getEmpData()

   def keyValue = dataFilter.key.keyValues[0]

   def iter = empData.iterator()
   while (iter.hasNext())
   {
      def emp = iter.next();
      if (emp.empno.equals(keyValue))
      {
         returnList.add(emp);
      }
   }
}
```

Key Framework Methods to Override for ProgrammaticViewRowImpl Based View Objects

The framework base class `oracle.jbo.server.ProgrammaticViewRowImpl` is an implementation class for programmatic `ViewRow`.

Oracle ADF provides multiple hookpoints to interact with programmatic view row during various lifecycle phases. Not all of the following API methods are mandatory to override.

- `createRowData(HashMap attrNameValueMap)`

  If the use case requires creation of new rows for a programmatic view object, then override this API. This hookpoint allows applications to create a `DataProvider` for this newly created row and return it to the framework for further management.
- `updateDataProvider(Object rowDataProvider , ViewAttributeDefImpl attrDef , Object newValue)`

  If the programmatic view object is created without a backing entity object, then override this API to retain the modified values of attributes in the row. Applications should override this API and update `newvalue` into `DataProvider`.
- `convertToSourceType(ViewAttributeDefImpl vad, String sourceType , Object val)`

  If required, the application should override this API to provide custom implementation for conversion of value from JavaType to sourceType. The framework calls this hookpoint prior to updating the `value` in `DataProvider`.

- `convertToAttributeType(AttributeDefImpl attrDef, Class javaTypeClass , Object val)`

  If required, applications should override this API to provide custom implementation for conversion of value from `sourceType` to `JavaType`. The framework calls this hookpoint as it fetches the value from `DataProvider`.

Using Classic Style Programmatic View Objects for Alternative Data Sources

You can override methods in a custom Java class of an ADF view object to programmatically retrieve data from data sources such as in-memory array, Java `*.properties file`, `REF CURSOR` etc.

By default view objects read their data from the database and automate the task of working with the Java Database Connectivity (JDBC) layer to process the database result sets. However, by overriding appropriate methods in its custom Java class, you can create a view object that programmatically retrieves data from alterative data sources like a `REF CURSOR`, an in-memory array, or a Java `*.properties` file, to name a few.

> **Note:**
>
> The sections describing alternative data sources are retained for legacy purposes. Starting in JDeveloper release 12.2.1.1.0, ADF Business Components provides improved support for programmatic view objects. New framework base classes for programmatic view objects implements common methods for custom data sources and provides hook points to interact with the view row during various phases of the view object lifecycle. This feature must be enabled in the View Objects page of the Tools-Preferences dialog. For more details, see How to Create a View Object Class Extending ProgrammaticViewObjectImpl.

The general process for enabling a programmatic view object to work with an non-database, data source involves overriding the default lifecycle methods of the view object. For a complete description of the lifecycle methods, see Key Framework Methods to Override for Programmatic View Objects.

To enable programmatic view objects for alternative data sources, follow this general process:

1. Create the view object in the view object wizard and select **Programmatic** for the data source.
2. In view object overview editor, generate the custom view object implementation class (`ViewObjImpl.java`).
3. In the custom view object implementation class, override `create()` to initialize any state required by the programmatic view object. At a minimum, remove all traces of a SQL query for this view object.
4. Override `executeQueryForCollection()` and call `setUserDataForCollection()` to associate the result of your custom query with a `UserData` object.

5. Override `hasNextForCollection()` and iterate over the `UserData` object to test for more records to retrieve. While `hasNextForCollection()` returns `true`:
   a. Override `createRowFromResultSet()` to create the row on the view object query collection.
   b. In your overridden `createRowFromResultSet()` method call the `getUserDataForCollection()` method to get an instance of the stored `UserData` object and `createNewRowForCollection()` to create a new blank row.
   c. If necessary, convert each record to an attribute data type supported by ADF Business Components view object attributes and call `populateAttributeForRow()` to store attribute values on the query collection.
   d. When `hasNextForCollection()` returns `false`, inform the view object that the query collection is done being populated by calling `setFetechCompleteForCollection()`.
6. Override `releaseUserDataForCollection()` to release the result set.

Because the view object component can be related to several active row sets at runtime, many of these lifecycle methods receive an `Object` parameter (a query collection) in which the ADF Business Components framework passes the collection of rows that your custom code fills, as well as the array of bind variable values that might affect which rows get populated into the specific collection.

You can store a `UserData` object with each collection of rows so that your custom data source implementation can associate any needed data source context information. The framework provides the `setUserDataForCollection()` and `getUserDataForCollection()` methods to get and set this context information for each collection. Each time one of the overridden framework methods is called, you can use the `getUserDataForCollection()` method to retrieve the correct `ResultSet` object associated with the collection of rows the framework populates.

How to Create a Read-Only Programmatic View Object

To create a read-only programmatic view object, you use the Create View Object wizard.

To create the read-only programmatic view object:

1. In the Applications window, right-click the project in which you want to create the view object and choose **New**.
2. In the New Gallery, expand **Business Tier**, select **ADF Business Components** and then **View Object**, and click **OK**.
3. In the Create View Object wizard, in the Name page, provide a name and package for the view object. For the data source, select **Programmatic**.
4. In the Attributes page, click **New** one or more times to define the view object attributes your programmatic view object requires.
5. In the Attribute Settings page, adjust any setting you may need to for the attributes you defined.
6. In the Java page, select **Generate View Object Class** to enable a custom view object class (`ViewObjImpl`) to contain your code.
7. Click **Finish** to create the view object.

In your view object's custom Java class, override the methods described in Key Framework Methods to Override for Programmatic View Objects to implement your custom data retrieval strategy.

How to Create an Entity-Based Programmatic View Object

To create a entity-based view object with programmatic data retrieval, create the view object in the normal way, enable a custom Java class for it, and override the methods described in the next section to implement your custom data retrieval strategy.

Key Framework Methods to Override for Programmatic View Objects

A programmatic view object typically overrides all of the following methods of the base `ViewObjectImpl` class to implement its custom strategy for retrieving data:

- `create()`

  This method is called when the view object instance is created and can be used to initialize any state required by the programmatic view object. At a minimum, this overridden method will contain the following lines to ensure the programmatic view object has no trace of a SQL query related to it:

  ```
  // Wipe out all traces of a query for this VO
  getViewDef().setQuery(null);
  getViewDef().setSelectClause(null);
  setQuery(null);
  ```

- `executeQueryForCollection()`

  This method is called whenever the view object's query needs to be executed (or reexecuted). Your implementation must not override this method and completely change the query or change the list of parameters. For implementation best practices, see The Overridden executeQueryForCollection() Method.

- `hasNextForCollection()`

  This method is called to support the `hasNext()` method on the **row set iterator** for a row set created from this view object. Your implementation returns `true` if you have not yet exhausted the rows to retrieve from your programmatic data source.

- `createRowFromResultSet()`

  This method is called to populate each row of retrieved data. Your implementation will call `createNewRowForCollection()` to create a new blank row and then `populateAttributeForRow()` to populate each attribute of data for the row. Note that calls to `populateAttributeForRow()` require that you populate each attribute with the correctly typed value, as allowed by ADF Business Components view objects. The method does not handle data type conversion, and therefore your program is responsible for ensuring that the data type of each value matches an allowed view object attribute type.

- `getQueryHitCount()`

  This method is called to support the `getEstimatedRowCount()` method. Your implementation returns a count, or estimated count, of the number of rows that will be retrieved by the programmatic view object's query.

- `getCappedQueryHitCount()`

The row count query is optimized to determine whether the query would return more rows than the capped value specified by a global row fetch limit. If the number of potential rows is greater, the method returns a negative number. Although the scroll bar in the user interface may not be able to use the row count query to show an accurate scroll position, your row count query will perform well even with large tables. Your implementation can check the `Cap` value and return the actual row count. The framework obtains the `Cap` value from the row fetch limit specified in the `adf-config.xml` file. Note that `oldCap` is not used

The method returns a row count when the value is less than or equal to the `Cap`; otherwise, returns a negative number. Your implementation can use this information to build a performing row count query. If performance is not important, you may just return the same value as `getQueryHitCount()`, but the `getCappedQueryHitCount()` now provides additional information about the number of rows it is looking for. For example, assume the `DEPT` table has one million rows. When `getQueryHitCount()` is called, the framework will execute `SELECT COUNT(*) FROM DEPT` against the database and return all one million rows. However, when you intend to show at most 500 rows, you can set a global row fetch limit and call `getCappedQueryHitCount()`. In this case, the framework will execute `SELECT COUNT(*) FROM DEPT WHERE ROWNUM <= 500`. The query will complete much faster and also provides a performance improvements. The method wraps the view object query in a nested clause like `SELECT COUNT(*) FROM (SELECT DEPTNO, LOC FROM DEPT) WHERE ROWNUM <= :cap` (assuming that `SELECT DEPTNO, LOC FROM DEPT` is the view object query statement).

- `protected void releaseUserDataForCollection()`

  Your code can store and retrieve a user data context object with each row set. This method is called to allow you to release any resources that may be associated with a row set that is being closed.

The examples in the following sections each override these methods to implement different kinds of programmatic view objects.

How to Create a View Object on a REF CURSOR

Sometimes your application might need to work with the results of a query that is encapsulated within a stored procedure. PL/SQL allows you to open a cursor to iterate through the results of a query, and then return a reference to this cursor to the client. This so-called `REF CURSOR` is a handle with which the client can then iterate the results of the query. This is possible even though the client never actually issued the original SQL `SELECT` statement.

> **Note:**
>
> The example in this section refers to the `oracle.summit.model.viewobjectonrefcursor` package in the `SummitADF_Examples` application workspace.

Declaring a PL/SQL package with a function that returns a `REF CURSOR` is straightforward. The following example shows how your package might look.

```
CREATE OR REPLACE PACKAGE RefCursorExample IS
  TYPE ref_cursor IS REF CURSOR;
  FUNCTION get_orders_for_customer(p_custId NUMBER) RETURN ref_cursor;
  FUNCTION count_orders_for_customer(p_custId NUMBER) RETURN NUMBER;
END RefCursorExample;
.
/
show errors
CREATE OR REPLACE PACKAGE BODY RefCursorExample IS
  FUNCTION get_orders_for_customer(p_custIdl NUMBER) RETURN ref_cursor IS
    the_cursor ref_cursor;
  BEGIN
    OPEN the_cursor FOR
      SELECT o.id, o.date_ordered, o.total
        FROM s_ord o, s_customer c
        WHERE o.customer_id = c.id
        AND c.id = p_custId;
      RETURN the_cursor;
  END get_orders_for_customer;

  FUNCTION count_orders_for_customer(p_custId NUMBER) RETURN NUMBER IS
    the_count NUMBER;
  BEGIN
    SELECT COUNT(*)
      INTO the_count
      FROM s_ord o, s_customer c
      WHERE o.customer_id = c.id
      AND c.id = p_cust_id;
    RETURN the_count;
  END count_orders_for_customer;
END RefCursorExample;
.
/
show errors
```

After defining an entity-based `OrdersForCustomer` view object with an entity usage for an `Order` **entity object**, go to its custom Java class `OrdersForCustomerImpl.java`. Then, override the methods of the view object as described in the following sections.

The Overridden create() Method

The `create()` method removes all traces of a SQL query for this view object. The following example shows how to override the `create()` method for this purpose.

```
protected void create() {
  getViewDef().setQuery(null);
  getViewDef().setSelectClause(null);
  setQuery(null);
}
```

The Overridden executeQueryForCollection() Method

The `executeQueryForCollection()` method is executed when the framework needs to issue the database query for the query collection based on this view object. One view object can produce many related result sets, each potentially the result of different bind variable values. If the row set in query is involved in a framework-coordinated master/detail view link, then the `params` array will contain one or more framework-supplied name-value pairs of bind parameters from the source view object. If there are any user-supplied bind parameter values, they will *precede* the framework-supplied bind variable values in the params array, and the number of user parameters will be indicated by the value of the numUserParams argument.

The method calls a helper method retrieveRefCursor() to execute the stored function and return the REF CURSOR return value, cast as a JDBC ResultSet. The following example shows how to override the executeQueryForCollection() method for this purpose.

```
protected void executeQueryForCollection(Object qc,Object[] params,
                                         int numUserParams) {
  storeNewResultSet(qc,retrieveRefCursor(qc,params));
  super.executeQueryForCollection(qc, params, numUserParams);
}
```

Then, it calls the helper method storeNewResultSet() that uses the setUserDataForCollection() method to store this ResultSet with the collection of rows for which the framework is asking to execute the query. The following example shows how to define the storeNewResultSet() method for this purpose.

```
private void storeNewResultSet(Object qc, ResultSet rs) {
  ResultSet existingRs = getResultSet(qc);
  // If this query collection is getting reused, close out any previous rowset
  if (existingRs != null) {
     try {existingRs.close();} catch (SQLException s) {}
  }
  setUserDataForCollection(qc,rs);
  hasNextForCollection(qc); // Prime the pump with the first row.
}
```

The retrieveRefCursor() method uses the helper method described in Invoking Stored Procedures and Functions to invoke the stored function and return the REF CURSOR. The following example shows how to define the retrieveRefCursor() method for this purpose.

```
private ResultSet retrieveRefCursor(Object qc, Object[] params) {
  ResultSet rs = (ResultSet)callStoredFunction(OracleTypes.CURSOR,
                  "RefCursorExample.get_orders_for_customer(?)",
                  new Object[]{getNamedBindParamValue("bv_custId",params)});
  return rs ;
}
```

The Overridden createRowFromResultSet() Method

For each row that the framework needs fetched from the data source, it will invoke your overridden createRowFromResultSet() method. The implementation retrieves the collection-specific ResultSet object from the user-data context. It uses the getResultSet() method to retrieve the result set wrapper from the query-collection user data, and the createNewRowForCollection() method to create a new blank row in the collection, and then uses the populateAttributeForRow() method to populate the attribute values for each attribute defined at design time in the view object overview editor.

Note:

Calls to `populateAttributeForRow()` assume that attribute values that you populate on the view object already conform to the data types expected for ADF Business Components view object attributes. The method will not handle data type conversion, and therefore your program is responsible for converting the data type of the populated value to match an allowed view object attribute type.

The following example shows how to override the `createRowFromResultSet()` method for this purpose.

```
protected ViewRowImpl createRowFromResultSet(Object qc, ResultSet rs) {
  /*
   * We ignore the JDBC ResultSet passed by the framework (null anyway) and
   * use the resultset that we've stored in the query-collection-private
   * user data storage
   */
  rs = getResultSet(qc);

  /*
   * Create a new row to populate
   */
  ViewRowImpl r = createNewRowForCollection(qc);
  try {
    /*
     * Populate new row by type correct attribute slot number for current row
     * in Result Set
     */
    populateAttributeForRow(r,0, rs.getLong(1));
    populateAttributeForRow(r,1, rs.getString(2));
    populateAttributeForRow(r,2, rs.getString(3));
  }
  catch (SQLException s) {
    throw new JboException(s);
  }
  return r;
}
```

The Overridden hasNextForCollectionMethod()

The overridden implementation of the framework method `hasNextForCollection()` has the responsibility to return `true` or `false` based on whether there are more rows to fetch. When you've hit the end, you call the `setFetchCompleteForCollection()` to tell the view object that this collection is done being populated. The following example shows how to override the `hasNextForCollection()` method for this purpose.

```
protected boolean hasNextForCollection(Object qc) {
  ResultSet rs = getResultSet(qc);
  boolean nextOne = false;
  try {
    nextOne = rs.next();
    /*
     * When were at the end of the result set, mark the query collection
     * as "FetchComplete".
     */
    if (!nextOne) {
      setFetchCompleteForCollection(qc, true);
      /*
       * Close the result set, we're done with it
       */
      rs.close();
    }
  }
  catch (SQLException s) {
   throw new JboException(s);
  }
  return nextOne;
}
```

The Overridden releaseUserDataForCollection() Method

Once the collection is done with its fetch-processing, the overridden `releaseUserDataForCollection()` method gets invoked and closes the `ResultSet` cleanly so no database cursors are left open. The following example shows how to override the `releaseUserDataForCollection()` method for this purpose.

```
  protected void releaseUserDataForCollection(Object qc, Object rs) {
     ResultSet userDataRS = getResultSet(qc);
     if (userDataRS != null) {
      try {
        userDataRS.close();
      }
      catch (SQLException s) {
        /* Ignore */
      }
    }
    super.releaseUserDataForCollection(qc, rs);
  }
```

The Overridden getQueryHitCount() Method

Lastly, in order to properly support the view object's `getEstimatedRowCount()` method, the overridden `getQueryHitCount()` method returns a count of the rows that would be retrieved if all rows were fetched from the row set. Here the code uses a stored function to get the job done. Since the query is completely encapsulated behind the stored function API, the code also relies on the PL/SQL package to provide an implementation of the count logic as well to support this functionality. The following example shows how to override the `getQueryHitCount()` method for this purpose.

```
public long getQueryHitCount(ViewRowSetImpl viewRowSet) {
  Long result = (Long)callStoredFunction(NUMBER,
                   "RefCursorExample.count_orders_for_customer(?)",
                   viewRowSet.getParameters(true));
  return result.longValue();
}
```

Creating a View Object with Multiple Updatable Entities

When adding a secondary entity usage to an ADF view object with multiple entity usages, the secondary entity usage is non-updateable by default. You can override this default behavior and make it updateable.

By default, when you create a view object with multiple entity usages, each secondary entity usage that you add to a view object in the overview editor is configured with these settings:

- The **Updatable** checkbox is deselected
- The **Reference** checkbox is selected

You can change the default behavior to enable a secondary entity usage to be updatable by selecting the usage in the **Selected** list of the **Entity Objects** page of the view object overview editor and selecting the **Updatable** checkbox.

Additionally, for each secondary entity usage, you can decide whether to leave **Reference** select to control whether or not to refresh the attributes of the secondary entity when the entity lookup information changes. By default, **Reference** is selected to ensure attributes of each secondary entity objects will be refreshed. For details about this setting when you allow row inserts with multiple entity usages, see What Happens at Runtime: View Row Creation.

Table 11-4 summarizes the combinations you can select when you define secondary entity usages for a view object.

Table 11-4 View Object Properties to Control View Row Creation Behavior

Updatable	Reference	View Row Behavior
`true`	true	This combination allows the entity usage's attributes to be updated and keeps its attributes synchronized with the value of the primary key. Since this combination works fine with the view link consistency feature, you can use it to make sure your view object only has one entity object usage that will participate in inserts.
`true`	false	This combination allows the entity usage's attributes to be updated but prevents its attributes from being changed by the a primary key lookup. This is a rather rare combination, and works best in situations where you only plan to use the view object to update or delete existing data. With this combination, the user can update attributes related to any of the nonreference, updatable entity usages and the view row will delegate the changes to the appropriate underlying entity rows. **Note:** The combination of the view link consistency feature with a view object having some of its secondary entity usages set as `Updatable=true`, `Reference=false` can end up creating unwanted extra new entities in your application.

Table 11-4 (Cont.) View Object Properties to Control View Row Creation Behavior

Updatable	Reference	View Row Behavior
false	true	This is the default behavior, described in How to Create Joins for Entity-Based View Objects. This combination assumes you do not want the entity usage to be updatable.

If you need a view object with multiple updatable entities to support creating new rows (`Updatable=true`, `Reference=false`) and the association between the entity objects is not a composition, then you need to write a bit of code, as described in How to Programmatically Create New Rows With Multiple Updatable Entity Usages.

How to Programmatically Create New Rows With Multiple Updatable Entity Usages

If you need a view object with multiple updatable entities to support creating new rows (`Updatable=true`, `Reference=false`) and the association between the entity objects is not a composition, then you need to override the `create()` method of the view object's custom view row class to enable that to work correctly.

> **Note:**
>
> You only need to write code to handle creating new rows when the association between the updatable entities is not a composition. If the association is a composition, then ADF Business Components handles this automatically.

When you call `createRow()` on a view object with multiple update entities, it creates new entity row parts for each updatable entity usage. Since the multiple entities in this scenario are related by an association, there are three pieces of code you might need to implement to ensure the new, associated entity rows can be saved without errors:

1. You may need to override the `postChanges()` method on entity objects involved to control the correct posting order.
2. If the primary key of the associated entity is populated by a database sequence using `DBSequence`, and if the multiple entity objects are associated but not composed, then you need to override the `postChanges()` and `refreshFKInNewContainees()` method to handle cascading the refreshed primary key value to the associated rows that were referencing the temporary value.
3. You need to override the `create()` method of the view object's custom view row class to modify the default row creation behavior to pass the context of the parent entity object to the newly created child entity.

To understand the code for steps 1 and 2, see the example with associated entity objects described in How to Control Entity Posting Order to Prevent Constraint Violations. The last thing you need to understand is how to override `create()` method on the view row. Consider a ProductInventoryVO view object with a primary entity usage of Product and secondary entity usage of Inventory. Assume the Product entity usage is marked as updatable and nonreference, while the Inventory entity usage is a reference entity usage.

> **Note:**
>
> The example in this section refers to the oracle.summit.model.multieoupdate package in the SummitADF_Examples application workspace.

The following example shows the commented code required to correctly sequence the creation of the multiple, updatable entity row parts during a view row create operation.

```
/**
 * By default, the framework will automatically create the new
 * underlying entity object instances that make up this
 * view object row being created.
 *
 * We override this default view object row creation to explicitly
 * pre-populate the new (detail) InventoryImpl instance using
 * the new (master) ProductImpl instance. Since all entity objects
 * implement the AttributeList interface, we can directly pass the
 * new ProductImpl instance to the InventoryImpl create()
 * method that accepts an AttributeList.
 */
protected void create(AttributeList attributeList) {
  // The view row will already have created "blank" entity instances
  // so be sure to use the respective names of the entity instance.
  ProductImpl newProduct = getProduct();
  InventoryImpl newInventory = getInventory();
   try {
      // Let inventory "blank" entity instance to do programmatic defaulting
      newProduct.create(attributeList);
      // Let inventory "blank" entity instance to do programmatic
      // defaulting passing in new ProductImpl instance so its attributes
      // are available to the ProductInventoryVORowImpl's create method.
      newInventory.create(newProduct);
   }
   catch (JboException ex) {
     newProduct.revert();
     newInventory.revert();
     throw ex;
   }
   catch (Exception otherEx) {
     newProduct.revert();
     newInventory.revert();
     throw new RowCreateException(true      /* EO Row? */,
                                 "Inventory" /* EO Name */,
                                 otherEx   /* Details */);
   }
}
```

In order for this ProductInventoryVO view object's view row class (ProductInventoryVORowImpl class) to be able to invoke the protected create()

method on the Product and Inventory entity objects, the entity object classes need to override their create() methods for method accessibility:

```
/**
 * Overridding this method in this class allows friendly access
 * to the create() method by other classes in this same package, like the
 * ProductInventoryVO view object implementation class, whose overridden
 * create() method needs to call this.
 * @param nameValuePair
 */
  protected void create(AttributeList nameValuePair) {
    super.create(nameValuePair);
  }
```

When overriding the create() method, the declaration of the method will depend on the following conditions:

- If the view object and entity objects are in the same package, the overridden create() method can have protected access and the ProductInventoryVORowImpl class will have access to them.
- If either entity object is in a different package, then ProductImpl.create() and InventoryImpl.create() (whichever is in a different package) have to be declared public in order for the ProductInventoryVORowImpl class to be able to invoke them.

What Happens at Runtime: View Row Creation

If you need a view object with multiple updatable entities to support creating new rows, you will want to understand that the **Reference** flag controls behavior related to view row creation, as well as automatic association-driven lookup of information. If you disable the **Reference** flag for a given entity usage, then:

- Each time a new view row is created, a new entity instance will be created for that entity usage.
- The entity row to which the view row points for its storage of view row attributes related to that entity usage is never changed automatically by the framework.

Conversely, if you leave the **Reference** flag enabled (default) for an entity usage then:

- No new entity instance will be created for that entity usage when a new view row is created.
- The entity row to which the view row points for storage of view row attributes related to that entity usage will automatically be kept in sync with changes made to attributes in the view row that participate in an association with said entity.

Consider a ProductInventoryVO view object that joins information from the PRODUCT and INVENTORY tables to show ProductId, PName, ProductInventoryId (a name chosen in the view object editor to identify this as an attribute of ProductInventoryVO), InventoryInventoryId (a name chosen in the view object editor to identify this as an attribute of ProductInventoryVO) and AmountInStock attributes.

Now, consider what happens at runtime for the default case where you set up the secondary entity object marked as both updatable and reference:

- The Product entity object is the primary entity usage.

- The Inventory entity object is a secondary entity usage and is marked as a Reference.

When the user creates a new row in the ProductInventoryVO, ADF Business Components only creates a new Product entity instance (since the Reference flag for the Inventory entity usage is enabled). If the user changes the product's InventoryInventoryId attribute to 10, then ADF Business Components will automatically look up the Inventory entity with primary key 10 in the entity cache (reading it in from the database if not already in the cache) and make this new view row's Inventory entity usage point to this inventory 10 entity. That has the result of automatically reflecting the right AmountInStock value for inventory 10.

In the default scenario, the reference lookup occurs both because the entity usage for Inventory is marked as a reference, as well as the fact that an association exists between the Product entity and the Inventory entity. Through the association definition, ADF Business Components knows which attributes are involved on each side of this association. When any of the attributes on the Product side of the ProductToInventory association are modified, if the Inventory entity usage is marked as a Reference, ADF Business Components will perform that automatic reference lookup. If the user changes value of the AmountInStock to 20 in this new view row, after committing the change, the database will have a new product for inventory 10 and have updated the amount of inventory 10 to 20.

Now, consider what happens at runtime where you set up the secondary entity object marked as updatable and reference is disabled:

- The Product entity object is the primary entity usage.
- The Inventory entity object is a secondary entity usage, but this usage is not marked as a Reference.

In this scenario, when the user creates a new row in the ProductInventoryVO, ADF Business Components will create both a new Product entity instance and a new Inventory entity instance (since the Reference flag for the Inventory entity usage is disabled). If the user changes the product's InventoryId attribute to 10, it will have no effect on the value of the AmountInStock attribute being displayed in the row. Additionally, if the user sets InventoryInventoryId to 99 and AmountInStock to 20 in this new view row, after committing the changes, the database will have both a new product for inventory 10 and a new inventory number 99.

Programmatically Creating View Definitions and View Objects

When you create an ADF view object, the application creates a view definition object utilizing the XML file. Alternatively, you can create the view definition object programmatically using a framework class.

The oracle.jbo.server.ViewDefImpl class lets you dynamically define the view definition meta-object for view object instances. The view definition describes the view object's structure.

Typically, the application creates the view definition object by loading an XML file that you create using JDeveloper overview editors. When the application needs to create a view object instance, it queries the MetaObjectManager for the view object's view definition by the view definition name, it then finds the XML file, opens it, parses it, and constructs a view definition object in memory.

Alternatively, you can create the view definition programmatically using methods of the ViewDefImpl class. When you create a programmatic view definition, your application code begins with code like:

```
ViewDefImpl viewDef = new ViewDefImpl("MyViewDef");
viewDef.setFullName("sessiondef.mypackage.MyViewDef");
```

A view definition that you create must be uniquely identified by its full name, where the full name is a package-qualified name. Thus, you call setFullName() to pass in the package-qualified name of your view definition object (for example, sessiondef.mypackage.MyViewDef).

The MyViewDef name that you initially pass in is the short name of the view definition you create. Your application may pass the short name when an API requires a view definition name. For example, your application might request the defName parameter when invoking ApplicationModule.createViewObject(String, String).

To create a view definition and then create a view object instance based on that definition, follow these basic steps (as illustrated in the following example):

1. Create the view definition object and set the full name.
2. Define the view object SQL statement.
3. Resolve the view definition and save it into the MDS repository.
4. With the view definition, construct instance of view objects based on it.

Note:

To save the view definition into the MDS repository, the adf-config.xml file must be appropriately configured for the saved view definition. For details about configuring the adf-config.xml file, see What You May Need to Know About MDS Repository Configuration..

```
/*
  * 1. Create the view definition object.
  */
ViewDefImpl v = new ViewDefImpl("DefNameForTheObject");

v.setFullName("sessiondef.some.unique.DefNameForTheObject");

/*
* 2. Then, define the view object's SQL statement by using a fully-
* specified custom SQL query.
*/
v.setQuery("select e.empno,e.ename,e.sal,e.deptno,d.dname,d.loc,"+
               "d.deptno,trunc(sysdate)+1 tomorrow_date, "+
               "e.sal + nvl(e.comm,0) total_compensation, "+
               "to_char(e.hiredate,'dd-mon-yyyy') formated_hiredate"+
               "  from emp e, dept d "+
               " where e.deptno = d.deptno (+)"+
               " order by e.ename");
v.setFullSql(true);

/*
* 3. Then resolve and save the view definition.
*/
```

```
v.resolveDefObject();
v.writeXMLContents();
v.saveXMLContents();

/*
* 4. Finally, use the dynamically-created view definition to construct
* instances of view objects based on it.  myAM is an instance of
* oracle.jbo.ApplicationModule that will parent this VO instance.
*/
ViewObject vo = myAM.createViewObject("SomeInstanceName", v.getFullName());
```

What You May Need to Know About MDS Repository Configuration

After defining a view definition, it is important to write and save the view definition into the MDS repository. If it is not properly saved, you may encounter issues when the request is redirected to a different node in a cluster because the definition cannot be loaded and accessed from the other node.

In order to save the definition, you need to define the `mds-config` element of `adf-config.xml`. For example, your `adf-config.xml` file should contain definitions similar to those shown in the following example.

```
<mds-config version="11.1.1.000">
   <persistence-config>
<!-- metadata-namespaces must define /sessiondef and /persdef namespaces -->
     <metadata-namespaces>
       <namespace path="/sessiondef" metadata-store-usage="mymdsstore">
       <namespace path="/persdef" metadata-store-usage="mymdsstore">
     </metadata-namespaces>

     <metadata-store-usages>
       <metadata-store-usage id="mymdsstore" default-cust-store="true">
         <metadata-store name="fs1"
               class-name="oracle.mds.persistence.stores.file.FileMetadataStore">
<!-- metadata-path value should be the absolute dir path where you want
     the metadata documents to be written -->
           <property name="metadata-path" value="/tmp">
         </metadata-store>
       </metadata-store-usage>
     </metadata-store-usages>
   </persistence-config>

   <cust-config>
     <match path="/">
       <customization-class name="oracle.adf.share.config.UserCC">
     </match>
   </cust-config>
</mds-config>
```

If your `adf-config.xml` file already defines the `metadata-store-usage` element, then you may be able to define the two namespaces `/sessiondef` and `/persdef` so that they use that `metadata-store-usage` definition. For more information about MDS configuration entries in the `adf-config.xml` file, see adfc-config.xml. For more information about configuring MDS repositories, see the "Managing the Metadata Repository" chapter in *Administering Oracle Fusion Middleware*.

What You May Need to Know About Creating View Objects at Runtime

It's important to understand the overhead associated with creating view objects at runtime, as described in Programmatically Creating View Definitions and View Objects. Avoid the temptation to do this without a compelling business requirement. For example, if your application issues a query against a table whose name you know at design time and if the list of columns to retrieve is also fixed, then create a view object at design time. When you do this, your SQL statements are neatly encapsulated, can be easily explained and tuned during development, and incur no runtime overhead to discover the structure and data types of the resulting rows.

In contrast, when you use the `createViewObjectFromQueryStmt()` API on the `ApplicationModule` interface at runtime, your query is buried in code, it's more complicated to proactively tune your SQL, and you pay a performance penalty each time the view object is created. Since the SQL query statement for a dynamically created view object could theoretically be different each time a new instance is created using this API, an extra database round trip is required to discover the "shape" of the query results on-the-fly. Only create queries dynamically if you cannot know the name of the table to query until runtime. Most other needs can be addressed using a design-time created view object in combination with runtime API's to set bind variables in a fixed where clause, or to add an additional WHERE clause (with optional bind variables) at runtime.

Declaratively Preventing Insert, Update, and Delete

ADF Business Components provides an extension class that allows you to control insert, update and delete on a view object.

Some 4GL tools like Oracle Forms provide declarative properties that control whether a given data collection allows inserts, updates, or deletes. While the view object does not yet support this as a built-in feature in the current release, it's easy to add this facility using a framework extension class that exploits custom metadata properties as the developer-supplied flags to control insert, update, or delete on a view object.

Note:

The example in this section refers to the `oracle.summit.model.declblock` package in the `SummitADF_Examples` application workspace.

To allow developers to have control over individual view object instances, you could adopt the convention of using application module custom properties by the same name as the view object instance. For example, if an application module has view object instances named `ProductsInsertOnly`, `ProductsUpdateOnly`, `ProductsNoDelete`, and `Products`, your generic code might look for application module custom properties by these same names. If the property value contains `Insert`, then insert is enabled for that view object instance. If the property contains `Update`, then update allowed. And, similarly, if the property value contains `Delete`, then delete is allowed. You could use helper methods like this to test for these application module properties and determine whether insert, update, and delete are allowed for a given view object:

```
private boolean isInsertAllowed() {
  return isStringInAppModulePropertyNamedAfterVOInstance("Insert");
}
private boolean isUpdateAllowed() {
  return isStringInAppModulePropertyNamedAfterVOInstance("Update");
}
private boolean isDeleteAllowed() {
  return isStringInAppModulePropertyNamedAfterVOInstance("Delete");
}
private boolean isStringInAppModulePropertyNamedAfterVOInstance(String s) {
  String voInstName = getViewObject().getName();
  String propVal = (String)getApplicationModule().getProperty(voInstName);
  return propVal != null ? propVal.indexOf(s) >= 0 : true;
}
```

The following example shows the other code required in a custom framework extension class for view rows to complete the implementation. It overrides the following methods:

- `isAttributeUpdateable()`

  To enable the user interface to disable fields in a new row if insert is not allowed or to disable fields in an existing row if update is not allowed.

- `setAttributeInternal()`

  To prevent setting attribute values in a new row if insert is not allowed or to prevent setting attributes in an existing row if update is not allowed.

- `remove()`

  To prevent remove if delete is not allowed.

- `create()`

  To prevent create if insert is not allowed.

```
public class CustomViewRowImpl extends ViewRowImpl {
  public boolean isAttributeUpdateable(int index) {
    if (hasEntities() &&
        ((isNewOrInitialized() && !isInsertAllowed()) ||
         (isModifiedOrUnmodified() && !isUpdateAllowed()))) {
      return false;
    }
    return super.isAttributeUpdateable(index);
  }
  protected void setAttributeInternal(int index, Object val) {
    if (hasEntities()) {
      if (isNewOrInitialized() && !isInsertAllowed())
        throw new JboException("No inserts allowed in this view");
      else if (isModifiedOrUnmodified() && !isUpdateAllowed())
        throw new JboException("No updates allowed in this view");
    }
    super.setAttributeInternal(index, val);
  }
  public void remove() {
    if (!hasEntities() || isDeleteAllowed() || isNewOrInitialized())
      super.remove();
    else
        throw new JboException("Delete not allowed in this view");
  }
  protected void create(AttributeList nvp) {
    if (isInsertAllowed()) {
      super.create(nvp);
    } else {
      throw new JboException("Insert not allowed in this view");
    }
  }
  // private helper methods omitted from this example
}
```

What You May Need to Know About Programmatic View Object Triggers

You can use triggers defined in the Groovy scripting language in place of overridden methods of `ProgrammaticViewObjectImpl.java` to perform data retrieval operations on the custom datasource. Triggers can be called from the various programmatic view object lifecycle points.

The application allows you to call triggers that are defined in Groovy script in place of Java hook points. These triggers interact with a custom data source and perform data retrieval operations. They can be called from the various programmatic view object lifecycle points. Although the option of subclassing the framework class, namely `ProgrammaticViewObjectImpl` and overriding certain methods to define implementations specific to a custom datasource remains, in order for the trigger to execute, the Java hook points should not be defined. Alternatively, if Java hook points are defined, then the triggers shouldn’t exist for the Java hook points to execute. Note that if Java hook points as well as triggers are defined, then the triggers will not execute. The application allows you to utilize Groovy scripts in the overridden methods to interact with the custom data source. These triggers can be called from various programmatic view object lifecycle points.

Specifically, the programmatic view object triggers are:

`FetchData`

This trigger is used to retrieve a data collection from the custom datasource. This trigger is invoked in place of the `getScrollableData()` and `getRangePagingData()` methods of `ProgrammaticViewObjectImpl.java` depending on whether the access mode of the view object is scroll or range paging. The `returnList` data collection is populated with the records (objects/maps) retrieved from the custom data source.

The following sample code illustrates usage of this trigger. The following context specific keywords is used in the sample code.

- `dataFilter` – Returns the DataFilter object which is a ScrollableDataFilter or RangePagingDataFilter depending on the mode of the view object.
- `dataFilter.fetchSize` – Returns the range size of the data to be fetched.
- `dataFilter.fetchStart` — Returns the range start value of the data to be fetched.
- `dataFilter.searchCriteria` - Returns a list of ViewCriteria applied on the view object.
- `dataFilter.sortCriteria` - Returns a list of SortCriteria applied on the view object.
- `dataFilter.bindParameterValues` - Returns a map containing the bind parameter names and values corresponding to the rowset that is currently being executed.

- `dataFilter.masterData` - When used in the context of a detail view object, it returns the dataprovider object of the master row.
- `returnList` - The collection that is to be populated with the records (objects/maps) from the custom data source.

```
@TriggerExpression(triggerType="FetchData", name="FetchData_Rule_0")
def FetchData_Rule_0_ValidationRuleScript_Trigger()
{
   def empData = model.data.DataCenter.getInstance().getEmpData()
   def empFilteredData = new ArrayList()
   model.data.DeptObject master = (model.data.DeptObject)dataFilter.masterData

   for (emp in empData)
   {
      if (master == null || emp.deptno.equals(master.deptno))
      {
         empFilteredData.add(emp)
      }
   }

   def fetchStart = dataFilter.fetchStart
   def fetchSize = dataFilter.fetchSize
   if(fetchStart == -1)
   {
     fetchStart = 0;
   }
   for(int i = fetchStart; i < fetchSize + fetchStart && i <
empFilteredData.size() ; i++)
   {
     returnList.add(empFilteredData.get(i))
@TriggerExpression(triggerType="RetrieveDataByKey",
name="RetrieveDataByKey_Rule_0")
def RetrieveDataByKey_Rule_0_ValidationRuleScript_Trigger()
{
   def empData = model.data.DataCenter.getInstance().getEmpData()

   def keyValue = dataFilter.key.keyValues[0]

   def iter = empData.iterator()
   while (iter.hasNext())
   {
      def emp = iter.next();
      if (emp.empno.equals(keyValue))
      {
         returnList.add(emp);
      }
   }
   }
}
```

`RetrieveDataByKey Trigger`

This trigger is used to retrieve a data collection matching a key value from the custom datasource. It is invoked in place of the `retrieveDataByKey` method of `ProgrammaticViewObjectImpl.java`. This trigger is invoked when the required data matching the key is not present in the current range of rows retrieved by the view object. This trigger retrieves the collection of objects/maps matching the key value.

The following sample code illustrate the usage of this trigger. The following context specific keywords is used in the sample code.

- `dmlFilter.key` - Returns an oracle.jbo.Key containing the key values of the data/ object to be retrieved.
- `dmlFilter.fetchSize` - Returns the fetch size.
- `returnList` - The collection that is to be populated with the records (objects/maps) from the custom data source that match the key value.

```
@TriggerExpression(triggerType="RetrieveDataByKey",
name="RetrieveDataByKey_Rule_0")
def RetrieveDataByKey_Rule_0_ValidationRuleScript_Trigger()
{
   def empData = model.data.DataCenter.getInstance().getEmpData()

   def keyValue = dataFilter.key.keyValues[0]

   def iter = empData.iterator()
   while (iter.hasNext())
   {
      def emp = iter.next();
      if (emp.empno.equals(keyValue))
      {
         returnList.add(emp);
      }
   }
}
```

Defining Triggers for Programmatic View Objects

ADF Business Components allows you to apply triggers in a programmatic view object's lifecycle. Triggers execute at these set lifecycle events. You can use Groovy script to implement business rules that are executed in response to these programmatic view object triggers.

To add a trigger to a programmatic view object.

1. In the Applications Navigator, double-click the view object you want to add a trigger.
2. In the overview editor, click the **Business Rules** navigation tab.
3. On the Business Rules page, select the view-level Triggers node, and click the **Create New Trigger** icon.
4. From the Add Trigger dialog box, select the appropriate trigger point from the Type dropdown list.
5. Enter a Groovy expression in the Trigger Definition tab.

The application creates a `<trigger>` tag in the view object's XML file and also populates the `.bcs` file associated with the view object. When a Groovy expression is evaluated at runtime, it is assumed to be untrusted unless explicitly identified as a trusted expression. To avoid the runtime exception, use the Properties window to identify the expression as a trusted expression..

12 Implementing Validation and Business Rules Programmatically

This chapter describes how to use ADF entity object events and features to programmatically implement business rules in an Oracle ADF application. It also describes how to invoke custom validation code, for example, using setter methods to populate entity rows.
This chapter includes the following sections:

About Programmatic Business Rules

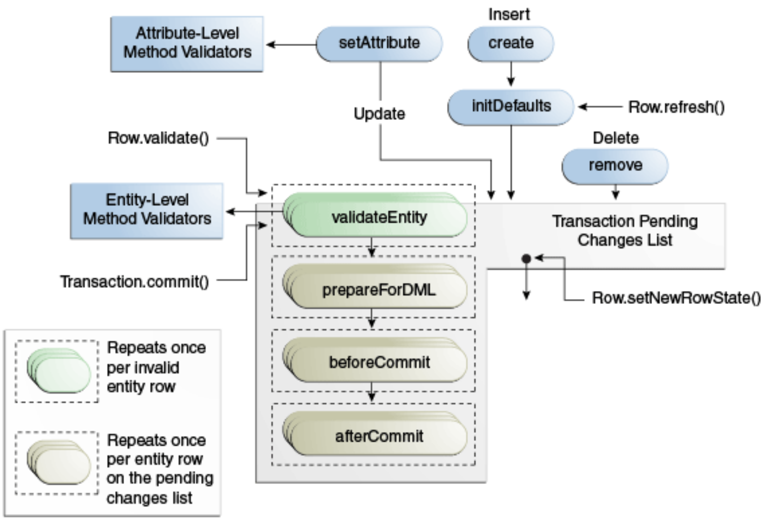
ADF Business Components provide method validators and events to programmatically define business logic in addition to using built-in declarative validation features.

Complementing the built-in *declarative* validation features, **entity objects** and **view objects** contain method validators and several events you can handle that allow you to *programmatically* implement encapsulated business logic using Java code. These concepts are illustrated in Figure 12-1.

- Attribute-level method validators trigger validation code when an attribute value changes.
- Entity-level method validators trigger validation code when an entity row is validated.
- You can override the following key methods in a custom Java class for an entity:
  - `create()`, to assign default values when a row is created

- `initDefaultExpressionAttributes()`, to assign defaults either when a row is created or when a new row is refreshed
- `remove()`, to conditionally disallow deleting
- `isAttributeUpdateable()`, to make attributes conditionally updatable
- `setAttribute()`, to trigger attribute-level method validators
- `validateEntity()`, to trigger entity-level method validators
- `prepareForDML()`, to assign attribute values before an entity row is saved
- `beforeCommit()`, to enforce rules that must consider all entity rows of a given type
- `afterCommit()`, to send notifications about a change to an entity object's state

Figure 12-1 Key Entity Objects Features and Events for Programmatic Business Logic


Attribute-Level Method Validators
setAttribute
Insert
create
Update
initDefaults
Row.refresh()
Row.validate()
Delete
remove
Entity-Level Method Validators
validateEntity
Transaction Pending Changes List
Transaction.commit()
prepareForDML
Row.setNewRowState()
beforeCommit
afterCommit
Repeats once per invalid entity row
Repeats once per entity row on the pending changes list

Note:

Only new and modified rows participate in transaction validation. Although deleted rows are part of the transaction, they are not included in the validation.

When coding programmatic business rules, it's important to have a firm grasp of the validation cycle. See Understanding the Validation Cycle.

Programmatic Business Rules Use Cases and Examples

While much of your validation can be implemented using basic declarative behavior, you can implement more complex business rules for your business domain layer when needed, using the Method validator to invoke custom validation code.

Some examples of when you might want to use programmatic business rules include:

- Eagerly defaulting an attribute value from a database sequence
- Assigning values derived from complex calculations
- Undoing pending changes to an entity object
- Accessing and storing information about the current user session
- Determining conditional updatability for attributes

Additional Functionality for Programmatic Business Rules

You may find it helpful to understand other **Oracle ADF** features before you start using programmatic validation. Following are links to other functionality that may be of interest.

- Before implementing business rules programmatically, you should see if the declarative validation can handle the needs of your application. For more information, see Defining Validation and Business Rules Declaratively.
- You can use resource bundles to provide localizable validation error messages, as described in Working with Resource Bundles.
- How to Synchronize with Trigger-Assigned Values, explains how to use the `DBSequence` type for primary key attributes whose values need to be populated by a database sequence at *commit* time.
- For information about security features in Oracle Fusion Web Applications, including how to access information about the authenticated user, see Enabling ADF Security in a Fusion Web Application.
- For information about using Groovy script in your entity object business logic, see Using Groovy Scripting Language with Business Components.

Using Method Validators

You can define custom Java code that is triggered by method validators. ADF Method validators allow you to supplement declarative validation rules. You can define attribute or entity level method validators.

Method validators are the primary way of supplementing declarative validation rules and Groovy-scripted expressions using your own Java code. Method validators trigger Java code that you write in your own validation methods at the appropriate time during the entity object validation cycle. There are many types of validation you can code with a method validator, either on an attribute or on an entity as a whole.

You can add any number of attribute-level or entity-level method validators, provided they each trigger a distinct method name in your code. All validation method names must begin with the word `validate`; however, following that rule you are free to name them in any way that most clearly identifies the functionality. For an attribute-level validator, the method must take a single argument of the same type as the entity attribute. For an entity-level validator, the method takes no arguments. The method must also be public, and must return a boolean value. Validation will fail if the method returns `false`.

> **Note:**
>
> Although it is important to be aware of these rules, when you use JDeveloper to create method validators, JDeveloper creates the correct interface for the class.

At runtime, the Method validator passes an entity attribute to a method implemented in your entity object class.

In the following example, the method accepts strings that start with a capital letter and throws an exception on null values, empty strings, and strings that do not start with a capital letter.

```
public boolean validateIsCapped(String text)
{
  if (text != null &&
      text.length() != 0 &&
      text[0] >= 'A' &&
      text[0] <= 'Z')
  {
    return true;
  }
}
```

How to Create an Attribute-Level Method Validator

The use of method validators is a programmatic approach that supplements your declarative validation rules.

Before you begin:

It may be helpful to have an understanding of what method validators are. For more information, see Using Method Validators.

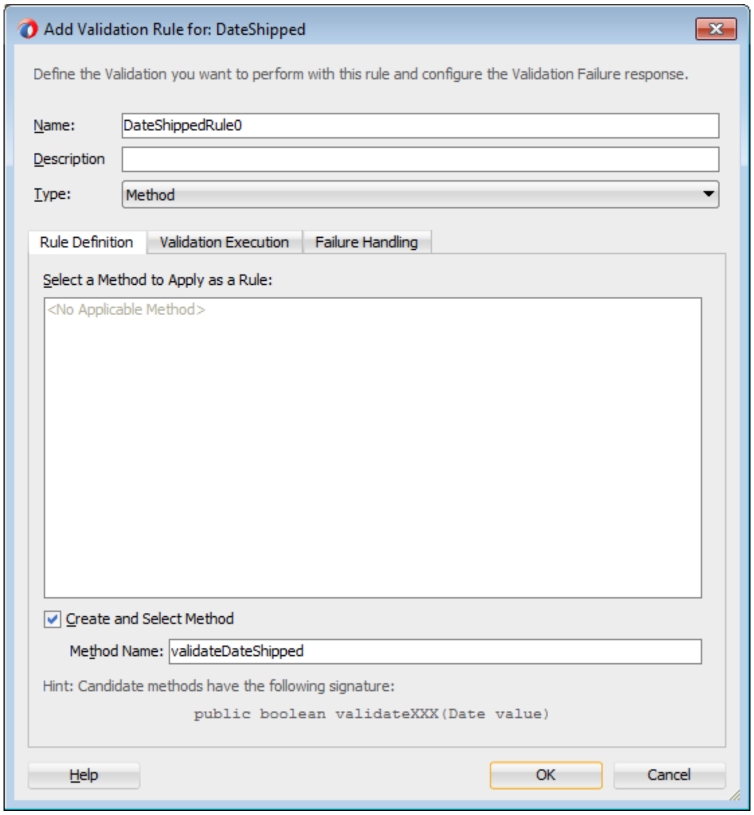
You may also find it helpful to understand additional functionality that can be added using other validation features. For more information, see Additional Functionality for Programmatic Business Rules.

To create an attribute-level Method validator:

1. In the Applications window, double-click the desired entity object.
2. In the overview editor, click the **Java** navigation tab.

   The Java page shows the Java generation options that are currently enabled for the entity object. If your entity object does not yet have a custom entity object class, then you must generate one before you can add a Method validator. To generate the custom Java class, click the **Edit** icon, then select **Generate Entity Object Class**, and click **OK** to generate the `*.java` file.
3. Click the **Business Rules** navigation tab, and then expand the **Attributes** section and select the attribute that you want to validate.
4. Click the **New** icon to add a validation rule.
5. In the Add Validation Rule dialog, select **Method** from the **Rule Type** dropdown list.

Figure 12-2 Adding an Attribute-Level Method Validator



Add Validation Rule for: DateShipped

Define the Validation you want to perform with this rule and configure the Validation Failure response.

Name: DateShippedRule0

Description:

Type: Method

Select a Method to Apply as a Rule:

<No Applicable Method>

☑ Create and Select Method

Method Name: validateDateShipped

Hint: Candidate methods have the following signature:

```
public boolean validateXXX(Date value)
```

The Add Validation Rule dialog displays the expected method signature for an attribute-level validation method. You have two choices:

- If you already have a method in your entity object's custom Java class of the appropriate signature, it will appear in the list and you can select it after deselecting the **Create and Select Method** checkbox.
- If you leave the **Create and Select Method** checkbox selected (see Figure 12-2), you can enter any method name in the **Method Name** box that begins with the word `validate`. When you click **OK,** JDeveloper adds the method to your entity object's custom Java class with the appropriate signature.

6. Optionally, click the **Validation Execution** tab to enter criteria for the execution of the rule, such as dependent attributes and a precondition expression. For more information, see Triggering Validation Execution.
7. Click the **Failure Handling** tab and enter or select the error message that will be shown to the user if the validation rule fails.

   For more information, see Creating Validation Error Messages.

What Happens When You Create an Attribute-Level Method Validator

When you add a new method validator, JDeveloper updates the XML document to reflect the new validation rule. If you asked to have the method created, the method is added to the entity object's custom Java class. The following example illustrates a simple attribute-level validation rule that ensures that the `OrderShippedDate` of an order is a date in the current month. Notice that the method accepts an argument of the same type as the corresponding attribute, and that its conditional logic is based on the value of this incoming parameter. When the attribute validator fires, the attribute value has not yet been set to the new value in question, so calling the `getOrderShippedDate()` method inside the attribute validator for the `OrderShippedDate` attribute would return the attribute's *current* value, rather than the *candidate* value that the client is attempting to set.

> **Note:**
>
> The return value of the `compareTo()` method is zero (`0`) if the two dates are equal, negative one (`-1`) if the first date is less than the second, or positive one (`1`) if the first date is greater than the second.

```
public boolean validateOrderShippedDate(Date  data) {
  if (data != null && data.compareTo(getFirstDayOfCurrentMonth()) <= 0) {
    return false;
  }
  return true;
}
```

How to Create an Entity-Level Method Validator

Entity-level method validators are similar to attribute-level method validators, except that they have a broader scope: the entire entity rather than a single attribute.

In addition, you can defer execution of the validator to time of the transaction when using an entity-level method validator.

Before you begin:

It may be helpful to have an understanding of what method validators are. For more information, see Using Method Validators.

You may also find it helpful to understand additional functionality that can be added using other validation features. For more information, see Additional Functionality for Programmatic Business Rules.

To create an entity-level method validator:

1. In the Applications window, double-click the desired entity object.
2. In the overview editor, click the **Java** navigation tab.

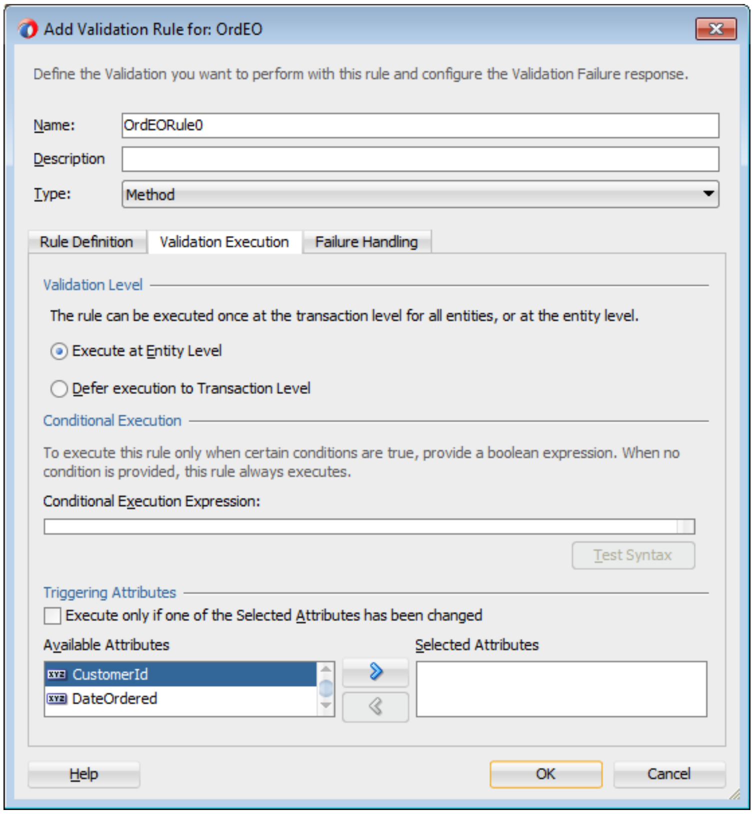
   The Java page shows the Java generation options that are currently enabled for the entity object. If your entity object does not yet have a custom entity object class, then you must generate one before you can add a Method validator. To generate the custom Java class, click the **Edit** icon, then select **Generate Entity Object Class**, and click **OK** to generate the `*.java` file.
3. Click the **Business Rules** navigation tab, and then select the **Entity** node and click the **New** icon to add a validation rule.
4. In the Add Validation Rule dialog, select **Method** from the **Rule Type** dropdown list.

The Add Validation Rule dialog displays the expected method signature for an entity-level validation method. You have two choices:

- If you already have a method in your entity object's custom Java class of the appropriate signature, it will appear in the list and you can select it after deselecting the **Create and Select Method** checkbox.
- If you leave the **Create and Select Method** checkbox selected (see Figure 12-3), you can enter any method name in the **Method Name** box that begins with the word `validate`. When you click **OK,** JDeveloper adds the method to your entity object's custom Java class with the appropriate signature.

5. Optionally, click the **Validation Execution** tab to enter criteria for the execution of the rule, such as dependent attributes and a precondition expression. For more information, see Triggering Validation Execution.

Figure 12-3 Adding an Entity-Level Method Validator



Add Validation Rule for: OrdEO

Define the Validation you want to perform with this rule and configure the Validation Failure response.

Name: OrdEORule0

Description

Type: Method

Validation Level

The rule can be executed once at the transaction level for all entities, or at the entity level.

● Execute at Entity Level

○ Defer execution to Transaction Level

Conditional Execution

To execute this rule only when certain conditions are true, provide a boolean expression. When no condition is provided, this rule always executes.

Conditional Execution Expression:

Triggering Attributes

□ Execute only if one of the Selected Attributes has been changed

Available Attributes: CustomerId, DateOrdered

Selected Attributes

For Method entity validators, you can also use the **Validation Execution** tab to specify the validation level. If you select **Defer Execution to Transaction Level**, the validator will fire when the entity is committed. This is useful when multiple rows are displayed for editing in a table.

6. Click the **Failure Handling** tab and enter or select the error message that will be shown to the user if the validation rule fails.

   For more information, see Creating Validation Error Messages.

What Happens When You Create an Entity-Level Method Validator

When you add a new method validator, JDeveloper updates the XML document to reflect the new validation rule. If you asked to have the method created, the method is added to the entity object's custom Java class. The following example illustrates a simple entity-level validation rule that ensures that the `DateShipped` of an order comes after the `DateOrdered`.

```
public boolean validateDateShippedAfterDateOrdered() {
  Date DateShipped = getDateShipped();
  Date DateOrdered  = getDateOrdered();
  if (DateShipped != null && DateShipped.compareTo(DateOrdered) < 0) {
    return false;
  }
  return true;
}
```

If you select **Defer Execution to Transaction Level** on the Validation Execution tab of the Add Validation Rule dialog and set `SkipValidation="skipDataControls"` on the page that displays the entity object, you can postpone the validation until transaction time. This allows the user to modify multiple rows in a table and then validate them as a group when the user commits. The following example shows a method validator that verifies that the department name is not null for a collection of entity rows.

```
// Validation method for Departments.
public boolean validateDepartments(ArrayList ctxList) {
   Iterator iter = ctxList.iterator();
   while (iter.hasNext()) {
      JboValidatorContext entValContext = (JboValidatorContext)iter.next();
      DepartmentsImpl deptEO = (DepartmentsImpl)entValContext.getSource();
      if (deptEO.getDepartmentName() == null) {
         // if Dept is null, throw error
         return false;
      }
   }
   return true;
}
```

What You May Need to Know About Translating Validation Rule Error Messages

Like the locale-specific UI **control hints** for entity object attributes, the validation rule error messages are added to the entity object's component message bundle file. These entries in the message bundle represent the strings for the default locale for your application. To provide translated versions of the validation error messages, follow the same steps as for translating the UI control hints, as described in Working with Resource Bundles.

Assigning Programmatically Derived Attribute Values

ADF Business Components allows you to perform programmatic defaulting when declarative defaulting is insufficient.

When declarative defaulting falls short of your needs, you can perform programmatic defaulting in your entity object:

- When an entity row is first created
- When the entity row is first created or when refreshed to null values
- When the entity row is saved to the database
- When an entity attribute value is set

How to Provide Default Values for New Rows at Create Time

The `create()` method provides the entity object event you can handle to initialize default values the first time an entity row is created. The following example shows the overridden create method of an entity object. It calls an attribute setter method to populate the `DateOrdered` attribute in a new order entity row.

You can also define default values using a Groovy expression. For more information, see How to Define a Static Default Value.

> **Note:**
>
> Calling the `setAttribute()` method inside the overridden `create()` method does not mark the new row as being changed by the user. These programmatically assigned defaults behave like declaratively assigned defaults. Also note that any changes you make to the `create()` method must occur after the call to `super.create()`.

```
// In entity object implementation class
protected void create(AttributeList nameValuePair) {
   super.create(nameValuePair);
   this.setDateOrdered(new Date());
}
```

Choosing Between create() and initDefaultExpressionAttributes() Methods

You should override the `initDefaultExpressionAttributes()` method for programmatic defaulting logic that you want to fire both when the row is first created, and when it might be refreshed back to initialized status.

If an entity row has `New` status and you call the `refresh()` method on it, then the entity row is returned to an `Initialized` status if you do not supply either the `REFRESH_REMOVE_NEW_ROWS` or `REFRESH_FORGET_NEW_ROWS` flag. As part of this process, the entity object's `initDefaultExpressionAttributes()` method is invoked, but not its `create()` method again.

Eagerly Defaulting an Attribute Value from a Database Sequence

How to Synchronize with Trigger-Assigned Values, explains how to use the `DBSequence` type for primary key attributes whose values need to be populated by a database sequence at *commit* time. Sometimes you may want to eagerly allocate a sequence number at entity row creation time so that the user can see its value and so that this value does not change when the data is saved. To accomplish this, use the `SequenceImpl` helper class in the `oracle.jbo.server` package in an overridden `create()` method as shown in the following example. It shows code from the custom Java class for an entity object. After calling `super.create()`, it creates a new instance of the SequenceImpl object, passing the sequence name and the current transaction object. Then it calls the setWarehouseId() attribute setter method with the return value from SequenceImpl's getSequenceNumber() method.

> **Note:**
>
> For a metadata-driven alternative to this approach, see Assigning the Primary Key Value Using an Oracle Sequence.

```
// In entity object implementation class
import oracle.jbo.server.SequenceImpl;
// Default WarehouseId value from WAREHOUSE_SEQ sequence at entity row create
time
protected void create(AttributeList attributeList) {
    super.create(attributeList);
    SequenceImpl sequence = new SequenceImpl("WAREHOUSE_SEQ",getDBTransaction());
    setWarehouseId(sequence.getSequenceNumber());
}
```

How to Assign Derived Values Before Saving

If you want to assign programmatic defaults for entity object attribute values before a row is saved, override the prepareForDML() method and call the appropriate attribute setter methods to populate the derived attribute values. To perform the assignment only during INSERT, UPDATE, or DELETE, you can compare the value of the operation parameter passed to this method against the integer constants DML_INSERT, DML_UPDATE, DML_DELETE respectively.

The following example shows an overridden prepareForDML() method that assigns derived values.

```
protected void prepareForDML(int operation, TransactionEvent e) {
  super.prepareForDML(operation, e);
  //Populate GL Date
  if (operation == DML_INSERT) {
    if (this.getGlDate() == null) {
      String glDateDefaultOption =
        (String)this.getInvoiceOption().getAttribute("DefaultGlDateBasis");
      if ("I".equals(glDateDefaultOption)) {
        setAttribute(GLDATE, this.getInvoiceDate());
      } else {
        setAttribute(GLDATE, this.getCurrentDBDate());
      }
    }
  }

  //Populate Exchange Rate and Base Amount if null
  if ((operation == DML_INSERT) || (operation == DML_UPDATE)) {
    BigDecimal defaultExchangeRate = new BigDecimal(1.5);
    if ("Y".equals(this.getInvoiceOption().getAttribute("UseForeignCurTrx"))) {
      if (!(this.getInvoiceCurrencyCode().equals(
                          this.getLedger().getAttribute("CurrencyCode")))) {
        if (this.getExchangeDate() == null) {
          setAttribute(EXCHANGEDATE, this.getInvoiceDate());
        }
        if (this.getExchangeRateType() == null) {
          String defaultConvRateType =
            (String)this.getInvoiceOption().getAttribute("DefaultConvRateType");
          if (defaultConvRateType != null) {
            setAttribute(EXCHANGERATETYPE, defaultConvRateType);
          } else {
            setAttribute(EXCHANGERATETYPE, "User");
          }
        }
        if (this.getExchangeRate() == null) {
          setAttribute(EXCHANGERATE, defaultExchangeRate);
        }
        if ((this.getExchangeRate() != null) &&
            (this.getInvoiceAmount() != null)) {
          setAttribute(INVAMOUNTFUNCCURR,
                     (this.getExchangeRate().multiply(this.getInvoiceAmount())));
        }
      } else {
        setAttribute(EXCHANGEDATE, null);
        setAttribute(EXCHANGERATETYPE, null);
        setAttribute(EXCHANGERATE, null);
        setAttribute(INVAMOUNTFUNCCURR, null);
      }
    }
  }
}
```

How to Assign Derived Values When an Attribute Value Is Set

To assign derived attribute values whenever another attribute's value is set, add code to the latter attribute's setter method. The following example shows the setter method for an AssignedTo attribute in an entity object.

After the call to setAttributeInternal() to set the value of the AssignedTo attribute, it uses the setter method for the AssignedDate attribute to set its value to the current date and time.

> **Note:**
>
> It is safe to add custom code to the generated attribute getter and setter methods as shown here. When JDeveloper modifies code in your class, it intelligently leaves your custom code in place.

```
public void setAssignedTo(Number value) {
  setAttributeInternal(ASSIGNEDTO, value);
  setAssignedDate(getCurrentDateWithTime());
}
```

Undoing Pending Changes to an Entity Using the Refresh Method

Use the refresh(int flag) method on an ADF Business Components row to refresh pending row-changes. The int flag argument allows different values that control the behavior of this method.

You can use the `refresh(int flag)` method on a row to refresh any pending changes it might have. The behavior of the `refresh()` method depends on the flag that you pass as a parameter. The three key flag values that control its behavior are the following constants in the `Row` interface:

- `REFRESH_WITH_DB_FORGET_CHANGES` forgets modifications made to the row in the current transaction, and the row's data is refreshed from the database. The latest data from the database replaces data in the row regardless of whether the row was modified or not.
- `REFRESH_WITH_DB_ONLY_IF_UNCHANGED` works just like `REFRESH_WITH_DB_FORGET_CHANGES`, but for *unmodified* rows. If a row was already modified by this transaction, the row is not refreshed.
- `REFRESH_UNDO_CHANGES` works the same as `REFRESH_WITH_DB_FORGET_CHANGES` for *unmodified* rows. For a modified row, this mode refreshes the row with attribute values at the beginning of this transaction. The row remains in a modified state if it had been previously posted but not committed in the current transaction prior to performing the refresh operation.

How to Control What Happens to New Rows During a Refresh

By default, any entity rows with `New` status that you `refresh()` are reverted back to blank rows in the `Initialized` state. Declarative defaults are reset, as well as programmatic defaults coded in the `initDefaultExpressionAttributes()` method, but the entity object's `create()` method is not invoked during this blanking-out process.

You can change this default behavior by combining one of the flags in Undoing Pending Changes to an Entity Using the Refresh Method with one of the following two flags (using the bitwise-`OR` operator):

- `REFRESH_REMOVE_NEW_ROWS`, new rows are removed during refresh.
- `REFRESH_FORGET_NEW_ROWS`, new rows are marked `Dead`.

How to Cascade Refresh to Composed Children Entity Rows

You can cause a `refresh()` operation to cascade to composed child entity rows by combining the `REFRESH_CONTAINEES` flag (using the bitwise-`OR` operator) with any of the valid flag combinations described in Undoing Pending Changes to an Entity Using the Refresh Method and How to Control What Happens to New Rows During a Refresh. This causes the entity to invoke `refresh()` using the same mode on any composed child entities it contains.

Using View Objects for Validation

You can manage pending changes for ADF entity-based view objects in the entity cache, while read-only view objects only retrieve data.

When your business logic requires performing SQL queries, the natural choice is to use a view object to perform that task. Keep in mind that the SQL statements you execute for validation will "see" pending changes in the entity cache only if they are entity-based view objects. Read-only view objects will only retrieve data that has been posted to the database.

How to Use View Accessors for Validation Against View Objects

Since entity objects are designed to be reused in any application scenario, they should not depend *directly* on a view object instance in the data model of any specific **application module**. Doing so would prevent them from being reused in other application modules, which is highly undesirable.

Instead, you should use a **view accessor** to validate against a view object. For more information, see How to Create a View Accessor for an Entity Object or View Object.

Using a view accessor, your validation code can access the view object and set bind variables, as shown in the following example.

```
// Sample entity-level validation method
public boolean validateSomethingUsingViewAccessor() {
  RowSet rs = getMyValidationVO();
  rs.setNamedBindParameter("Name1", value1);
  rs.setNamedBindParameter("Name2", value2);
  rs.executeQuery();
  if ( /* some condition */) {
  /*
   * code here returns true if the validation succeeds
   */
  }
  return false;
}
```

Best Practice:

Any time you access a row set programmatically, you should consider creating a secondary iterator for the row set. This ensures that you will not disturb the current row set of the default **row set iterator** that may be utilized when your expose your view objects as **data controls** to the user interface project. You can call `createRowSetIterator()` on the row set you are working with to create a secondary named row set iterator. When you are through with programmatic iteration, your code should call `closeRowSetIterator()` on the row set to remove the secondary iterator from memory.

As the sample code suggests, view objects used for validation typically have one or more named bind variables in them. In this example, the bind variables are set using the `setNamedBindParameter()` method. However, you can also set these variables declaratively in JDeveloper using Groovy expressions in the view accessor definition page.

Depending on the kind of data your view object retrieves, the "`/* some condition */`" expression in the example will look different. For example, if your view object's SQL query is selecting a `COUNT()` or some other aggregate, the condition will typically use the `rs.first()` method to access the first row, then use the `getAttribute()` method to access the attribute value to see what the database returned for the count.

If the validation succeeds or fails based on whether the query has returned zero or one row, the condition might simply test whether `rs.first()` returns `null` or not. If `rs.first()` returns `null`, there is no "first" row. In other words, the query retrieved no rows. In other cases, you may be iterating over one or more query results retrieved by the view object to determine whether the validation succeeds or fails.

How to Validate Conditions Related to All Entities of a Given Type

The `beforeCommit()` method is invoked on *each* entity row in the pending changes list after the changes have been posted to the database, but before they are committed. This can be a useful method in which to execute view object-based validations that must assert some rule over all entity rows of a given type.

> **Note:**
>
> You can also do this declaratively using a transaction-level validator (see How to Set Transaction-Level Validation).

If your `beforeCommit()` logic can throw a `ValidationException`, you must set the `jbo.txn.handleafterpostexc` property to `true` in your configuration to have the framework automatically handle rolling back the in-memory state of the other entity objects that may have already successfully posted to the database (but not yet been committed) during the current commit cycle.

For example, consider the overridden `beforeCommit()` shown in the following example. In this example, there are three view objects based on polymorphic entity objects (`Persons`, `Staff`, and `Supplier`), with the `PersonTypeCode` attribute as the discriminator. The `PersonsImpl.java` file has an overridden `beforeCommit()` method that calls a validation method. The validation method uses the fourth view object, `PersonsValidator`, to make sure that the principal name is unique across each person type. For example, there is a `PrincipalName` of `SKING` for the `Staff` view object, but there cannot be another `SKING` in this or the other person types.

```
// In entity object implementation class
. . .
@Override
public void beforeCommit(TransactionEvent transactionEvent) throws
ValidationException {
  String principalName = getPrincipalName();
  if (!validatePrincipalNameIsUniqueUsingViewAccessor(principalName)) {
    throw new ValidationException("Principal Name must be unique across person
types");
  }
    super.beforeCommit(transactionEvent);
}

public boolean validatePrincipalNameIsUniqueUsingViewAccessor(String
principalName) {
RowSet rs = getPersonsValidatorVO();
rs.setNamedWhereClauseParam("principalName", principalName);
rs.setRangeSize(-1);
rs.executeQuery();
Row[] validatorRows = rs.getAllRowsInRange();
if (validatorRows.length > 1)
  // more than one row has the same princpalName
{
    return false;
}
rs.closeRowSetIterator();
return true;
}
```

What You May Need to Know About Row Set Access with View Accessors

If your entity object or view object business logic iterates over its own view accessor row set, and that view accessor is not also used by a model-defined List of Values, then there is no need to use a secondary row set iterator. For example, if an entity object has a view accessor named `AirportValidationVA` for a view object that takes one named bind parameter, it can iterate its own view accessor row set using either Groovy script or Java. The following example shows Groovy script that iterates over a view accessor row set.

```
AirportValidationVA.setNamedWhereClauseParam("VarTla",newValue)
AirportValidationVA.executeQuery();
return AirportValidationVA.first() != null;
```

The following example shows a Java method validator that iterates over a view accessor row set.

```
public boolean validateJob(String job) {
   getAirportValidationVA().setNamedWhereClauseParam("VarTla",job);
   getAirportValidationVA().executeQuery();
   return getAirportValidationVA().first() != null;
}
```

Accessing Related Entity Rows Using Association Accessors

Use an ADF entity object’s custom Java class’s association accessor method to access information from related entity objects.

To access information from related entity objects, you use an association accessor method in your entity object's custom Java class. By calling the accessor method, you can easily access any related entity row — or set of entity rows — depending on the cardinality of the association.

How to Access Related Entity Rows

You can use an association accessor to access related entity rows. The following example shows code from the `controlpostorder` module in the `SummitADF` application workspace that shows the overridden `postChanges()` method in the `EmpEO` entity object's custom Java class. It uses the `getDeptEO()` association accessor to retrieve the related department for the employee.

```
// In EmpEOImpl.java
public void postChanges(TransactionEvent transactionEvent) {
   /* If current entity is new or modified */
   if (getPostState() == STATUS_NEW ||
      getPostState() == STATUS_MODIFIED) {
      /* Get the associated dept for the employee */
      DeptEOImpl dept = getDeptEO();
      /* If there is an associated dept */
```

```
        if (dept != null) {
           /* And if it's post-status is NEW */
           if (dept.getPostState() == STATUS_NEW) {
              /*
     * Post the department first, before posting this
     * entity by calling super below
     */
              dept.postChanges(transactionEvent);
           }
        }
     }
     super.postChanges(transactionEvent);
  }
```

How to Access Related Entity Row Sets

If the cardinality of the association is such that multiple rows are returned, you can use the association accessor to return sets of entity rows.

The following example illustrates the code for the overridden `postChanges()` method in the `DeptEO` entity object's custom Java class. It shows the use of the `getEmpEO()` association accessor to retrieve the `RowSet` object of `EmpEO` rows in order to update the `DeptId` attribute in each row using the `setDeptId()` association accessor.

```
// In DeptEOImpl.java in the controlpostorder module
// of the SummitADF application workspace
RowSet newEmployeesBeforePost = null;
@Override
public void postChanges(TransactionEvent transactionEvent) {
      /* Update references only if Department is a NEW one */
   if (getPostState() == STATUS_NEW) {
      /*
      * Get a rowset of employees related
      * to this new department before calling super
      */
   newEmployeesBeforePost = (RowSet)getEmpEO();
   }
   super.postChanges(transactionEvent);
    }
@Override
protected void refreshFKInNewContainees() {
   if (newEmployeesBeforePost != null) {
      Number newDeptId = getId().getSequenceNumber();
         /*
         * Process the rowset of employees that referenced
         * the new department prior to posting, and update their
         * Id attribute to reflect the refreshed Id value
         * that was assigned by a database sequence during posting.
         */
      while (newEmployeesBeforePost.hasNext()){
         EmpEOImpl emp = (EmpEOImpl)newEmployeesBeforePost.next();
         emp.setDeptId(newDeptId);
      }
      closeNewProductRowSet();
   }
}
```

Referencing Information About the Authenticated User

Use the Configure ADF Security wizard to enable the ADF authentication servlet to trigger user login and logout by the web application container. You can also retrieve authenticated user credentials.

If you have run the Configure ADF Security wizard on your application to enable the ADF authentication servlet to support user login and logout, the `oracle.jbo.server.SessionImpl` object provides methods you can use to get information about the name of the authenticated user and about the roles of which they are a member. This is the implementation class for the `oracle.jbo.Session` interface that clients can access.

For information about how to access information about the authenticated user, see How to Determine Membership of a Java EE Security Role.

For more information about security features in Oracle Fusion Web Applications, read Enabling ADF Security in a Fusion Web Application.

Accessing Original Attribute Values

Use the `getPostedAttribute()` method to retrieve an ADF entity object attribute's original value before it was changed in entity row in the current transaction.

If an entity attribute's value has been changed in the current transaction, when you call the attribute getter method for it you will get the pending changed value. Sometimes you want to get the original value before it was changed. Using the `getPostedAttribute()` method, your entity object business logic can consult the original value for any attribute as it was read from the database before the entity row was modified. This method takes the attribute *index* as an argument, so pass the appropriate generated attribute index enums that JDeveloper maintains for you.

Storing Information About the Current User Session

Use the user data hash table provided the Session object to store current user session information that the ADF entity object business logic can utilize.

If you need to store information related to the current user session in a way that entity object business logic can reference, you can use the user data hash table provided by the `Session` object.

How to Store Information About the Current User Session

When a new user accesses an application module for the first time, the `prepareSession()` method is called. As shown in the following example, the application module overrides `prepareSession()` to retrieve information about the authenticated user by calling a `retrieveUserInfoForAuthenticatedUser()` method on the view object instance. Then, it calls the `setUserIdIntoUserDataHashtable()` helper method to save the user's numerical ID into the user data hash table.

```
// In the application module
protected void prepareSession(Session session) {
  super.prepareSession(session);
  /*
   * Query the correct row in the VO based on the currently logged-in
   * user, using a custom method on the view object component
   */
  getLoggedInUser().retrieveUserInfoForAuthenticatedUser();
  setUserIdIntoUserDataHashtable();
}
```

The following example shows the code for the view object's `retrieveUserInfoForAuthenticatedUser()` method. It sets its own `EmailAddress` bind variable to the name of the authenticated user from the session and then calls `executeQuery()` to retrieve the additional user information from the `USERS` table.

```
// In the view object's custom Java class
public void retrieveUserInfoForAuthenticatedUser() {
  SessionImpl session = (SessionImpl)getDBTransaction().getSession();
  setEmailAddress(session.getUserPrincipalName());
  executeQuery();
  first();
}
```

One of the pieces of information about the authenticated user that the view object retrieves is the user's numerical ID number, which that method returns as its result. For example, the user `sking` has the numeric `UserId` of `300`.

The next example shows the `setUserIdIntoUserDataHashtable()` helper method — used by the `prepareSession()` code in the sample above— that stores this numerical user ID in the user data hash table, using the key provided by the string constant `CURRENT_USER_ID`.

```
// In the application module
private void setUserIdIntoUserDataHashtable() {
  Integer userid = getUserIdForLoggedInUser();
  Hashtable userdata = getDBTransaction().getSession().getUserData();
  userdata.put(CURRENT_USER_ID, userid);
}
```

The corresponding entity objects in this example can have an overridden `create()` method that references this numerical user ID using a helper method like the one in the following example to set the `CreatedBy` attribute programmatically to the value of the currently authenticated user's numerical user ID.

```
protected Number getCurrentUserId() {
  Hashtable userdata = getDBTransaction().getSession().getUserData();
  Integer userId = (Integer)userdata.get(CURRENT_USER_ID);
  return userdata != null ? Utils.intToNumber(userId):null;
}

// In the application module
protected void prepareSession(Session session) {
  super.prepareSession(session);
  /*
   * Query the correct row in the VO based on the currently logged-in
   * user, using a custom method on the view object component
   */
  getLoggedInUser().retrieveUserInfoForAuthenticatedUser();
  setUserIdIntoUserDataHashtable();
}
```

How to Use Groovy to Access Information About the Current User Session

The top-level `adf` object allows you access to objects that the framework makes available to Groovy script. The `adf.userSession` object returns a reference to the ADF Business Components user session, which you can use to reference values in the `userData` hash map that is part of the session.

The following example shows the Groovy script you would use to reference a `userData` hash map key named `MyKey`.

```
adf.userSession.userData.MyKey
```

Accessing the Current Date and Time

ADF Business Components includes pre-defined Groovy expressions that you can use in entity object business logic to retrieve current date and time.

You might find it useful to reference the current date and time in your entity object business logic. You can reference the current date or current date and time using the following Groovy script expressions:

- `adf.currentDate` — returns the current date (time truncated)
- `adf.currentDateTime` — returns the current date and time

For information about using Groovy script in your entity object business logic, see Using Groovy Scripting Language with Business Components.

Sending Notifications Upon a Successful Commit

Upon successful commit of pending row changes, the `afterCommit()` method is called on such ADF Business Components entity rows. Use this method to send a notification as well.

The `afterCommit()` method is invoked on *each* entity row that was in the pending changes list and got successfully saved to the database. You can use this method to send a notification on a commit.

A better way to send notifications upon a successful commit is by declaring a business event. For information on how to create a business event, see Creating Business Events.

Conditionally Preventing an Entity Row from Being Removed

Use the `remove()` method to control removal of an ADF entity row.

Before an entity row is removed, the `remove()` method is invoked on an entity row. You can throw a `JboException` in the `remove()` method to prevent a row from being removed if the appropriate conditions are not met.

For example, you can add a test in the `remove()` method that determines the state of the entity object and allows the removal only if it is a new record. The following example illustrates this technique.

Note:

The entity object offers declarative prevention of deleting a master entity row that has existing, composed children rows. You configure this option on the Relationship page of the overview editor for the association.

```
// In entity object custom Java class
private boolean isDeleteAllowed() {
    byte s = this.getEntityState();
  return s==STATUS_NEW;
}

/**
 * Add entity remove logic in this method.
 */
public void remove() {
  if (isDeleteAllowed())
    super.remove();
  else
      throw new JboException("Delete not allowed in this view");
}
```

Determining Conditional Updatability for Attributes

Check if a given ADF entity object attribute is `updateable` or not at runtime by overriding the `isAttributeUpdateable()` method in the entity object class.

You can override the `isAttributeUpdateable()` method in your entity object class to programmatically determine whether a given attribute is updatable or not at runtime based on appropriate conditions.

The following example shows how an entity object can override the `isAttributeUpdateable()` method to enforce that its `PersonTypeCode` attribute is updatable only if the current authenticated user is a staff member. Notice that when the entity object fires this method, it passes in the integer attribute index whose updatability is being considered.

You can implement conditional updatability logic for a particular attribute inside an `if` or `switch` statement based on the attribute index. Here `PERSONTYPECODE` is referencing the integer attribute index enums that JDeveloper maintains in your entity object custom Java class.

> **Note:**
>
> Entity-based view objects inherit this conditional updatability as they do everything else encapsulated in your entity objects. Should you need to implement this type of conditional updatability logic in a way that is specific to a transient view object attribute, or to enforce some condition that involves data from multiple entity objects participating in the view object, you can override this same method in a view object's view row class to achieve the desired result.

```
// In the entity object custom Java class
public boolean isAttributeUpdateable(int index) {
  if (index == PERSONTYPECODE) {
    if (!currentUserIsStaffMember()) {
      return super.isAttributeUpdateable(index);
    }
    return CUSTOMER_TYPE.equals(getPersonTypeCode()) ? false : true;
  }
  return super.isAttributeUpdateable(index);
}
```

Implementing Custom Validation Rules

ADF Business Components allows you to create custom validation rule classes that capture common validation code and use it along with built-in declarative validation rules.

ADF Business Components comes with a base set of built-in declarative validation rules that you can use. However, a powerful feature of the validator architecture for entity objects is that you can create your own custom validation rules. When you notice that you or your team are writing the same kind of validation code over and over, you can build a custom validation rule class that captures this common validation "pattern" in a parameterized way.

After you've defined a custom validation rule class, you can register it in JDeveloper so that it is as simple to use as any of the built-in rules. In fact, you can even bundle your custom validation rule with a custom UI panel that JDeveloper leverages to facilitate developers' using and configuring the parameters your validation rule might require.

How to Create a Custom Validation Rule

To write a custom validation rule for entity objects, you need a Java class that implements the `JboValidatorInterface` in the `oracle.jbo.rules` package. You can create a skeleton class from the New Gallery.

Before you begin:

It may be helpful to have an understanding of custom validation rules. For more information, see Implementing Custom Validation Rules.

You may also find it helpful to understand additional functionality that can be added using other validation features. For more information, see Additional Functionality for Programmatic Business Rules.

To create a custom validator:

1. In the Applications window, right-click the project in which you want to create the validator and choose **New > From Gallery**.
2. In the New Gallery, expand **Business Tier**, select **ADF Business Components** and then **Validation Rule**, and click **OK**.
3. In the Create Validation Rule Class dialog, enter a name and package for the rule.
4. Enter a display name and a description, and click **OK**.

As shown in the following example, `JBOValidatorInterface` contains one main `validate()` method, and a getter and setter method for a `Description` property.

```
package oracle.jbo.rules;
public interface JboValidatorInterface {
  void validate(JboValidatorContext valCtx) { }
  java.lang.String getDescription() { }
  void setDescription(String description) { }
}
```

If the behavior of your validation rule will be parameterized to make it more flexible, then add additional bean properties to your validator class for each parameter. For example, the code in the following example implements a custom validation rule called `DateMustComeAfterRule` which validates that one date attribute must come after another date attribute. To allow the developer using the rule to configure the names of the date attributes to use as the initial and later dates for validation, this class defines two properties `initialDateAttrName` and `laterDateAttrName`.

The following example shows the code that implements the custom validation rule. It extends the `AbstractValidator` to inherit support for working with the entity object's custom message bundle, where JDeveloper saves the validation error message when a developer uses the rule in an entity object.

The `validate()` method of the validation rule gets invoked at runtime whenever the rule class should perform its functionality. The code performs the following basic steps:

1. Ensures validator is correctly attached at the entity level.
2. Gets the entity row being validated.
3. Gets the values of the initial and later date attributes.
4. Validate that initial date is before later date.
5. Throws an exception if the validation fails.

For easier reuse of your custom validation rules, you would typically package them into a JAR file for reference by applications that make use of the rules.

```
// package and imports omitted
public class DateMustComeAfterRule extends AbstractValidator
       implements JboValidatorInterface {
  /**
   * This method is invoked by the framework when the validator should do its job
   */
  public void validate(JboValidatorContext valCtx) {
    // 1. If validator is correctly attached at the entity level...
    if (validatorAttachedAtEntityLevel(valCtx)) {
      // 2. Get the entity row being validated
      EntityImpl eo = (EntityImpl)valCtx.getSource();
      // 3. Get the values of the initial and later date attributes
      Date initialDate = (Date) eo.getAttribute(getInitialDateAttrName());
      Date laterDate = (Date) eo.getAttribute(getLaterDateAttrName());
      // 4. Validate that initial date is before later date
      if (!validateValue(initialDate,laterDate)) {
        // 5. Throw the validation exception
        RulesBeanUtils.raiseException(getErrorMessageClass(),
                                      getErrorMsgId(),
                                      valCtx.getSource(),
                                      valCtx.getSourceType(),
                                      valCtx.getSourceFullName(),
                                      valCtx.getAttributeDef(),
                                      valCtx.getNewValue(),
                                      null, null);
      }
    }
    else {
      throw new RuntimeException("Rule must be at entity level");
    }
  }
  /**
   * Validate that the initialDate comes before the laterDate.
   */
  private boolean validateValue(Date initialDate, Date laterDate) {
    return (initialDate == null) || (laterDate == null) ||
    (initialDate.compareTo(laterDate) < 0);
  }
  /**
   * Return true if validator is attached to entity object
   * level at runtime.
   */
  private boolean validatorAttachedAtEntityLevel(JboValidatorContext ctx) {
    return ctx.getOldValue() instanceof EntityImpl;
  }
  // NOTE: Getter and Setter Methods omitted
  private String description;
  private String initialDateAttrName;
  private String laterDateAttrName;
}
```

How to Register and Use a Custom Rule in JDeveloper

After you've created a custom validation rule, you can add it to the project or application level in the JDeveloper IDE so that other developers can use the rule declaratively.

Registering a Custom Validator at the Project Level

When you register a custom validation rule at the project level, you can use it within the project.

Before you begin:

It may be helpful to have an understanding of custom validation rules. For more information, see Implementing Custom Validation Rules.

You may also find it helpful to understand additional functionality that can be added using other validation features. For more information, see Additional Functionality for Programmatic Business Rules.

To register a custom validation rule, you must have already created it, as described in How to Create a Custom Validation Rule.

To register a custom validation rule in a project containing entity objects:

1. In the Applications window, right-click the project containing the entity objects and choose **Project Properties**.
2. In the Project Properties dialog, expand **ADF Business Components** and select **Registered Rules**.
3. On the Registered Rules page, click **Add**.
4. In the Register Validation Rule dialog, browse to find the validation rule you have created and click **OK**.

Registering a Custom Validator at the IDE Level

When you register a custom validation rule at the IDE level for JDeveloper, you can use it in other projects as well as your current project.

Before you begin:

It may be helpful to have an understanding of custom validation rules. For more information, see Implementing Custom Validation Rules.

You may also find it helpful to understand additional functionality that can be added using other validation features. For more information, see Additional Functionality for Programmatic Business Rules.

To register a custom validation rule, you must have already created it, as described in How to Create a Custom Validation Rule.

To register a custom validator at the IDE level:

1. From the main menu, choose **Tools > Preferences**.
2. In the Preferences dialog, expand **ADF Business Components** and select **Register Rules**.
3. From the Register Rules page, you can add a one or more validation rules.

   When adding a validation rule, provide the fully qualified name of the validation rule class, and supply a validation rule name that will appear in JDeveloper's list of available validators.

13
Implementing Business Services with Application Modules

This chapter describes how to create ADF application modules that encapsulate the data model of an Oracle ADF application derived from a JDBC data source. This chapter also describes how to combine business service methods with that data model to implement a complete business service.

This chapter includes the following sections:

About Application Modules

An ADF application module encapsulates end-user tasks as logical units of work. User interface clients use the application module to manage application data.

An **application module** is the transactional component that UI clients use to work with application data. The application module is an ADF business component that encapsulates the business service methods and data model for a logical unit of work related to an end-user task.

In the early phases of application development, architects and designers often use UML use case techniques, as well as **ADF task flows**, to create a high-level description of the application's planned end-user functionalities. Each high-level, end-user use case identified during the design phase typically depends on:

- The domain business objects involved. To answer the question, "What core business data is relevant to the use case?"
- The user-oriented view of business data required. To answer the questions, "What subset of columns, what filtered set of rows, sorted in what way, grouped in what way, is needed to support the use case?"

The identified domain objects involved in each use case help you identify the required **entity objects** from your business domain layer. The user-oriented view of the required business data helps to define the right SQL queries captured as view objects and to retrieve the data in the exact way needed by the end user. For best performance, this includes retrieving the minimum required details necessary to support the use case. In addition to leveraging **view object** queries and **view criteria** to shape the data, you've learned how to use a **view link** to set up a master-detail hierarchy in your data model to match exactly the kind of end-user experience you want to offer the user to accomplish the use case.

The application module is the "work unit" container that includes instances of the reusable view objects required for the use case in question. These view object instances are related through metadata to the underlying entity objects in your reusable business domain layer where the end-user use cases determine the information being presented or modified.

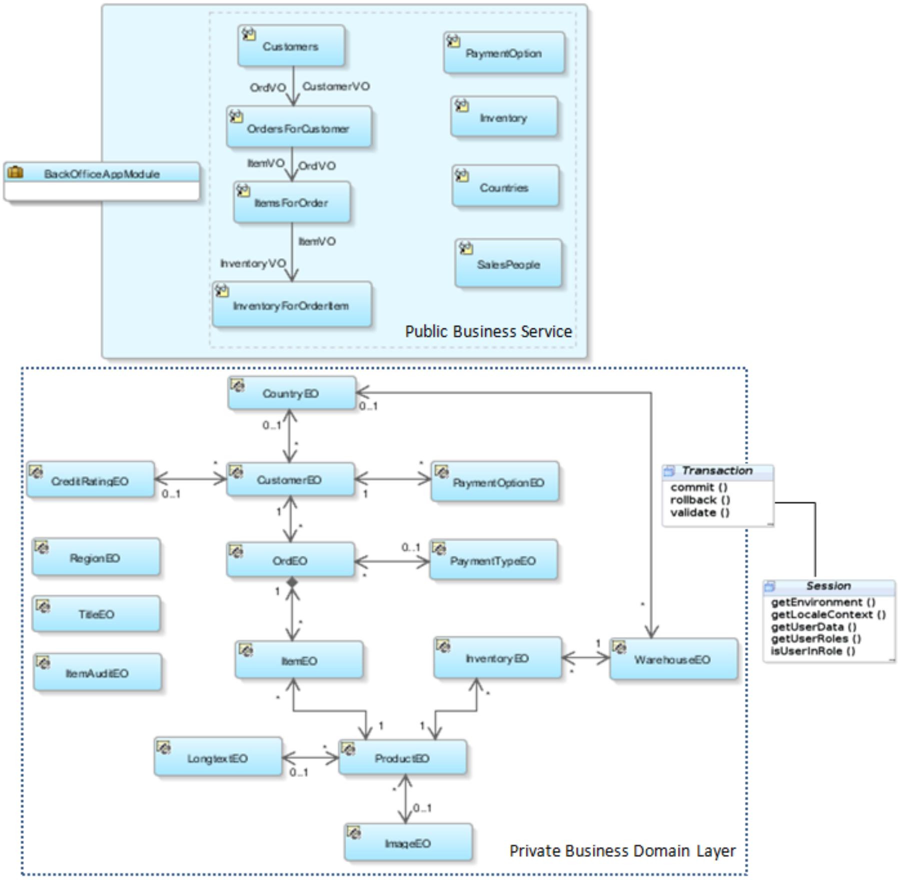
Application Module Use Cases and Examples

This chapter illustrates the following concepts illustrated in Figure 13-1, and more:

- You expose instances of view objects in an application module to define its data model.
- You write service methods to encapsulate task-level business logic.
- You expose selected methods on the client interface for UI clients to call.
- You expose selected methods on the service interface for programmatic use in application integration scenarios.

  Note that starting in release 12c, support for EJB Session Bean-enabled application modules in an **ADF Business Components** model project has been deprecated in Oracle JDeveloper. It is no longer possible to create or run EJB Session Bean-enabled application modules in JDeveloper. As an alternative for existing EJB Session Bean-enabled application modules, you may want to migrate application modules services to Web Service-enabled application modules, as described in Creating SOAP Web Services with Application Modules. This approach is consistent with the best practice for exposing service methods using external services.
- You use the **application module instance** from a pool during a logical transaction that can span multiple web pages or views.
- Your application module works with a `Transaction` object that acquires a database connection and coordinates saving or rolling back changes made to entity objects.
- The related `Session` object provides runtime information about the current application user.

Figure 13-1 Application Module Is a Business Service Component Encapsulating a Unit of Work



Additional Functionality for Application Modules

You may find it helpful to understand other **Oracle ADF** features before you start working with application modules. Following are links to other functionality that may be of interest.

- For details about creating **shared application module**s, see Sharing Application Module View Instances.
- For details about exposing ADF Business Components service methods using external services, see Creating SOAP Web Services with Application Modules.
- For details about how the **Data Controls panel** exposes the application module to UI developers, see Using ADF Model in a Fusion Web Application.
- For details about configuring application module instances to improve runtime performance, see Using State Management in a Fusion Web Application and Tuning Application Module Pools.
- For details about how to maximize the performance and scalability in the production environment, see the "Tuning Oracle Application Development Framework (ADF)" chapter in *Tuning Performance*.

- For a quick reference to the most common code that you will typically write, use, and override in your custom application module classes, see Most Commonly Used ADF Business Components Methods.
- For API documentation related to the `oracle.jbo` package, see the following Javadoc reference document:
  - *Java API Reference for Oracle ADF Model*

Creating and Modifying an Application Module

You create a single or multiple ADF application modules depending on the size of the application in terms of user tasks.

In a large application, you typically create one application module to support each coarse-grained end-user task. In a smaller-sized application, you may decide that creating a single application module is adequate to handle the needs of the complete set of application functionality. Defining Nested Application Modules provides additional guidance on this subject.

How to Create an Application Module

Any view object you create is a reusable component that can be used in the context of one or more application modules. Each view object performs the query it encapsulates in the context of that application module's transaction. The set of view object instances used by an application module defines its **data model**, in other words, the set of data that a client can display and manipulate through a user interface.

To add an application module to your existing ADF Business Components project, use the Create Application Module wizard, which is available in the New Gallery.

Before you begin:

It may be helpful to have an understanding of application modules. For more information, see Creating and Modifying an Application Module.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Application Modules.

You will need to complete this task:

> Create the desired view objects, as described in How to Create an Entity-Based View Object, and How to Create a Custom SQL Mode View Object.

To manually create an application module:

1. In the Applications window, right-click the project in which you want to create the application module and choose **New** and **Application Module**.
2. In the Create Application Module wizard, on the Name page, provide a package name and an application module name. Click **Next**.

> **Note:**
>
> In Fusion web applications, the reserved words `data`, `bindings`, `security`, and `adfContext` must not be used to name your application module. Also, avoid using the "_" (underscore) at the beginning of the name. For more information, see How to Edit an Existing Application Module.

3. On the Data Model page, include instances of the view objects you have previously defined and edit the view object instance names to be exactly what you want clients to see in the Data Controls panel. Then click **Next**.
4. On the Java page, you can optionally generate the Java files that allow you to programmatically customize the behavior of the application module or to expose methods on the application module's client interface that can be called by clients. To generate an XML-only application module component, leave the fields unselected and click **Finish**.

   Initially, you may want to generate only the application module XML definition component. After you complete the wizard, you can subsequently use the overview editor to generate the application module class files when you require programmatic access. For details about the programmatic use of the application module, see Customizing an Application Module with Service Methods.

For more step by step details, see Adding Master-Detail View Object Instances to an Application Module.

What Happens When You Create an Application Module

When you create an application module, JDeveloper creates the XML document file that represents its declarative settings and saves it in the directory that corresponds to the name of its package. For example, in the Summit ADF sample application the application module named `BackOfficeAppModule` in the `summit.model` package has the XML file `./summit/model/BackOfficeAppModule.xml` under the project's source path. This XML file contains the information needed at runtime to re-create the view object instances in the application module's data model.

If you are curious to view its contents, you can see the XML file for the application module by double-clicking the application module node in the Applications window to open the overview editor. In the editor window, click the **Source** tab to view the XML so that you can inspect it. The Structure window shows the structure of the XML file.

When you create business components, JDeveloper automatically creates a data control that contains all the functionality of the application module. **Data controls** are an ADF Model abstraction layer that provides supplemental metadata to describe the application module's operations and data collections (row sets of view object instances), including information about the attributes, methods, and types involved. Developers can then use the representation of the data control displayed in JDeveloper's **Data Controls panel** to create UI components that are automatically bound to the application module. At runtime, the **ADF Model** layer reads the metadata describing the data controls and bindings from appropriate XML files and implements the two-way connection between the user interface and the business service.

For example, the `BackOfficeAppModule` application module implements the business service layer of the `SummitADF` application workspace. Its data model contains numerous view object instances, including several master-detail hierarchies. The view layer of the core Summit ADF sample application consists of JSF pages whose UI components are bound to data from the view object instances in the `BackOfficeAppModule`'s data model, and to built-in operations and service methods on its client interface. For details about how the Data Controls panel exposes the application module to UI developers, see Exposing Application Modules with ADF Data Controls.

How to Add a View Object Instance to an Application Module

You can add a view object instance to an application module as you create the application module with the Create Application Module wizard, or you can add it later to an already created application module.

For information about using the Create Application Module wizard, see How to Create an Application Module.

Adding a View Object Instance to an Existing Application Module

You can add a view object instance to an application module that you have already created. To add a view object instance to an existing application module, and optionally, customize the view object instance, use the Data Model page of the overview editor for the application module.

Before you begin:

It may be helpful to have an understanding of application modules. For more information, see Creating and Modifying an Application Module.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Application Modules.

You will need to complete this task:

Create the desired view objects, as described in How to Create an Entity-Based View Object, and How to Create a Custom SQL Mode View Object.

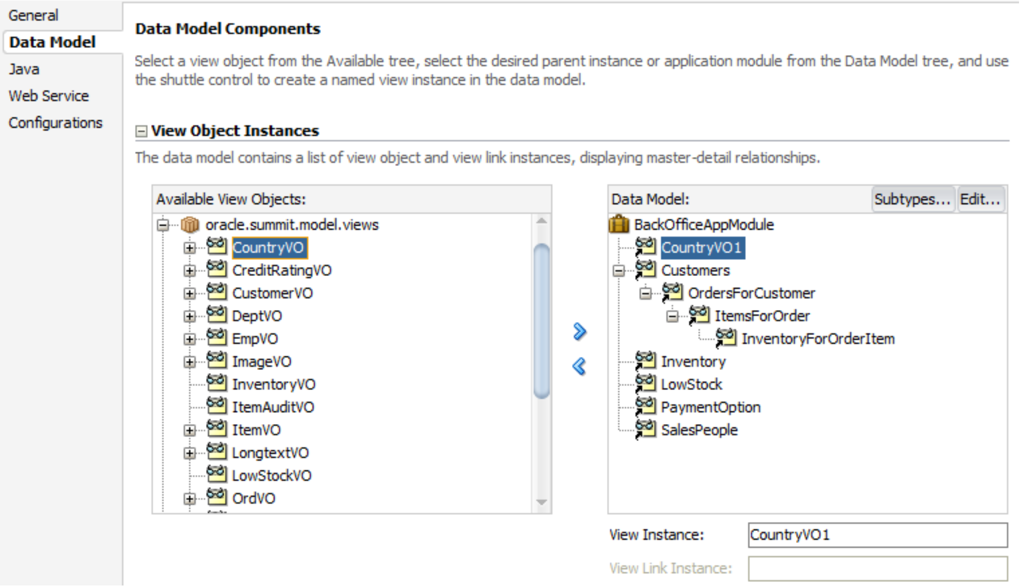
To add a view object instance to an existing application module:

1. In the Applications window, double-click the application module for which you want to define a new view instance.
2. In the overview editor, click the **Data Model** navigation tab.
3. On the Data Model Components page, expand the **View Object Instances** section and, in the **Available View Objects** list, select the view object you want to add.

   The **New View Instance** field below the list shows the name that will be used to identify the next instance of that view object that you add to the data model.
4. With the desired view object selected, shuttle the view object to the **Data Model** list.

   For example, Figure 13-2 shows the `CountryVO` view object in the `SummitADF` workspace shuttled to the data model, where it appears as `CountryVO1`.

Figure 13-2   Data Model Displays Added View Instances



Data Model Components

Select a view object from the Available tree, select the desired parent instance or application module from the Data Model tree, and use the shuttle control to create a named view instance in the data model.

View Object Instances

The data model contains a list of view object and view link instances, displaying master-detail relationships.

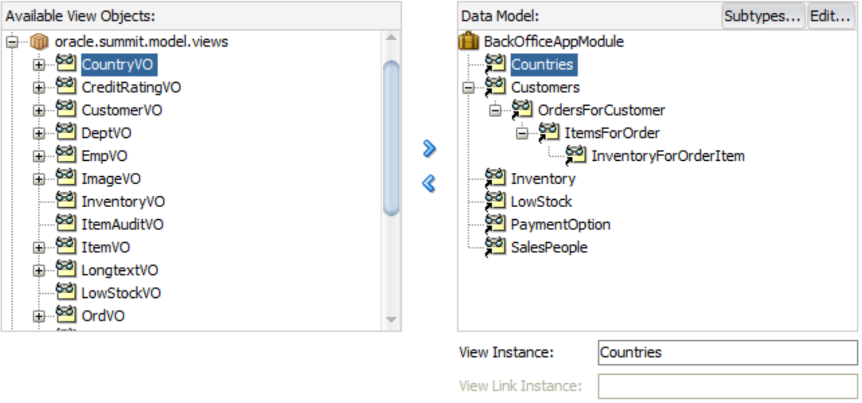
View Instance: CountryVO1

View Link Instance:

5. To change the name of the newly created view instance, enter a different name in the **New View Instance** field.

   For example, you might change the view instance name `CountryVO1` to `Countries`, as shown in Figure 13-3.

Figure 13-3   Overview Editor Displays View Instance Renamed



View Instance: Countries

View Link Instance:

Adding Master-Detail View Object Instances to an Application Module

You can use the data model that the application module overview editor displays to create a hierarchy of view instances, based on existing view links that your project defines. If you have defined view links that establish more than one level of master-detail hierarchy, then you can proceed to create as many levels of master-detail view instances as your application supports.

Before you begin:

It may be helpful to have an understanding of application modules. For more information, see Creating and Modifying an Application Module.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Application Modules.

You will need to complete this task:

> Create hierarchical relationships between view objects, as described in Working with Multiple Tables in a Master-Detail Hierarchy.

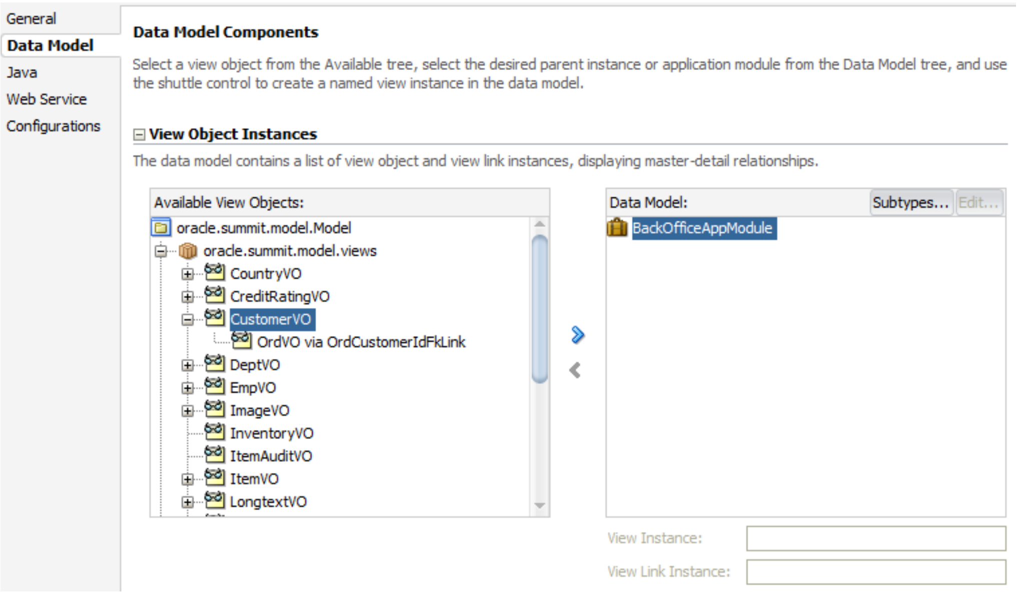
To add master-detail view object instances to a data model:

1. In the Applications window, double-click the application module for which you want to define a new master-detail view instances.
2. In the overview editor, click the **Data Model** navigation tab.
3. On the Data Model Components page, expand the **View Object Instances** section and, in the **Available View Objects** list, select the instance of the view object that you want to be the actively coordinating master.

   The master view object will appear with a plus sign in the list indicating the available view links for this view object. The view link must exist to define a master-detail hierarchy.

   For example, the `SummitADF` workspace of the Summit ADF sample application contains the master view object `CustomerVO` and detail view object `OrdVO` related by the view link `OrdCustomerIdFkLink`. To view the master-detail hierarchy in the **Available View Objects** list, you would expand `CountryVO`, as shown in Figure 13-4.

   Figure 13-4 Master View Object Selected



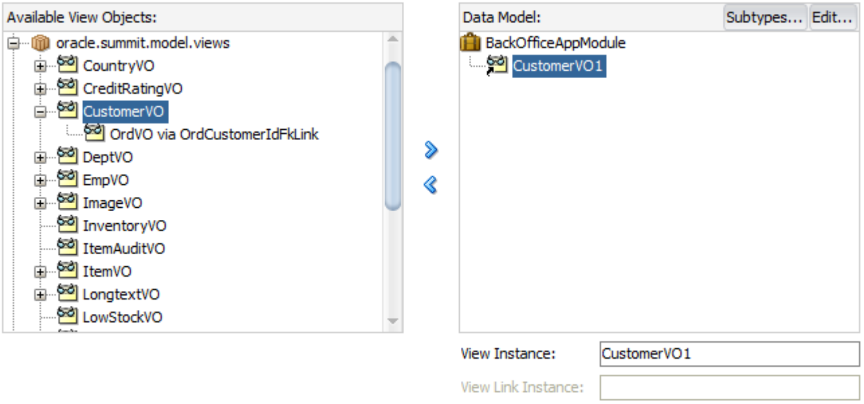
Select a view object from the Available tree, select the desired parent instance or application module from the Data Model tree, and use the shuttle control to create a named view instance in the data model.

The data model contains a list of view object and view link instances, displaying master-detail relationships.

4. Shuttle the selected master view object to the **Data Model** list.

   For example, you might shuttle the master view object `CustomerVO` to the data model, where it receives the view instance name `CustomerVO1`, as shown in Figure 13-5.

Figure 13-5 Master View Instance Created

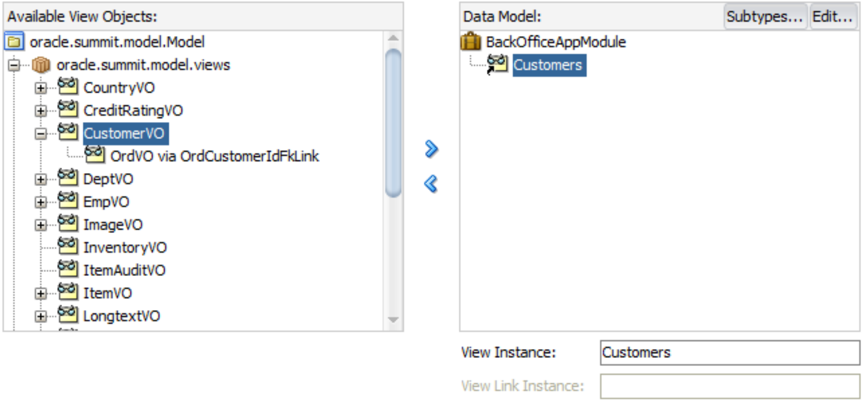


View Instance: CustomerVO1

5. To change the name of the newly created master view instance, enter a different name in the **View Instance** field.

   For example, you might change the view instance name `CustomerVO1` to `Customers`, as shown in Figure 13-6.

Figure 13-6 Overview Editor Displays View Instance Renamed

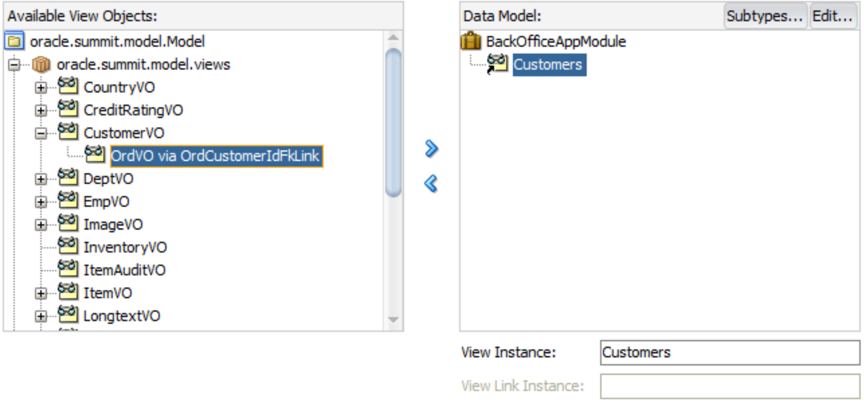


View Instance: Customers

6. In the **Data Model** list, leave the newly created master view instance selected so that it appears highlighted. This will be the target of the detail view instance you will add. Then locate and select the detail view object beneath the master view object in the **Available View Objects** list.

   For example, Figure 13-7 shows the detail `OrdVO` indented beneath master `CustomerVO` with the name `OrdVO via OrdCustomerIdFkLink`. The name identifies the view link `OrdCustomerIdFkLink`, which defines the master-detail hierarchy between `CustomerVO` and `OrdVO`. Notice also that `OrdVO` will have the view instance name `OrdersForCustomers` when added to the data model.

Figure 13-7   Detail View Object Selected



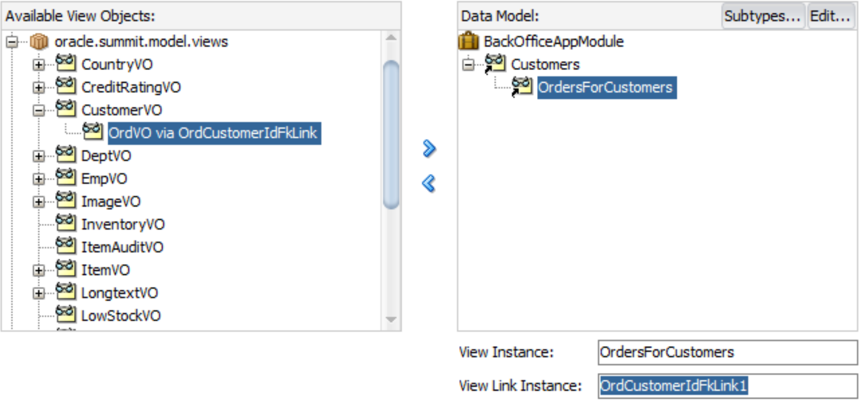
View Instance: Customers

View Link Instance:

7. To add the detail instance to the previously added master instance, shuttle the detail view object to the **Data Model** list below the selected master view instance and rename the view instance.

   For example, you might shuttle `OrdVO via OrdCustomerIdFkLink` to the data model beneath the selected master view instance `Customers`, as shown in Figure 13-8.

Figure 13-8   Detail View Instance Created and Renamed



View Instance: OrdersForCustomers

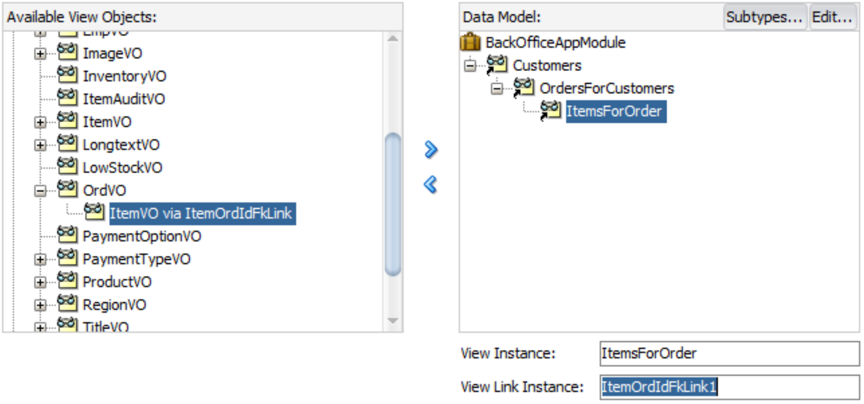
View Link Instance: OrdCustomerIdFkLink1

8. To add another level of hierarchy, repeat Step 3 through Step 6, but select the newly added detail in the **Data Model** list, then shuttle over the new detail, which itself has a master-detail relationship with the previously added detail instance.

   For example, the `SummitADF` workspace contains the view object `ItemVO via ItemOrdIdFkLink` which is a detail of the `OrdVO` view object, which is itself a detail of the master view object `CustomerVO`.

   To create the master-detail-detail data model hierarchy, you might shuttle the detail view object `ItemVO via ItemOrdIdFkLink` to the data model beneath the view instance `OrdersForCustomers` (renamed from `OrdVO1`) and rename the new detail view instance `ItemsForOrder`, as shown in Figure 13-9. In the **Data Model** list the view instance `Customers` (renamed from `CustomerVO1`) is the master of `OrdersForCustomer` (renamed from `OrdVO1`), which is, in turn, a master of `ItemsForOrder` (renamed from `ItemVO1`).

Figure 13-9   Master-Detail-Detail Hierarchy Created



View Instance: ItemsForOrder

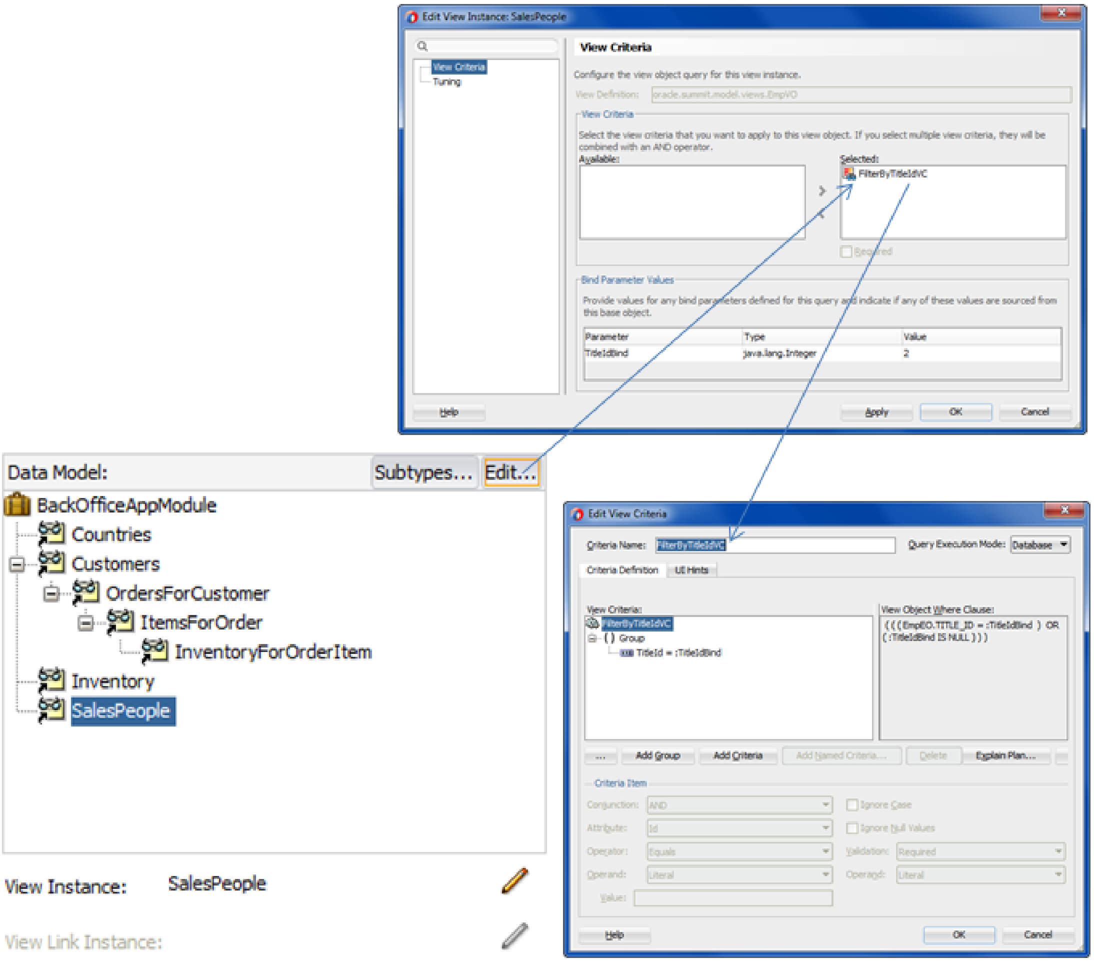
View Link Instance: ItemOrdIdFkLink1

Customizing a View Object Instance that You Add to an Application Module

You can optionally customize the view object instance by using the Data Model Components page of the overview editor for the application module. For example, you might want to apply a filter to set the controlling attribute for a master-detail view object relationship.

For example, in the `SummitADF` workspace of the Summit ADF sample application, the view instance `SalesPeople` has been defined for the `BackOfficeAppModule` application module and a view criteria `FilterByTitleIdVC` has been defined for this view instance that filters the sales staff by a title. Figure 13-10 shows the Edit View Instance dialog opened for the `SalesPeople` view instance with the `FilterByTitleIdVC` selected. The bind variable `TitleIdBind` defines a default value of 2, which sets the value of the `TitleId` attribute, as the controlling attribute for the `SalesPeople` view instance. The controlling attribute, when set by the view criteria filter, provides a way to retrieve only the view rows for the matching ID.

Figure 13-10 Customized View Object Instance Using a View Criteria FIlter



Edit View Instance: SalesPeople

View Criteria

Configure the view object query for this view instance.

View Definition: oracle.summit.model.views.EmpVO

Select the view criteria that you want to apply to this view object. If you select multiple view criteria, they will be combined with an AND operator.

Selected: FilterByTitleIdVC

□ Required

Provide values for any bind parameters defined for this query and indicate if any of these values are sourced from this base object.

Parameter	Type	Value
TitleIdBind	java.lang.Integer	2

Data Model:

View Instance: SalesPeople

View Link Instance:

Edit View Criteria

Criteria Name: FilterByTitleIdVC

Query Execution Mode: Database

View Criteria:

```
FilterByTitleIdVC
  ( ) Group
      TitleId = :TitleIdBind
```

View Object Where Clause:

```
( ( ( EmpEO.TITLE_ID = :TitleIdBind ) OR
( :TitleIdBind IS NULL ) ) )
```

Conjunction: AND

Attribute: Id

Operator: Equals

Operand: Literal

Value:

□ Ignore Case

□ Ignore Null Values

Validation: Required

Operand: Literal

Before you begin:

It may be helpful to have an understanding of application modules. For more information, see Creating and Modifying an Application Module.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Application Modules.

To customize a view object instance that you add to an existing application module:

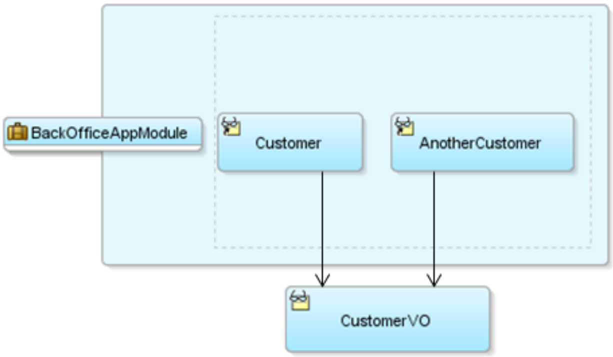
1. In the Applications window, double-click the application module for which you want to customize an existing view instance.
2. In the overview editor, click the **Data Model** navigation tab.
3. On the Data Model Components page, expand the **View Object Instances** section and, in the **Data Model** list, select the view object instance you want to customize and click the **Edit** button.
4. In the Edit View Instance dialog, perform any of the following steps, and then click **OK**.

- In the **View Criteria** group box, select one or more view criteria that you want to apply to the view object instance. The view criteria will be appended as a `WHERE` clause to the instance query. For details about defining view criteria, see Working with Named View Criteria.
- In the **View Criteria** group box, for each view criteria that you add to the **Selected** list, you may define one or more view criteria as mandatory filters on the view instance. Select the **Required** checkbox for the currently selected view criteria when it should remain applied on the view instance at all times. Normally, when multiple view criteria are applied to a view instance, one will be the default view criteria and should remain applied at all times. Mandatory view criteria, will remain applied on the view criteria list even when the application programmatically applies view criteria (using `applyViewCriteria()` for example).
- In the **Bind Parameters Values** group box, enter any values that you wish the instance to use when applying the defined view criteria. For more information about defining bind variables, see Working with Bind Variables.

What Happens When You Add a View Object Instance to an Application Module

You add instances of view object components to define the data model of an application module. Figure 13-11 shows a JDeveloper business components diagram of a `BackOfficeAppModule` application module that appears in the SummitADF workspace of the Summit sample application.

Figure 13-11 Application Module Containing Two Instances of a View Object Component



The sample application module contains two instances of the `CustomerVO` view object component, with member names of `Customer` and `AnotherCustomer` to distinguish them. At runtime, both instances share the same `CustomerVO` view object component definition—this ensures that they have the same attribute structure and view object behavior—however, each might be used independently to retrieve data about different users. For example, some of the runtime properties, like an additional filtering `WHERE` clause or the value of a bind variable, might be different on the two distinct instances.

The following example shows how the `BackOfficeAppModule` application module defines its member view object instances in its XML document file.

```
<AppModule
   Name="BackOfficeAppModule"
   ...>
   <ViewUsage
      Name="Customer"
      ViewObjectName="oracle.summit.model.appmodule.CustomerVO"/>
   <ViewUsage
      Name="AnotherCustomer"
      ViewObjectName="oracle.summit.model.appmodule.CustomerVO"/>
</AppModule>
```

How to Edit an Existing Application Module

After you've created a new application module, you can edit any of its settings by using the Edit Application Module dialog. To launch the editor, in the Applications window, right-click the application module node and choose **Open**, or double-click the application module. By visiting the different pages of the editor, you can adjust the data model to determine whether or not to reference nested application modules, specify Java generation settings, client interface methods, runtime instantiation behavior, and custom properties.

If you edit the name of your application module, choose a name that is not among the reserved words that Oracle Application Development Framework (Oracle ADF) defines. In particular, reserved words are not valid for a data control usage name which JDeveloper automatically assigns based on your application module's name. In Fusion web applications, these reserved words consist of `data`, `bindings`, `security`, and `adfContext`. For example, you should not name an application module `data`. If JDeveloper creates a data control usage with an ID that collides with a reserved word, your application may not reliably access your data control objects at runtime and may fail with a runtime `ClassCastException`.

Do not name the application module with an initial underscore (_) character to prevent a potential name collision with a wider list of reserved words that begin with the underscore.

Application module names that incorporate a reserved word into their name (or that change the case of the reserved word) will not conflict. For example, `Product_Data`, `Product_data`, or just `Data` are all valid application module names since the whole name does not match the reserved word `data`.

How to Change the Data Control Name Before You Begin Building Pages

By default, an application module will appear in the Data Controls panel as an **application module data control** named *`AppModuleName`*`DataControl`. The user interface designer uses the Data Controls panel to bind data from the application module to the application's web pages. For example, if the application module is named `SummitAppModule`, the Data Controls panel will display the data control with the name `SummitAppModuleDataControl`. You can change the default data control name to make it shorter or to supply a more preferable name.

When the user interface designer works with the data control, they will see the data control name for your application module in the `DataBindings.cpx` file in the user interface project and in each data binding page definition XML file. In addition, you might refer to the data control name in code when needing to work programmatically with the application module service interface. For this reason, if you plan to change the name of your application module, do this change before you begin building your view layer.

For complete information about the application module data control, see Using ADF Model in a Fusion Web Application.

> **Note:**
>
> If you decide to change the application module's data control name after you have already referenced it in one or more pages, you will need to open the **page definition files** and `DataBindings.cpx` file where it is referenced and update the old name to the new name manually.

To change the application module data control name:

1. In the Applications window, double-click the application module for which you want to edit the data control name.
2. In the Properties window, expand the **Other** section, and enter your preferred data control name in the **Data Control Name** field.

What You May Need to Know About Application Module Granularity

A common question related to application modules is, "How big should my application module be?" In other words, "Should I build one big application module to contain the entire data model for my enterprise application, or many smaller application modules that allow me to compartmentalize the application according to functionality?" The answer depends on your situation.

In general, application modules should be as big as necessary to support the specific use case you have in mind for them to accomplish. They can be assembled from finer-grained application module components using a nesting feature, as described in Defining Nested Application Modules. Since a complex business application is not really a single use case, a complex business application implemented using Oracle ADF will typically not be just a single application module.

In actual practice, you may choose any granularity you wish to support the type of modularization you require. For example, the need to support disparate data sources or the need to configure separate tuning options are both legitimate cases for creating multiple application modules. Or, in a small application with one main use case and a "backend" supporting use case, you might create two application modules. However, for the sake of simplicity you can combine both use cases, rather than create a second application module that contains just a couple of view objects.

Other situations may dictate creating more than one **root application module**. For example, if you need to support more than one transaction per user session, you can create multiple application modules. Alternatively, you can work with data control scope and task flows to manage transactions, as described in Managing Transactions in Task Flows.

What You May Need to Know About View Object Components and View Object Instances

While designing an application module, you use instances of a view object component to define its data model. Just as the user interface may contain two instances of a `Button` component with member names of `myButton` and `anotherButton` to distinguish them, your application module contains two instances of the `CustomerVO` view object component, with member names of `CustomerList` and `AnotherCustomerList` to distinguish them.

Configuring Your Application Module Database Connection

JDeveloper allows you to modify JDBC data source definition for each ADF application module as an alternative to using the default runtime JDBC data source connection for all application modules.

When you initialize your data model project to use ADF Business Components, JDeveloper prompts you to supply database connection details. You must specify a database connection before you can run ADF Business Components wizards and work with the database's tables and views to create business components. After you specify the connection details, JDeveloper also creates a Java Database Connectivity (JDBC) data source that will be used as the default runtime connection for all application modules created in the current project.

You can use the overview editor for application module configurations (on the `bc4j.xcfg` file) to change the JDBC data source definition for each application module individually. To display the application module configuration overview editor, double-click the application module in the Applications window and, in the overview editor, select the **Configurations** navigation tab. Then, in the Configurations page of the overview editor, click the configuration hyperlink. In cases where you do not want to use a data source, you also use the overview editor to replace the connection type for individual application modules with a JDBC URL connection type.

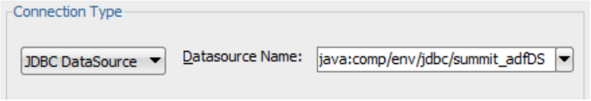
You can use either the JDBC data source or the JDBC URL connection type to run the application module in any context where Java can run. Your application is not restricted to running inside a Java Enterprise Edition (Java EE) application server. For example, although the Oracle ADF Model Tester is a standalone Java tool and does not run within the context of a Java EE application server, you can use either connection type to test your business components in the Oracle ADF Model Tester. You can use the application module configurations overview editor to select among both connection types for the existing application resource connections that appear in the Databases window.

JDeveloper configures the application module by default to use the data source connection type. A JDBC data source is a vendor-independent encapsulation of a database server connection. The JDBC data source offers advantages that the JDBC URL connection type does not. When you define a connection type based on a data source, you reconfigure the data source without changing the deployed application. The data source is also centrally defined at the application server level, whereas JDBC URL connections are not. In cases where a JDBC data source is not viable, at runtime, ADF Business Components will construct a default JDBC URL based on the data source connection information.

How to Use a JDBC Data Source Connection Type

The type of connection all default application module configurations use is a JDBC data source. You define a JDBC data source as part of your application server configuration information, and then the application module looks up the resource at runtime using a logical name. When you use the JDBC data source as the connection type, your application resource connection details may change, and you will not need to change the deployed application module configuration. For this reason the JDBC data source is the recommended choice for all application module configurations. Figure 13-12 shows how the default selection appears in the overview editor for application module configurations (on the `bc4j.xcfg` file). To display the application module configuration overview editor, double-click the application module in the Applications window and, in the overview editor, select the **Configurations** navigation tab. Then, in the Configurations page of the overview editor, click the configuration hyperlink.

Figure 13-12 JDBC DataSource Connection Type Setting



JDBC DataSource
Datasource Name: java:comp/env/jdbc/summit_adfDS

> **Note:**
>
> The other type of connection type displayed in the Connection Type dropdown is a JDBC URL connection. This connection type is supported for legacy applications and should not be used instead of the JDBC data source connection type.

The following example shows the `<resource-ref>` tags in the `web.xml` file of a Fusion web application. These define two logical data sources named `jdbc/Summit_adfDS` and `jdbc/Summit_adfCoreDS`. The application module configurations overview editor references this logical connection name after the prefix `java:comp/env` in the **Datasource Name** field. For example, the JDBC data source name for the same Fusion web application would display the value `java:comp/env/jdbc/summit_adfDS` that you can select. Therefore the **Datasource Name** field is prepopulated with the JNDI name for all available application resources connection names.

```
<!-- In web.xml -->
<resource-ref>
  <res-ref-name>jdbc/summit_adfDS</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
<resource-ref>
  <res-ref-name>jdbc/summit_adfCoreDS</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
```

You can directly edit the **Datasource Name** field when you want to specify a connection name for a global data source that is required to run the application on a target standalone application server. When you deploy to Oracle WebLogic Server, by default, the application-specific data source is not packaged with the application and Oracle WebLogic Server is configured to find a global data source named `jdbc/`*`applicationConnectName`*`DS` using the look up `java:comp/env/jdbc/`*`applicationConnectName`*`DS`. Therefore, by following this naming convention, you enable a single data source connection name to work correctly when running the application in JDeveloper using an application-specific data source or when running on the deployed standalone server using a global data source.

> **Note:**
>
> When configuring the ADF application module to access a highly available database system, such as redundant databases or Oracle Real Application Clusters (Oracle RAC) as the backend, the data source must be container-defined. In this scenario, the application module will use a multi data source; however, from the standpoint of the application module configuration, the naming convention for the multi data source is the same as it is for an non-multi data source. This ensures that the correct data source will be used at runtime. For details about configuring multi data sources for high availability applications, see Multi Data Sources in the *High Availability Guide*.

What Happens When You Create an Application Module Database Connection

When you select connection type in the Edit Configuration dialog, JDeveloper updates the application module configuration file, `bc4j.xcfg` in the `./common` subdirectory relative to the application module's XML document. The file defines configurations for all of the application modules in a single Java package. The `bc4j.xcfg` file does not appear in the Applications window. To view the `bc4j.xcfg` file in JDeveloper, double-click the application module in the Applications window and, in the overview editor, select the **Configurations** navigation tab. Then, in the Configurations page of the overview editor, click the configuration hyperlink.

For example, if you open the overview editor for application module configurations (on the `bc4j.xcfg` file) for the `SummitAppModule` application module in the `./classes/oracle/summit/model/services/common` directory of the core Summit ADF sample application's `Model` project, you will see the named configurations for this application module.

The configurations defined by the `bc4j.xcfg` file allow the Fusion web application to interact with specific, deployed application modules. In addition to the connection type for the application module, the `bc4j.xcfg` file contains metadata information about application module names and it contains the runtime parameters that are configured for the application module. The application resource database connection details for the named connection type are defined in the Connections node of the Applications window and saved in the application's `connections.xml` file.

The following example displays a sample `bc4j.xcfg` file from the core Summit ADF sample application. The configurations `SummitAppModuleLocal` and `SummitAppModule` both reference a data source (named `Summit_adfDS`) in the `JDBCDataSource` attribute.

The `JDBCDataSource` attribute in each configuration specifies the JNDI name for the application resources connection name in the form of `java:comp/env/jdbc/`*`applicationConnectName`*`DS`, where *`applicationConnectName`* is the name of the application resources database connection defined in JDeveloper (in this case, `Summit_adf`). This JNDI naming convention (with the application-specific name space `java:comp/env/jdbc/` and `DS` appended to the application resources database connection name) ensures that a deployed Fusion web application will run on Oracle WebLogic Server using the application's global data source and no changes will be required. The global data source is typically defined by the application server administrator using the Oracle WebLogic Server Administration Console.

```
<BC4JConfig version="11.1" xmlns="http://xmlns.oracle.com/bc4j/configuration">
   <AppModuleConfigBag 
ApplicationName="oracle.summit.model.services.SummitAppModule">
      <AppModuleConfig name="SummitAppModuleLocal"
                       DeployPlatform="LOCAL"
                       JDBCName="summit_adf"
                       jbo.project="oracle.summit.model.Model"

java.naming.factory.initial="oracle.jbo.common.JboInitialContextFactory"

ApplicationName="oracle.summit.model.services.SummitAppModule">
         <Database jbo.TypeMapEntries="Java"/>
         <Security 
AppModuleJndiName="oracle.summit.model.services.SummitAppModule"/>
         <Custom ns0:JDBCDataSource="java:comp/env/jdbc/summit_adfDS"
                                     xmlns:ns0="http://xmlns.oracle.com/bc4j/
configuration"/>
      </AppModuleConfig>
      ...
   </AppModuleConfigBag>
</BC4JConfig>
```

How to Change Your Application Module's Runtime Configuration

In addition to creating the application module XML definition, JDeveloper also adds a default configuration named *`appModuleName`*`Local` to the `bc4j.xcfg` file in the subdirectory named `common`, relative to the directory containing the application module XML definition file. The `bc4j.xcfg` file does not appear in the Applications window. To view the `bc4j.xcfg` file in JDeveloper, double-click the application module in the Applications window and, in the overview editor, select the **Configurations** navigation tab. Then, in the Configurations page of the overview editor, click the configuration hyperlink. To view the default settings or to change the application module's runtime configuration settings, select the **Database and Scalablity** tab in the overview editor and make the desired changes.

Before you begin:

It may be helpful to have an understanding of application module database connections. For more information, see Configuring Your Application Module Database Connection.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Application Modules.

To manage your application module's configuration:

1. In the Applications window, double-click the application module for which you want to edit the configuration.
2. In the overview editor, click the **Configurations** navigation tab.
3. In the Configurations page, click the configuration hyperlink that you want to edit.

   The Configuration page may display these three default configurations:

   - `*appModuleName*Local`: One local configuration is automatically defined when you create the application module. Create a new local configuration, for example, when you want to change the runtime properties for the application module for testing purposes.
   - `*appModuleName*Shared`: The shared application module configuration is present only when a named instance of a shared application module has been defined in the Project Properties dialog for the data model project. A shared application module is one that groups view instances when you want to reuse lists of static data across the application. For example, you can define a shared application module to group view instances that access lookup data, such as a list of countries.
   - `*appModuleName*Service` (type SI for service interface): The service interface configuration is present only when the service interface (web service implementation wrapper) has been defined in the Service Interface page of the overview editor for the application module. An application module exposed as a web service allows your application module to participate in a composite application to provide data access and method calls to web service clients. The same application module can support interactive web user interfaces using ADF data controls and web service clients.
4. In the overview editor for application module configurations (on the `bc4j.xcfg` file), click the **Database and Scalability** tab to customize the desired runtime properties and save the changes.

How to Change the Database Connection for Your Project

When you are developing applications, you may have a number of different users or schemas that you want to switch between. You can do this by changing the database connection properties of the project that contains the business components. The selection you make will automatically update the connection string for each configuration in your project's `bc4j.xcfg` file that specifies a JDBC URL type connection.

Before you begin:

It may be helpful to have an understanding of application module database connections. For more information, see Configuring Your Application Module Database Connection.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Application Modules.

To change the connection used by your application module's configuration:

1. In the Applications window, right-click the project that contains the application module and choose **Project Properties**.

2. In the Project Properties dialog, select **ADF Business Components** and from the **Connection** dropdown list, choose the desired connection and then click the **Edit** icon.
3. In the Edit Database Connection dialog, make the appropriate changes.
4. Click **OK**.

Defining Nested Application Modules

JDeveloper allows you to create composite ADF application modules by way of application module nesting. This allows a higher application module to use the inner application module’s function.

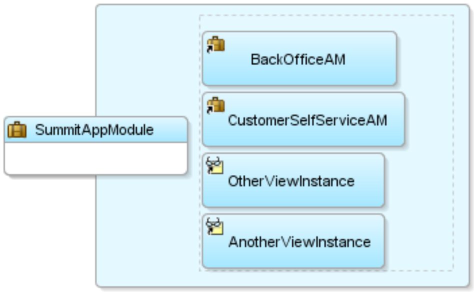
Application modules support the ability to create software components that mimic the modularity of your use cases, for which your higher-level functions might reuse a "subfunction" that is common to several business work flows. You can implement this modularity by defining composite application modules that you assemble using instances of other application modules. This task is referred to as **application module nesting**. That is, an application module can contain (logically) one or more other application module instances, as well as view object instances. The outermost containing application module is referred to as the **root application module**.

Declarative support for defining nested application modules is available through the overview editor for the application module, as shown in Figure 13-14. The API for application modules also supports nesting of application modules at runtime.

When you nest an instance of one application module inside another, you aggregate not only the view object instances in its data model, but also any custom service methods it defines. This feature of "nesting," or reusing, an instance of one application module inside of another is an important design aspect of ADF Business Components for implementing larger-scale, real-world application systems.

Considering that an application module represents an end-user use case or work flow, you can build application modules that cater to the data required by some shared, modular use case, and then reuse those application modules inside of other more complicated application modules that are designed to support a more complex use case. For example, imagine that after creating the application modules `BackOfficeAM` and `CustomerSelfServiceAM`, you later need to build an application that uses both of these services as an integral part of a new composite service application module. Figure 13-13 illustrates what this composite service would look like in a JDeveloper business components diagram. Notice that an application module like `SummitAppModule` can contain a combination of view object instances and application module instances.

Figure 13-13   Application Module Instances Can Be Reused to Assemble Composite Services



How to Define a Nested Application Module

To specify a composite root application module that nests an instance of an existing application module, use the overview editor for the application module. All of the nested component instances (contained by the application module instance) share the same transaction and entity object caches as the root application module that reuses an instance of them.

For example, in the `SummitADF` workspace of the Summit ADF sample application, the nested application module instances `BackOfficeAM` and `CustomerSelfServiceAM` share the same transaction and entity object caches as the root application module `SummitAppModule`.

> **Tip:**
>
> If you leverage nested application modules in your application, be sure to read How Nested Application Modules Appear in the Data Controls Panel to avoid common pitfalls when performing data binding involving them.

Before you begin:

It may be helpful to have an understanding of nested application modules. For more information, see Defining Nested Application Modules.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Application Modules.

You will need to complete this task:

Create the desired application modules, as described in How to Create an Application Module.

To define a nested application module:

1. In the Applications window, double-click the root application module for which you want to define a nested application module.
2. In overview editor, click the **Data Model** navigation tab.

3. In the Data Model Components page, expand the **Application Module Instances** section and, in the **Available** list, select the application module that you want to add to the data model.

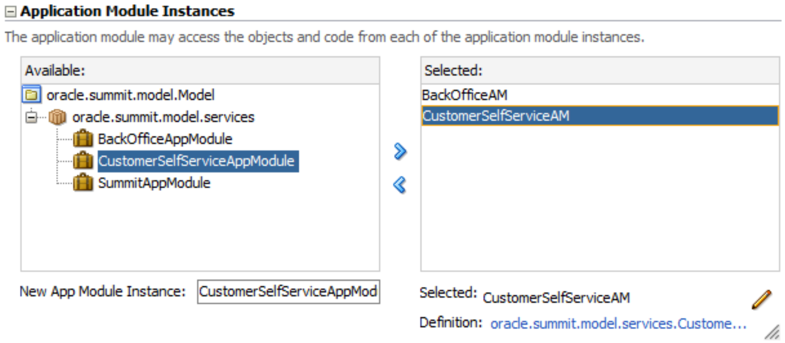
   The **New App Module Instance** field below the list shows the name that will be used to identify the nested application module that you add to the data model.

4. To change the name before adding it, type a different name in the **New App Module Instance** field.

5. With the desired application module selected, shuttle the application module to the **Selected** list.

   For example, in the `SummitADF` workspace of the Summit ADF sample application, the `SummitAppModule` application module defines two nested application module instances: `BackOfficeAM` and `CustomerSelfServiceAM`. Figure 13-14 shows the application module instances `BackOfficeAM` and `CustomerSelfServiceAM` have been added to the **Selected** list.

Figure 13-14 Data Model Displays Added Application Module Instances



⊟ Application Module Instances

The application module may access the objects and code from each of the application module instances.

Available:
- oracle.summit.model.Model
- oracle.summit.model.services
  - BackOfficeAppModule
  - CustomerSelfServiceAppModule
  - SummitAppModule

Selected:
- BackOfficeAM
- CustomerSelfServiceAM

New App Module Instance: CustomerSelfServiceAppMod

Selected: CustomerSelfServiceAM

Definition: oracle.summit.model.services.Custome...

What You May Need to Know About Root Application Modules Versus Nested Application Module Usages

At runtime, your application works with a *main* — or what's known as a *root* — application module. Any application module can be used as a root application module; however, in practice the application modules that are used as root application modules are the ones that map to more complex end-user use cases, assuming you're not just building a straightforward CRUD application. When a root application module contains other nested application modules, they all participate in the root application module's transaction and share the same database connection and a single set of entity caches. This sharing is handled for you automatically by the root application module and its `Transaction` object.

Additionally, when you construct an application using an **ADF bounded task flow**, to declaratively manage the transactional boundaries, Oracle ADF will automatically nest application modules used by the task flow at runtime. For details about bounded task flows and transactions, see Managing Transactions in Task Flows.

Creating an Application Module Diagram for Your Business Service

You can create an ADF Application Module UML diagram that can be referenced by other users. Use the diagram to modify Application Module details.

As you develop the business service's data model, it is often convenient to be able to visualize it using a UML model. JDeveloper supports easily creating a diagram for your application module that other developers can use for reference.

You can perform a number of tasks directly on the diagram, such as editing the application module, controlling display options, filtering methods names, showing related objects and files, publishing the application, and launching the Oracle ADF Model Tester.

How to Create an Application Module Diagram

To create an application module diagram, use the Create Business Components Diagram dialog, which is available in the New Gallery.

Before you begin:

It may be helpful to have an understanding of the UML model diagram For more information, see Creating an Application Module Diagram for Your Business Service.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Application Modules.

You will need to complete this task:

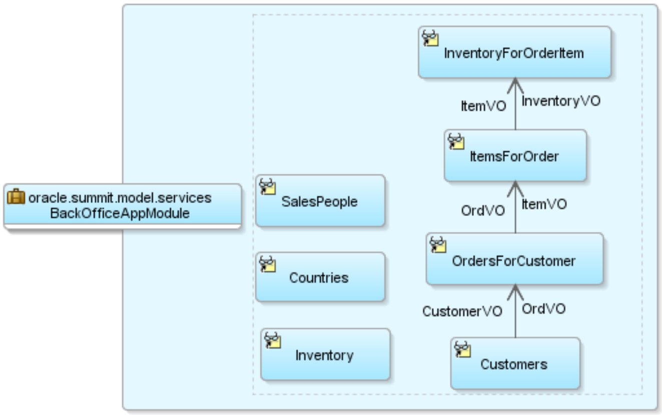
Create the desired application modules, as described in How to Create an Application Module.

To create a diagram of your application module:

1. In the Applications window, right-click the data model project in which you want to create the diagram and choose **New** and then **Business Components Diagram**.
2. In the Create Business Components dialog, enter a diagram name and a package name in which the diagram will be created.
3. Click **OK** to create the empty diagram and open the diagrammer.
4. In the Applications window, select the desired application module and drop it onto the diagram surface.
5. Inside the application module diagram, right-click the desired diagram node and choose **Visual Properties** and make the desired changes in the Edit Visual Properties dialog.
6. In the diagram editor, click anywhere outside of the application module diagram to display the Properties window for the diagram editor and make these changes:
   - Set the layout style
   - Change the font
   - Turn off the grid

After you complete these steps, the diagram looks similar to the diagram shown in Figure 13-15.

Figure 13-15 Partial UML Diagram of Application Module



What Happens When You Create an Application Module Diagram

When you create a business components diagram, JDeveloper creates a `.adfbc_diagram` file to represents the diagram in a subdirectory of the project's model path that matches the package name in which the diagram resides.

By default, the Applications window unifies the display of the project content's paths so that ADF components and Java files in the source path appear in the same package tree as the UML model artifacts in the project model path. You can use the **Applications Window Options > Directory View** toolbar option in the Applications window to switch between the unified directory view and a more distinct directory path view of the project content.

How to Use the Diagram to Edit the Application Module

The UML diagram of business components is not just a static picture that reflects the point in time when you dropped the application module onto the diagram. Rather, it is a UML-based rendering of the current component definitions, so it will always reflect the current state of affairs. The UML diagram is both a visualization aid and a visual navigation and editing tool.

You can bring up the overview editor for any application module in a diagram by choosing **Properties** from the context menu (or by double-clicking the application module).

You can also perform some application module editing tasks directly on the diagram, tasks such as renaming view object instances, dropping view object definitions from the Applications window onto the data model to create a new view object instance, and removing view object instances by pressing the Delete key.

Note:

Deleting components from the diagram only removes their visual representation on the diagram surface. The components and classes remain on the file system and in the Applications window.

How to Control Diagram Display Options

After you display the application module in the diagram, you can use the Edit Visual Properties dialog to control its display options. To display the dialog for the diagram, in the diagram, right-click the application module node and choose **Visual Properties**.

In the **Display** page, toggle properties like the following:

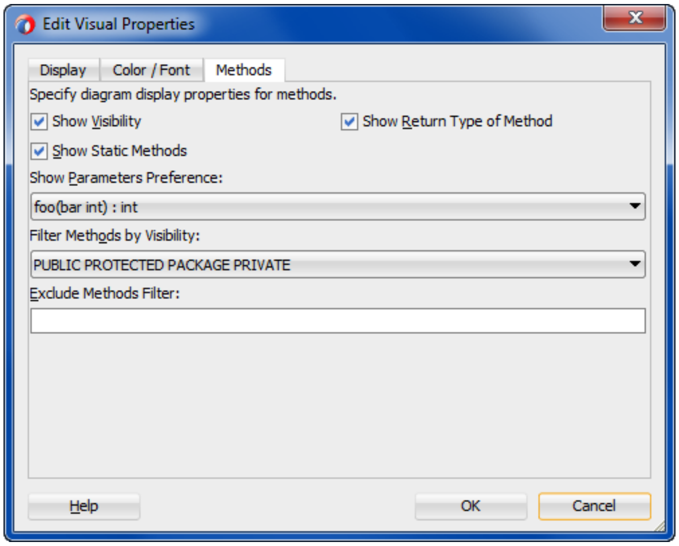
- **Show Package** — to display the package name
- **Show Methods** — to display service methods
- **Show Stereotype** — to display the type of object (for example "`<<application module>>`")
- **Show Usages** — to display all usages by other application modules in the current project

In the **Methods** page, consider changing the following settings depending on the amount of detail you want to provide in the diagram:

- **Show Visibility** (public, private, etc.)
- **Show Return Type of Method**
- **Show Static Methods**

By default, all operations of the application module are fully displayed, as shown by the Edit Visual Properties dialog settings in Figure 13-16.

Figure 13-16 Edit Visual Properties Dialog with Default Diagram Editor Options



Specify diagram display properties for methods.

☑ Show Visibility ☑ Show Return Type of Method

☑ Show Static Methods

Show Parameters Preference: foo(bar int) : int

Filter Methods by Visibility: PUBLIC PROTECTED PACKAGE PRIVATE

Exclude Methods Filter:

On the context menu of the diagram, you can also select to **View As**:

- **Compact** — to show only the icon and the name
- **Symbolic** — to show service operations
- **Expanded** — to show operations and data model (*default*)

How to Filter Method Names Displayed in the Diagram

Initially, if you show the methods for the application module, the diagram displays all the methods. Any method it recognizes as an overridden framework method displays in the **<<Framework>>** operations category. The rest display in the **<<Business>>** methods category.

The **Exclude Method Filters** setting in the **Methods** page of the Edit Visual Properties dialog is a regular expression that you can use to filter out methods you don't want to display on the diagram. For example, by setting **Exclude Method Filters** to:

```
findLoggedInUser.*|retrieveOrder.*|get.*
```

you can filter out all of the following application module methods:

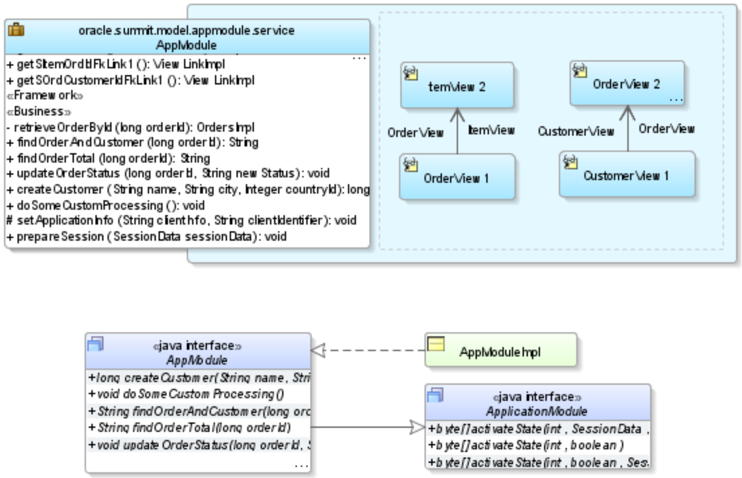
- `findLoggedInUserByEmail`
- `retrieveOrderById`
- All the generated view object getter methods

How to Show Related Objects and Implementation Files in the Diagram

After selecting the application module on the diagram — or any set of individual view object instances in its data model — you can choose **Show > Related Elements** from the context menu to display related component definitions on the diagram. In a similar fashion, choosing **Show > Implementation Files** will include the files that implement the application module on the diagram. You can repeat these options on the additional diagram elements that appear until the diagram includes the level of detail you want to convey.

Figure 13-17 illustrates how the diagram displays the implementation files for an application module. You will see the related elements for the application module's implementation class (`AppModuleImpl`). The diagram also draws an additional dependency line between the application module and the implementation class. If you have cast the application module instance to a specific custom interface, the diagram will also show that.

Figure 13-17 Adding Detail to a Diagram Using Show Related Elements and Show Implementation Files


oracle.summit.model.appmodule.service
AppModule
+ getSItemOrdIdFkLink1 (): ViewLinkImpl
+ getSOrdCustomerIdFkLink1 (): ViewLinkImpl
«Framework»
«Business»
- retrieveOrderById (long orderId): OrdersImpl
+ findOrderAndCustomer (long orderId): String
+ findOrderTotal (long orderId): String
+ updateOrderStatus (long orderId, String newStatus): void
+ createCustomer (String name, String city, Integer countryId): long
+ doSomeCustomProcessing (): void
# setApplicationInfo (String clientInfo, String clientIdentifier): void
+ prepareSession (SessionData sessionData): void
ItemView 2
OrderView 2
OrderView
ItemView
CustomerView
OrderView
OrderView 1
CustomerView 1
«java interface»
AppModule
+long createCustomer(String name, Stri
+void doSomeCustomProcessing()
+String findOrderAndCustomer(long orc
+String findOrderTotal(long orderId)
+void updateOrderStatus(long orderId, S
AppModuleImpl
«java interface»
ApplicationModule
+byte[] activateState(int, SessionData,
+byte[] activateState(int, boolean)
+byte[] activateState(int, boolean, Ses

How to Publish the Application Module Diagram

To publish the diagram to `PNG`, `JPG`, `SVG`, or compressed `SVG` format, choose **Publish Diagram** from the context menu on the diagram surface.

How to Test the Application Module from the Diagram

To launch the Oracle ADF Model Tester for an application module in the diagram, choose **Run** from the context menu.

Supporting Multipage Units of Work

Use ADF application module pooling to implement scalable, fault-tolerant applications.

While interacting with your Fusion web application, end users might:

- Visit the same pages multiple times, expecting fast response times
- Perform a logical unit of work that requires visiting many different pages to complete
- Need to perform a partial "rollback" of a pending set of changes they've made but haven't saved yet.
- Unwittingly be the victim of an application server failure in a server farm before saving pending changes

The **application module pooling** and state management features simplify implementing scalable, well-performing applications to address these requirements.

> **Note:**
>
> ADF **bounded task flows** can represent a transactional unit of work. You can specify options on the task flow to determine how to handle the transaction. For details about the declarative capabilities of ADF bounded task flows, see Managing Transactions in Task Flows.

How to Simulate State Management in the Oracle ADF Model Tester

To simulate what the state management functionality does, you can launch two instances of Oracle ADF Model Tester on an application module.

Before you begin:

It may be helpful to have an understanding of state management. For more information, see Supporting Multipage Units of Work.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Application Modules.

To simulate transaction state passivation using the Oracle ADF Model Tester:

1. Run the Oracle ADF Model Tester and double-click a view object instance to query its data.
2. Make a note of the current values of several attributes for a few rows.
3. Update those rows to have a different value for those attributes, but do not commit the changes.
4. Choose **File > Save Transaction State** from the Oracle ADF Model Tester main menu.

   A Passivated Transaction State dialog appears, indicating a numerical transaction ID number. Make a note of this number.
5. Exit the Oracle ADF Model Tester completely.
6. Restart the Oracle ADF Model Tester and double-click the same view object instance to query its data.
7. Notice that the data is *not* changed. The queried data from the data reflects the current state of the database without your changes.
8. Choose **File > Restore Transaction State** from the Oracle ADF Model Tester main menu, and enter the transaction ID you noted in Step 4.

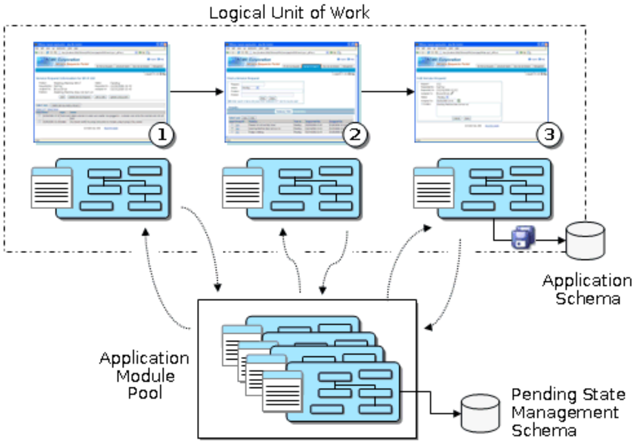
At this point, you'll see that your pending changes are reflected again in the rows you modified. If you commit the transaction now, your changes are permanently saved to the database.

What Happens at Runtime: How the Application Uses Application Module Pooling and State Management

Applications you build that leverage an application module as their business service take advantage of an automatic **application module pooling** feature. This facility manages a configurable set of application module instances that grows and shrinks as the end-user load on your application changes during the day. Due to the natural "think time" inherent in the end user's interaction with your application user interface, the number of application module instances in the pool can be smaller than the overall number of active users using the system.

As shown in Figure 13-18, as a given end user visits multiple pages in your application to accomplish a logical task, with each page request an application module instance in the pool is acquired automatically from the pool for the lifetime of that one request. At the end of the request, the instance is automatically returned to the pool for use by another user session. In order to protect the end user's work against application server failure, the application module supports the ability to freeze the set of pending changes in its entity caches to a persistent store by saving an XML snapshot describing the change set. For scalability reasons, this state snapshot is typically saved in a state management schema that is a different database schema than the one containing the application data.

Figure 13-18 Using Pooled Application Modules Throughout a Multipage, Logical Unit of Work



The pooling algorithm affords a tunable optimization whereby a certain number of application module instances will attempt to stay "sticky" to the last user session that returned them to the pool. The optimization is not a guarantee, but when a user can benefit from the optimization, they continue to work with the same application module instance from the pool as long as system load allows. When load is too high, the pooling algorithm uses any available instance in the pool to service the user's request and the frozen snapshot of their logical unit of work is reconstituted from the persistent store to allow the new instance of the application module to continue where the last one left off. The end user continues to work in this way until they commit or roll back their changes.

Using these facilities, the application module delivers the productivity of a stateful paradigm that can easily handle multipage work flows, in an architecture that delivers the runtime performance near that of a completely stateless application. You will learn more about these application module features in Using State Management in a Fusion Web Application and about how to tune them in Tuning Application Module Pools.

Customizing an Application Module with Service Methods

Any external client code which is a logical aspect of business functionality implementation can be accommodated in an ADF Application Module's custom Java class.

An application module can expose its data model of view object instances to clients without requiring any custom Java code. This allows client code to use the `ApplicationModule`, `ViewObject`, `RowSet`, and `Row` interfaces in the `oracle.jbo` package to work directly with any view object in the data model. However, just because you *can* programmatically manipulate view objects any way you want to in client code doesn't mean that doing so is always a best practice.

Whenever the programmatic code that manipulates view objects is a logical aspect of implementing your complete business service functionality, you should encapsulate the details by writing a custom method in your application module's Java class. This includes code that:

- Configures view object properties to query the correct data to display
- Iterates over view object rows to return an aggregate calculation
- Performs any kind of multistep procedural logic with one or more view objects

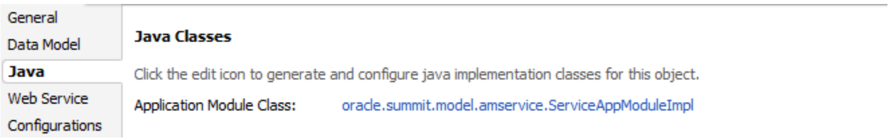
By centralizing these implementation details in your application module, you gain the following benefits:

- You make the intent of your code more clear to clients.
- You allow multiple client pages to easily call the same code if needed.
- You simplify regression-testing of your complete business service functionality.
- You keep the option open to improve your implementation without affecting clients.
- You enable *declarative* invocation of logical business functionality in your pages.

How to Generate a Custom Class for an Application Module

To add a custom service method to your application module, you must first enable a custom Java class for it. If you have configured your IDE-level Business Components Java generation preferences to automatically generate an application module class, a custom class will be present. As Figure 13-19 shows, if you're not sure whether your application module has a custom Java class, open the overview editor for the application module node in the Applications window. The Java Classes page of the editor displays the complete list of classes generated for the application module in the project. If the file exists because someone created it already, then the Java Classes page will display a linked file name identified as the **Application Module Class**. To open an existing file in the source editor, click the corresponding file name link.

Figure 13-19   Application Module's Custom Java Class in Overview Editor



Java Classes

Click the edit icon to generate and configure java implementation classes for this object.

Application Module Class: oracle.summit.model.amservice.ServiceAppModuleImpl

If no Java class exists in your project, you can generate one using the Java Classes page of the overview editor for the application module.

Before you begin:

It may be helpful to have an understanding of application module service methods. For more information, see Customizing an Application Module with Service Methods.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Application Modules.

You will need to complete this task:

Create the desired application modules, as described in How to Create an Application Module.

To generate a Java file for your application module class:

1. In the Applications window, double-click the application module.
2. In the overview editor, click the **Java** navigation tab and then click the **Edit java options** button.
3. In the Select Java Options dialog, select **Generate Application Module Class**.
4. Click **OK**.

   The new `.java` file will appear in the Java Classes page.

What Happens When You Generate a Custom Class for an Application Module

When you generate a custom class for an application module, JDeveloper creates the file in the same directory as the component's XML document file. The default name for its custom Java file will be *AppModuleName*`Impl.java`.

The Java generation option choices you made for the application module persist on the Java Classes page on subsequent visits to the overview editor for the application module. Just as with the XML definition file, JDeveloper keeps the generated code in your custom Java classes up to date with any changes you make in the editor. If later you decide you do not require a custom Java file, from the Java Classes page open the Select Java Options dialog and deselect **Generate Application Module Class** to remove the custom Java file from the project.

What You May Need to Know About Default Code Generation

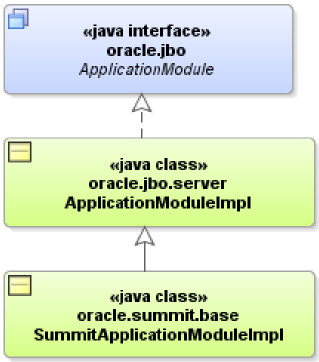
By default, the application module Java class will look similar to what you see in the following example when you've first enabled it. Of interest, it contains Getter methods for each view object instance in the data model.

```
package oracle.summit.model.services;

import oracle.jbo.server.ApplicationModuleImpl;
// ---------------------------------------------------------------------
// ---    File generated by ADF Business Components Design Time.
// ---    Custom code may be added to this class.
// ---    Warning: Do not modify method signatures of generated methods.
// ---------------------------------------------------------------------
```

```
public class AppModuleImpl extends ApplicationModuleImpl {
  /** This is the default constructor (do not remove) */
  public AppModuleImpl() { }

  /** Container's getter for YourViewObjectInstance1 */
  public YourViewObjectImpl getYourViewObjectInstance1() {
    return (YourViewObjectImpl)findViewObject("YourViewObjectInstance1");
  }

  // ... Additional ViewObjectImpl getters for each view object instance

  // ... ViewLink getters for view link instances here
}
```

As shown in Figure 13-20, your application module class extends the base ADF `ApplicationModuleImpl` class to inherit all the default behavior before adding your custom code.

Figure 13-20 Your Custom Application Module Class Extends ApplicationModuleImpl



How to Add a Custom Service Method to an Application Module

To add a custom service method to an application module, simply navigate to the application module's custom class and enter the Java code for a new method into the application module's Java implementation class. Use the following guidelines to decide on the appropriate visibility for the method:

- If you will use the method only inside this component's implementation as a helper method, make the method `private`.
- If you want to allow eventual subclasses of your application module to be able to invoke or override the method, make it `protected`.
- If you need clients to be able to invoke it, it must be `public`.

> **Note:**
>
> The application module examples in this chapter use strongly typed, custom entity object classes. For details about creating these entity classes, see Creating a Business Domain Layer Using Entity Objects.

The following example shows a `private retrieveOrderById()` helper method in the `AppModuleImpl.java` class for the `AppModule` application module. It uses the `static getDefinitionObject()` method of the `OrdEOImpl` entity object class to access its related entity definition, it uses the `createPrimaryKey()` method on the entity object class to create an appropriate Key object to look up the order, and then it uses the `findByPrimaryKey()` method on the entity definition to find the entity row in the entity cache. It returns an instance of the strongly typed `OrdersImpl` class, the custom Java class for the `Orders` entity object.

```
// In summit.model.appmodule.service.AppModuleImpl class
/*
 * Helper method to return a Order by Id
 */
private OrdersImpl retrieveOrderById(long orderId) {
  EntityDefImpl orderDef = OrdersImpl.getDefinitionObject();
  Key orderKey = OrdersImpl.createPrimaryKey(new DBSequence(orderId));
  return (OrdersImpl)orderDef.findByPrimaryKey(getDBTransaction(), orderKey);
}
```

The following example shows a `public findOrderAndCustomer()` method that allows the caller to pass in an ID of an order to be created. It uses the `getCustomer()` method of the `OrdersImpl` entity object class to access its related entity definition.

```
/*
 * Create a new Customer and Return its new id
 */
public String findOrderAndCustomer(long orderId) {
    OrdersImpl order = retrieveOrderById(orderId);
  if (order != null) {
        CustomerImpl cust = order.getCustomer();
    if (cust != null) {
      return "Customer: " + cust.getName() + ", Location: " + cust.getCity();
    }
    else {
      return "Unassigned";
    }
  }
  else {
    return null;
  }
}
```

How to Test the Custom Application Module Using a Static Main Method

When you are ready to test the methods of your custom application module, you can use JDeveloper to generate JUnit test cases. With JUnit, you can use any of the programmatic APIs available in the `oracle.jbo package` to work with the application module and invoke the custom methods. For details about using JUnit with ADF Business Components, see Regression Testing with JUnit.

As an alternative to JUnit test cases, a common technique to test your custom application module methods is to write a simple test case. For example, you could build the testing code into an object and include that code in a `static main()` method. The following example shows a sample `main()` method you could add to your custom application module class to test the sample methods you will write. You'll make use of a Configuration object (see How to Create a Command-Line Java Test Client) to instantiate and work with the application module for testing.

> **Note:**
>
> The fact that this Configuration object resides in the oracle.jbo.client package suggests that it is used for accessing an application module as an application *client*. Because a main() method is a kind of programmatic, command-line client, so this is an acceptable practice. Furthermore, even though you typically would not cast the return value of createRootApplicationModule() directly to an application module's implementation class, it is legal to do so in this one situation since despite being a client to the application module, the main() method's code resides right inside the application module implementation class itself.

A glance through the code in the following example shows that it exercises the methods created in the previous examples to:

1. Retrieve the total for order 101.
2. Retrieve the name of the customer for order 101.
3. Set the status of order 101 to the value "Y".

```
package oracle.summit.model.appmodule.service;

import oracle.jbo.ApplicationModule;
import oracle.jbo.JboException;
import oracle.jbo.client.Configuration;

public class TestAM {
    public TestAM() {
        super();
    }

    /*
     * Testing method
     */

    public static void main(String[] args) {
      String amDef = "oracle.summit.model.appmodule.service.AppModule";
      String config = "AppModuleLocal";
      ApplicationModule am =
                      Configuration.createRootApplicationModule(amDef,config);
        /*
         * NOTE: This cast to use the AppModuleImpl class is OK since this
         *       code is inside a business tier file and not in a
         *       client class that is accessing the business tier from "outside".
         */
        AppModuleImpl service = (AppModuleImpl)am;

        String customerName = service.findOrderAndCustomer(101);
        System.out.println("Customer for Order # 101 = " + customerName);
        try {
           service.updateOrderStatus(101,"Y");
        }
        catch (JboException ex) {
                System.out.println("ERROR: "+ex.getMessage());
            }
          Configuration.releaseRootApplicationModule(am,true);
        }
}
```

Running the custom application module class calls the `main()` method in the above example and produces the following output:

```
anonymous
Customer for Order # 101 = Customer: Kam's Sporting Goods, Location: Hong Kong
```

> **Note:**
>
> For an explanation of how you can use the client application to invoke the custom service methods that you create in your custom application module, see Publishing Custom Service Methods to UI Clients.

What You May Need to Know About Programmatic Row Set Iteration

Any time your application logic accesses a row set to perform programmatic iteration, you should use a secondary **row set iterator** when working with view object instances in an application module's data model, or view link accessor row sets of these view object instances, since they may be bound to user interface components. To create a secondary iterator, use the `createRowSetIterator()` method on the row set you are working with. When you are done using it, call the `closeRowSetIterator()` method on the row set to remove the secondary iterator from memory. The following example shows a typical application module custom method that correctly uses a secondary row set iterator for programmatic iteration, because the `CustomerView1` view object instance in its data model may be bound to a user interface (either now or at a later time).

```
// Custom method in an application module implementation class
public void doSomeCustomProcessing() {
    ViewObject vo = getCustomerView1();
    // create secondary row set iterator with system-assigned name
    RowSetIterator iter = vo.createRowSetIterator(null);
    while (iter.hasNext()) {
        Row r = iter.next();
        // Do something with the current row.
        Integer custId = (Integer)r.getAttribute("Id");
        String name  = (String)r.getAttribute("Name");
        System.out.println(custId + " " + name);
    }
    // close secondary row set iterator
    iter.closeRowSetIterator();
}
```

Note:

The same recommendation holds for custom code in a view object's implementation class that iterates its own default row set using that row set's default row set iterator.

There are two important reasons to follow this recommendation. Failing to do so can lead to confusion for the end user when the current row unexpectedly changes or it can introduce subtle business logic errors because the first or last row, or both rows get skipped.

- Confusing the end user by changing the current row unexpectedly

  The **iterator bindings** determine what row the end-user sees as the current row in the row set. If your own programmatic logic iterates through the row set using the same default row set iterator that the iterator binding uses, you may inadvertently change the current row the user has selected, leaving the user confused.

- Introducing subtle business logic errors by inadvertently skipping the first or last row

  Iterator bindings force their row set iterator to be on a valid row to guarantee that UI components display data when the row set is not empty. This has the side-effect of preventing your custom logic from navigating to the slot either before the first row or to the slot after the last row (when it is using the same row set iterator as an iterator binding). In concrete terms, this means that a typical `while (iter.hasNext())` row set iteration loop will either be skipped or start by processing the second row instead of the first as shown in the following example.

```
// Reset the default row set iterator (iter) to the slot before the first row
iter.reset();
// If an iterator binding is bound to the same default row set iterator,
// then it has already forced it to navigate to the first row here instead
// of being on the slot before the first row.
//
// If the row set iterator has only one row, the following will then return false
while (iter.hasNext()) {
  // If the row set has more than one row, the first time through the loop
  // this call to next() will return the second row rather than the first
  // row as expected.
  Row curRow = iter.next();
  // Do something with current row
}
```

Customizing Application Module Message Strings

You can add a resource bundle file for an ADF Application Module. You use this resource bundle file or .properties file to hold custom message strings.

The ADF application module does not normally require a resource bundle of its own. However, you can create a custom message string file (`.properties`) that you can use to add your custom message strings.

The `.properties` file you create can reference attribute properties by their fully qualified package name and custom method exception messages. For example, you might define message keys and strings as follows:

```
test.Order.Orderno_LABEL=Order Number
INVALID=You have called the method foo in an invalid way.
```

If your resource bundle defines a message for a method exception message, the custom method should appear in the application module client interface, as described in How to Publish a Custom Method on the Application Module's Client Interface.

For example, if you defined a message for the method `foo()` to replace the exception message `INVALID`, your interface might define this method to invoke the message from the resource bundle as:

```
public void foo() {
    ResourceBundleDef r = getResourceBundleDef();
    throw new JboException(r,"INVALID",null);
}
```

How to Add a Resource Bundle File for the Application Module

When you want to create a `.properties` file to contain custom message strings, you use the resource bundle option on the Resource Bundle page of the Project Properties dialog and select the option **One Bundle Per File**. By default JDeveloper sets the option to **One Bundle Per Project**, which produces a single `.properties` file for the project.

Before you begin:

It may be helpful to have an understanding of how projects use resource bundles. For more information, see Customizing Application Module Message Strings.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Application Modules.

You will need to complete these tasks:

1. Create the desired application module, as described in How to Create an Application Module.
2. Set the project properties for the data model project to generate one bundle per file, as described in Working with Resource Bundles.

To generate the resource bundle for the application module:

1. In the Applications window, double-click the application module for which you want to generate the resource bundle.
2. In the overview editor, click the **General** navigation tab and enter any display name.

   Adding a display name for the application module will generate the `.properties` file with a message key for the display name.
3. Save the file.

What Happens When You Add a Resource Bundle to an Application Module

When you generate a resource bundle for an application module, JDeveloper creates the `.properties` file in the same directory as the component's XML document file. The default name for its custom Java file will be *`AppModuleName`*`MsgBundle.properties`.

The following example shows the `ResourceBundle` element in the application module XML document that references the `.properties` file.

```
<AppModule
  ...
  <ResourceBundle>
    <PropertiesBundle
      PropertiesFile=
         "oracle.summit.model.services.common.BackOfficeAppModuleMsgBundle"/>
  </ResourceBundle>
</AppModule>
```

Publishing Custom Service Methods to UI Clients

For the user interface to invoke a public custom method in the ADF Application Module class, include this method on the Application Module’s UI client interface.

When you add a `public` custom method to your application module class, if you want your application's UI to be able to invoke it, you need to include the method on the application module's UI client interface.

How to Publish a Custom Method on the Application Module's Client Interface

To include a public method from your application module's custom Java class on the client interface, use the Java Classes page of the overview editor for the application module.

Before you begin:

It may be helpful to have an understanding of the purpose of the client interface. For more information, see Publishing Custom Service Methods to UI Clients.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Application Modules.

You will need to complete this task:

> Create the desired application modules, as described in How to Create an Application Module.

To publish a custom method on the client interface:

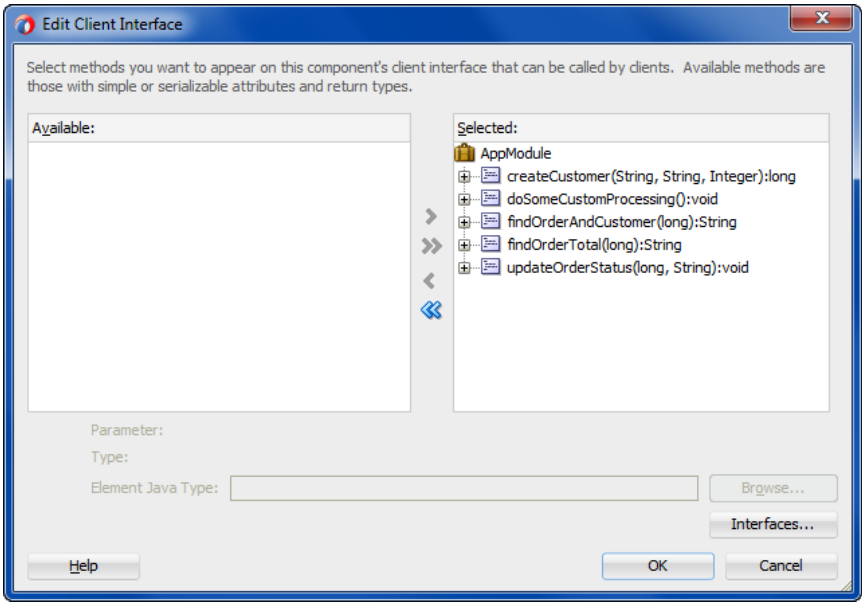
1. In the Applications window, double-click the application module for which you want to define the client interface.

2. In the overview editor, click the **Java** navigation tab and in the **Client Interface** section, click the **Edit application module client interface** button.
3. In the Edit Client Interface dialog, select one or more desired methods from the **Available** list and click the **Add** button to shuttle them into the **Selected** list.
4. Click **OK**.

   The new methods will appear in the **Client Interface** section of the Java Classes page.

Figure 13-21 shows multiple public methods added to the client interface.

Figure 13-21 Public Methods Added to an Application Module's Client Interface



Select methods you want to appear on this component's client interface that can be called by clients. Available methods are those with simple or serializable attributes and return types.

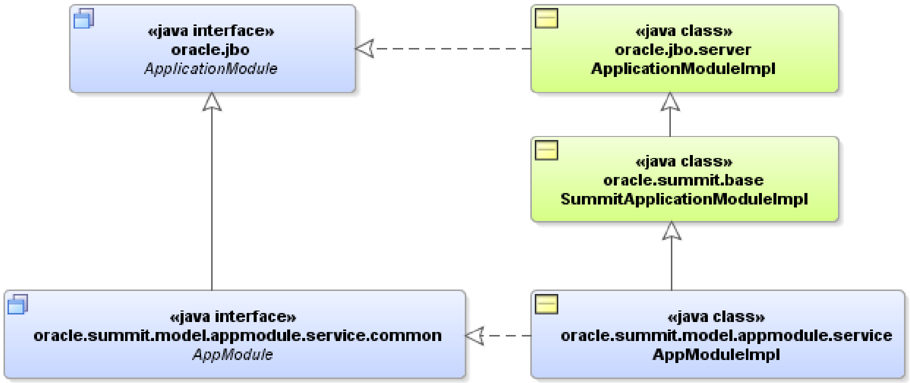
Selected:

- AppModule
  - createCustomer(String, String, Integer):long
  - doSomeCustomProcessing():void
  - findOrderAndCustomer(long):String
  - findOrderTotal(long):String
  - updateOrderStatus(long, String):void

What Happens When You Publish Custom Service Methods

When you publish custom service methods on the client interface, as shown in Figure 13-22, JDeveloper creates a Java interface with the same name as the application module in the `common` subpackage of the package in which your application module resides. For an application module named `AppModule` in the `summit.model` package, this interface will be named `AppModule` and reside in the `summit.model.common` package. The interface extends the base `ApplicationModule` interface in the `oracle.jbo` package, reflecting that a client can access all of the base functionality that your application module inherits from the `ApplicationModuleImpl` class.

Figure 13-22 Custom Client Interface Extends the Base ApplicationModule Interface



As shown in the following example, the `AppModule` interface includes the method signatures of all of the methods you've selected to be on the client interface of your application module.

```
package summit.model.appmodule.service.common;
import oracle.jbo.ApplicationModule;
// ---------------------------------------------------------------------
// ---    File generated by ADF Business Components Design Time.
// ---------------------------------------------------------------------
public interface AppModule extends ApplicationModule {
    String findOrderAndCustomer(long orderId);
    void updateOrderStatus(long orderId, String newStatus);
    String findOrderTotal (long orderId);
    long createCustomer(String name, String city, Integer countryId);
}
```

> **Note:**
>
> After adding new custom methods to the client interface, if your new custom methods do not appear to be available when you use JDeveloper's code insight context-sensitive statement completion, try recompiling the generated client interface. To do this, select the application module in the Applications window, select the source file for the interface of the same name in the Structure window, and choose **Rebuild** from the context menu. Consider this tip for new custom methods added to view objects and view rows as well.

How to Generate Client Interfaces for View Objects and View Rows

In addition to generating a client interface for your application module, it is also possible to generate strongly typed client interfaces for working with the other key client objects that you can customize. For example, you can add custom methods to the view object client interface and the view row client interface, respectively.

Before you begin:

It may be helpful to have an understanding of the purpose of the client interface. For more information, see Publishing Custom Service Methods to UI Clients.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Application Modules.

You will need to complete this task:

Create the desired view object, as described in Populating View Object Rows from a Single Database Table.

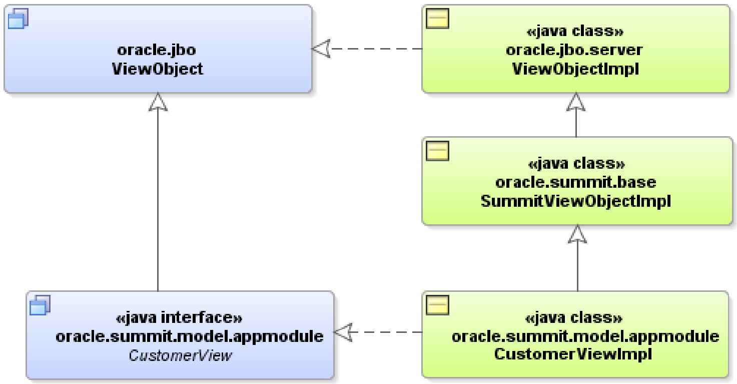
To publish a custom method on the client interface for view objects:

1. In the Applications window, double-click the view object for which you want to define the client interface.
2. In the overview editor, click the **Java** navigation tab and in the **Client Interface** section, click the **Edit java options** button.
3. In the Select Java Options dialog, click the **Generate View Object Class** and click the **Generate View Row Class**. Click **OK**.
4. In the overview editor, expand the **Client Interface** section and then click the **Edit view object client interface** button.
5. In the Edit Client Interface dialog, select one or more desired methods from the **Available** list and click the **Add** button to shuttle them into the **Selected** list.
6. In the overview editor, expand the **Client Row Interface** section and then click the **Edit view object client row interface** button.
7. In the Edit Client Interface dialog, select one or more desired methods from the **Available** list and click the **Add** button to shuttle them into the **Selected** list.
8. Click **OK**.

   The new methods will appear in the Java Classes page.

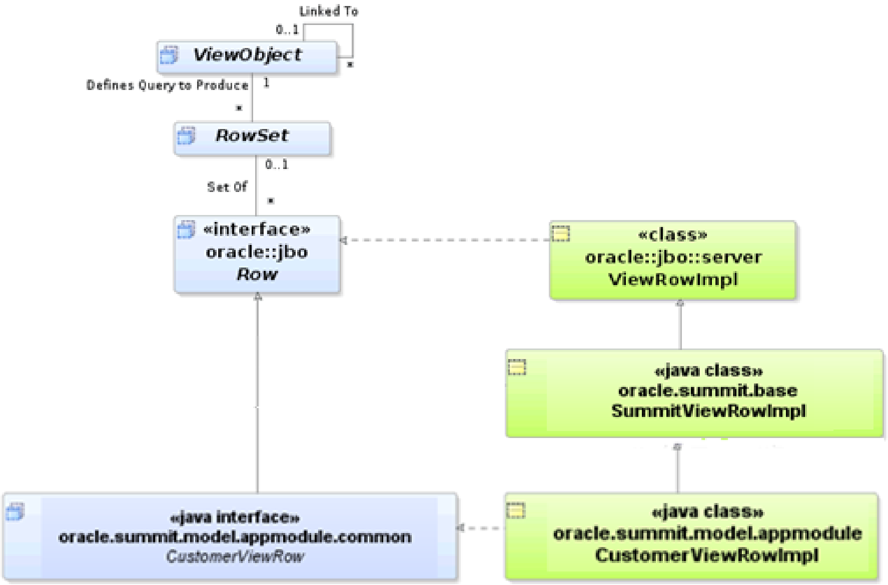
For example, if for the `CustomerView` view object in the `summit.model.views` package you were to enable the generation of a custom view object Java class and add one or more custom methods to the view object client interface, JDeveloper would generate the `CustomerViewImpl` class and `CustomerView` interface, as shown in Figure 13-23. As with the application module custom interface, notice that it gets generated in the `common` subpackage.

Figure 13-23 Custom View Object Interface Extends the Base ViewObject Interface



Likewise, if for the same view object you were to enable the generation of a custom view row Java class and add one or more custom methods to the view row client interface, JDeveloper would generate the `CustomerViewRowImpl` class and `CustomerViewRow` interface, as shown in Figure 13-24.

Figure 13-24 Custom View Row Interface Extends the Base Row Interface



How to Test Custom Service Methods Using the Oracle ADF Model Tester

You can test the methods of your custom application module in the Oracle ADF Model Tester after you have published them on the client interface, as described in Publishing Custom Service Methods to UI Clients.

Before you begin:

It may be helpful to have an understanding of the purpose of the client interface. For more information, see Publishing Custom Service Methods to UI Clients.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Application Modules.

You will need to complete this task:

Create the desired application modules, as described in How to Create an Application Module.

To test the service methods that you have published:

1. In the Applications window, right-click the application module and choose **Run**.

   Alternatively, choose **Debug** when you want to run the application in the Oracle ADF Model Tester with debugging enabled. The debugger process panel opens in the Log window and the various debugger windows. When debugging using the

Oracle ADF Model Tester, you can use these windows to view status message and exceptions, step in and out of source code, and manage breakpoints.

For information about receiving diagnostic messages specific to ADF Business Component debugging, see How to Enable ADF Business Components Debug Diagnostics.

2. To open the method testing panel for a method defined by a client interface, do one of the following:

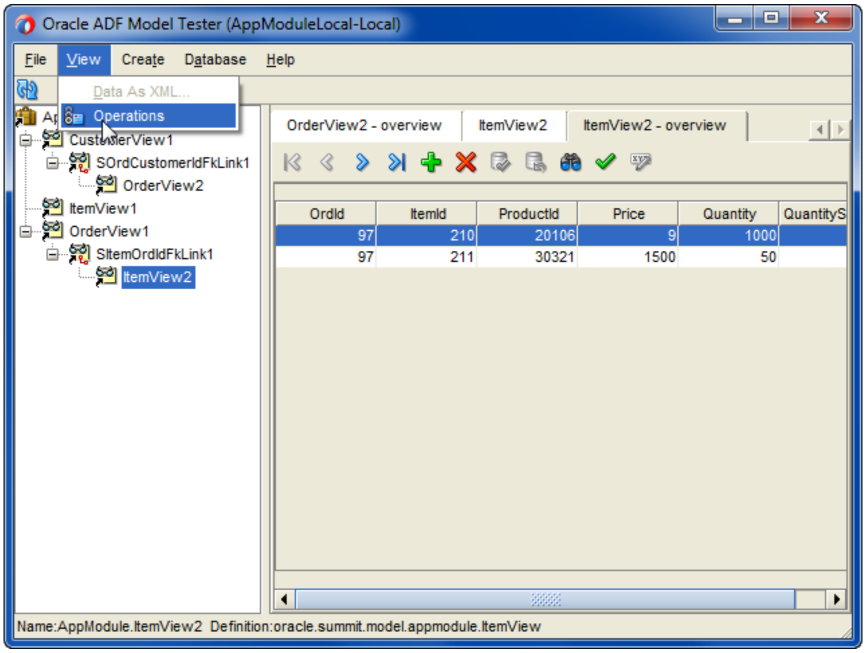
   - In the data model tree, select the application module node and in the main menu choose **View** and then **Operations** when you want to execute a method you published for the application module client interface. You can also double-click the application module node to display the method testing panel.
   - In the data model tree, select the desired view object node and in the main menu choose **View** and then **Operations** when you want to execute a method you published on the client interface for a view object. You can also right-click the view object node and choose **Operations**.

3. To open the method testing panel for a method defined by a client row interface for a view object row, expand the data model tree, right-click the desired view object node and choose **Show Table**. Then in the overview panel for the view instance, select the desired row, and in the main menu, choose **View** and then **Operations**.

   Do not select a master view instance in the data model tree since view row operations are not permitted on master view objects. Always select a detail view instance or a view instance that is not specified in a master-detail hierarchy, as shown in Figure 13-25.

> **Tip:**
>
> In the case of a detail view instance, you can open the master view instance to navigate to the detail with the desired row. The Oracle ADF Model Tester automatically synchronizes the data displayed in the open overview panel with the master view instance that you navigate to.

Figure 13-25 Menu Selection for View Row Operations in the Oracle ADF Model Tester



OrdId	ItemId	ProductId	Price	Quantity	QuantityS
97	210	20106	9	1000	
97	211	30321	1500	50	

Name:AppModule.ItemView2 Definition:oracle.summit.model.appmodule.ItemView

4. In the method panel, select the desired service method from the dropdown list, enter values to pass as method parameters, and click **Execute**.

   Notice that the method testing panel displays the parameter names to help you identify where to enter the values to pass. This is particularly useful when the method signature defines multiple parameters of the same data type.

   You can view the return value (if any) and test result. The result displayed in the Oracle ADF Model Tester will indicate whether or not the method executed successfully.

What You May Need to Know About Method Signatures on the Client Interface

You can include any custom method in the client interface that obeys these implementation rules:

- If the method has a non-`void` return type, the type must be serializable.
- If the method accepts any parameters, all their types must be serializable.
- If the method signature includes a throws clause, the exception must be an instance of `JboException` in the `oracle.jbo` package.

In other words, all the types in its method signature must implement the `java.io.Serializable` interface, and any checked exceptions must be `JboException` or its subclass. Your method can throw any unchecked exception — `java.lang.RuntimeException` or a subclass of it — without disqualifying the method from appearing on the application module's client interface.

Note that method signatures of type `java.util.List` are allowed as long as the implementing class for the interface is serializable. For example, `java.util.ArrayList` and `java.util.LinkedList` are both serializable implementing classes. The same requirement applies to element types within the collection. The ADF Business Components runtime will produce an error if you instantiate a class that implements the interface yet does not implement the `java.io.Serializable` interface.

> **Note:**
>
> If the method you've added to the application module class doesn't appear in the **Available** list, first verify that it doesn't violate any of the method implementation rules. If it seems like it should be a legal method, try recompiling the application module class before visiting the overview editor for the application module again.

What You May Need to Know About Passing Information from the Data Model

The private implementation of an application module custom method can easily refer to any view object instance in the data model using the generated accessor methods. By calling the `getCurrentRow()` method on any view object, it can access the same current row for any view object that the client user interface sees as the current row. As a result, while writing application module business service methods, you may not need to pass in parameters from the client. This is true if you would be passing in values only from the current rows of other view object instances in the same application module's data model.

For example, the custom application module method in the following example accepts no parameters. Internally, the `createOrderItem()` method calls `getGlobals().getCurrentRow()` to access the current row of the `Globals` view object instance. Then it uses the strongly typed accessor methods on the row to access the values of the `Description` and `LineItemId` attributes to set them as the values of corresponding attributes in a newly created `OrderItem` entity object row.

```
// In AppModuleImpl.java, createOrderItem() method
GlobalsRowImpl globalsRow = (GlobalsRowImpl)getGlobals().getCurrentRow();
newOrder.setDescription(globalsRow.getDescription());
newOrder.setLineItemId(globalsRow.getLineItemId());
```

Working Programmatically with an Application Module's Client Interface

The ADF Business Components application module client interface lets you work programmatically with application modules.

After publishing methods on your application module's client interface, you can invoke those methods from a client.

How to Work Programmatically with an Application Module's Client Interface

To work programmatically with an application module's client interface, do the following:

- Cast `ApplicationModule` to the more specific client interface.
- Call any method on the interface.

> **Note:**
>
> For simplicity, this section focuses on working only with the custom application module interface; however, the same downcasting approach works on the client to use a `ViewObject` interface as a view object interface like `Customers` or a `Row` interface as a custom view row interface like `CustomersRow`.

The following example illustrates a `TestClientEntity` class that puts these two steps into practice. You could also use the `main()` method of this class to test application module methods, as described in How to Test the Custom Application Module Using a Static Main Method. Here you use it to call all of the same methods from the client using the `AppModule` client interface.

> **Note:**
>
> If you work with your application module using the default `ApplicationModule` interface in the `oracle.jbo` package, you won't have access to your custom methods. Make sure to cast the application module instance to your more specific custom interface like the `AppModule` interface in this example.

The basic logic of the following example follows these steps:

1. Retrieve the total for order 1011.
2. Retrieve the name of the customer for order 1011.
3. Set the status of order 1011 to the value "Y".
4. Create anew customer supplying a null customer name.
5. Create a new customer with a customer name and display its newly assigned customer ID.

```
package oracle.summit.model.appmodule.client;

import oracle.jbo.client.Configuration;
import oracle.jbo.*;
import oracle.jbo.domain.Number;
import oracle.jbo.domain.*;
import oracle.summit.model.appmodule.service.common.AppModule;

public class TestClientCustomInterface {
  public static void main(String[] args) {
      String amDef = "oracle.summit.model.appmodule.service.AppModule";
      String config = "AppModuleLocal";

      /*
       * This is the correct way to use application custom methods
       * from the client, by using the application module's automatically-
       * maintained custom service interface.
       */
      AppModule service =

(AppModule)Configuration.createRootApplicationModule(amDef,config);
      String total = service.findOrderTotal(1011);
      System.out.println("Total for Order # 1011 = " + total);
      String custName = service.findOrderAndCustomer(1011);
      System.out.println("Customer for Order # 1011 = " + custName);
      try {
         service.updateOrderStatus(1011,"Y");
      }
      catch (JboException ex) {
         System.out.println("ERROR: "+ex.getMessage());
      }
      long id = 0;
      try {
         id = service.createCustomer(null, "Oakville", 14);
      }
      catch (JboException ex) {
         System.out.println("ERROR: "+ex.getMessage());
      }
      id = service.createCustomer("Oakville Curling Club", "Oakville", 14);
      System.out.println("New customer created successfully with id = " + id);
      Configuration.releaseRootApplicationModule(service,true);
  }
}
```

Running the custom application module class calls the `main()` method in the above example and produces the following output:

```
Total for Order # 1011 = 99.99
Customer for Order # 1011 = Customer: Zibbers, Location: Boston
ERROR: JBO-27014: You must provide a value for Name.
New customer created successfully with id = 133
```

Notice that the first attempt to call `findOrderAndCustomer()` with a null for the customer name raises an exception due to the built-in mandatory validation on the `Name` attribute of the `Customer` entity object.

What Happens at Runtime: How the Application Module's Client Interface is Accessed

Because the client layer accessing your application module will be located in the same tier of the Java EE architecture, the application module is deployed in what is known as **local mode**. In local mode, the client interface is implemented directly by your custom application module Java class. You access an application module in local mode whenever you access the application module in the web tier of a JavaServer Faces application.

How to Access an Application Module Client Interface in a Fusion Web Application

The `Configuration` class in the `oracle.jbo.client` package makes it very easy to get an instance of an application module for *testing*. This eases writing test client programs like the test client program described in Regression Testing with JUnit as part of the JUnit regression testing fixture.

> **Best Practice:**
>
> For Fusion web applications you should always work though the binding layer to access the application module. While developers may be tempted to use the class `createRootApplicationModule()` and `releaseApplicationModule()` methods *anywhere* to access an application module, the best approach in the view layer is to use the declarative features of the ADF Model layer.

When working with Fusion web applications using the ADF Model layer for data binding, JDeveloper configures a servlet filter in your user interface project called the `ADFBindingFilter`. It orchestrates the automatic acquisition and release of an appropriate application module instance based on declarative binding metadata, and ensures that the service is available to be looked up as a data control using a known **action binding** or iterator binding, specified by any page definition file in the user interface project. You may eventually want to read about the ADF **binding container**, **data controls**, **page definition files**, and **ADF bindings**, as described in Using ADF Model in a Fusion Web Application. For now, it is enough to realize that you can access the application module's client interface from this `DCBindingContainer` by naming an ADF action binding or an ADF iterator binding. You can reference the **binding context** and call methods on the custom client interface in a JSF managed bean, as shown in the following examples for an action binding and for an iterator binding.

To access the custom interface of your application module using an action binding, follow these basic steps (as illustrated in the following example):

1. Access the ADF binding container.
2. Find a named action binding. (Use the name of any available action binding in the page definition files of the user interface project.)
3. Get the data control by name from the action binding.
4. Access the application module data provider from the data control.
5. Cast the application module to its client interface.
6. Call any method on the client interface.

```
package oracle.summit.model.viewobjects;

import oracle.summit.model.appmodule.service.common.SummitAppModule;
import oracle.adf.model.binding.DCBindingContainer;
import oracle.adf.model.binding.DCDataControl;
import oracle.jbo.ApplicationModule;
import oracle.jbo.uicli.binding.JUCtrlActionBinding;
public class YourBackingBean {
  public String commandButton_action() {
    // Example using an action binding to get the data control
    public String commandButton_action() {
    // 1. Access the binding container
    DCBindingContainer bc = (DCBindingContainer)getBindings();
    // 2. Find a named action binding
    JUCtrlActionBinding action =
                    (JUCtrlActionBinding)bc.findCtrlBinding("SomeActionBinding");
    // 3. Get the data control from the iterator binding (or method binding)
    DCDataControl dc  = action.getDataControl();
    // 4. Access the data control's application module data provider
    ApplicationModule am = (ApplicationModule)dc.getDataProvider();
    // 5. Cast the AM to call methods on the custom client interface
    SummitAppModule service = (SummitAppModule)am;
    // 6. Call a method on the client interface
    service.doSomethingInteresting();
    return "SomeNavigationRule";
  }
}
```

To access the custom interface of your application module using an iterator binding, follow these basic steps (as illustrated in the following example):

1. Access the ADF binding container.
2. Find a named iterator binding. (Use the name of any iterator binding in the page definition files of the user interface project.)
3. Get the data control by name from the iterator binding.
4. Access the application module data provider from the data control.
5. Cast the application module to its client interface.
6. Call any method on the client interface.

```
package oracle.summit.model.viewobjects;

import oracle.summit.model.appmodule.service.common.SummitAppModule;
import oracle.adf.model.binding.DCBindingContainer;
import oracle.adf.model.binding.DCDataControl;
import oracle.adf.model.binding.DCIteratorBinding;
import oracle.jbo.ApplicationModule;
public class YourBackingBean {
  public String commandButton_action() {
    // Example using an iterator binding to get the data control
    public String commandButton_action() {
    // 1. Access the binding container
    DCBindingContainer bc = (DCBindingContainer)getBindings();
    // 2. Find a named iterator binding
    DCIteratorBinding iter = bc.findIteratorBinding("SomeIteratorBinding");
    // 3. Get the data control from the iterator binding
    DCDataControl dc  = iter.getDataControl();
    // 4. Access the data control's application module data provider
    ApplicationModule am = (ApplicationModule)dc.getDataProvider();
    // 5. Cast the AM to call methods on the custom client interface
    SummitAppModule service = (SummitAppModule)am;
    // 6. Call a method on the client interface
    service.doSomethingInteresting();
    return "SomeNavigationRule";
  }
}
```

These backing bean examples depend on the helper method shown in the following example.

```
public BindingContainer getBindings() {
{
        return BindingContext.getCurrent().getCurrentBindingsEntry();
}
```

If you create the backing bean class by overriding a button that is declaratively bound to an ADF action, then JDeveloper will automatically generate this method in your class. Otherwise, you will need to add the helper method to your class yourself.

Overriding Built-in Framework Methods

You can enhance the default behavior of an ADF Application Module by overriding built-in ADF methods for the Application Module Java class.

The `ApplicationModuleImpl` base class provides a number of built-in methods that implement its functionality. While Most Commonly Used ADF Business Components Methods provides a quick reference to the most common code that you will typically write, use, and override in your custom application module classes, this section focuses on helping you understand the basic steps to override one of these built-in framework methods to augment the default behavior.

How to Override a Built-in Framework Method

To override a built-in framework method for an application module, use the Override Methods dialog, which you open for the application module Java class from the main menu.

Before you begin:

It may be helpful to have an understanding of the application module base class. For more information, see Overriding Built-in Framework Methods.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Application Modules.

You will need to complete this task:

Create the desired application modules, as described in How to Create an Application Module.

To override an application module framework method:

1. In the Applications window, double-click the application module for which you want to override framework methods.
2. In the overview editor, click the **Java** navigation tab and then click the linked file name of the application module Java class that you want to customize.

   The class file opens in the source editor.
3. In the main menu, choose **Source** and then **Override Methods**.

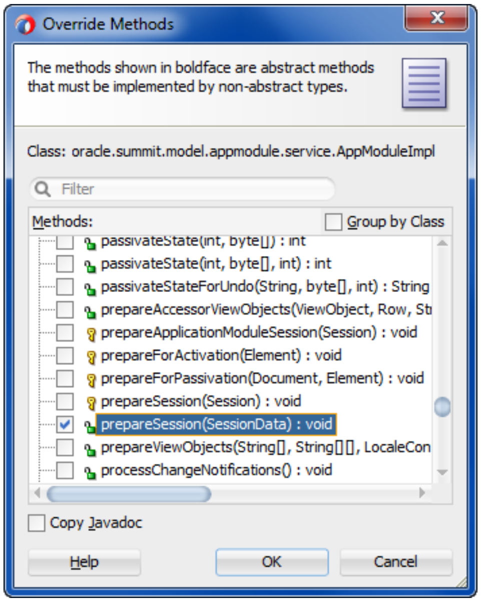
If the **Source** menu is not displayed, be sure that the desired Java class file is open and that the source editor is visible.

4. In the Override Methods dialog, scroll the list to locate the desired methods or type the first few letters of the method name to perform an incremental search.
5. Select one or more methods.

   The Override Methods dialog allows you to select any number of methods to override simultaneously.

   For example, if you wanted to override the application module's `prepareSession()` method to augment the default functionality when a new user session begins working with an application module service component for the first time, you would select the checkbox next to the `prepareSession(Session)` method, as shown in Figure 13-26.

Figure 13-26   Overriding a Built-in Framework Method



The methods shown in boldface are abstract methods that must be implemented by non-abstract types.

Class: oracle.summit.model.appmodule.service.AppModuleImpl

□ passivateState(int, byte[]) : int
□ passivateState(int, byte[], int) : int
□ passivateStateForUndo(String, byte[], int) : String
□ prepareAccessorViewObjects(ViewObject, Row, St
□ prepareApplicationModuleSession(Session) : void
□ prepareForActivation(Element) : void
□ prepareForPassivation(Document, Element) : void
□ prepareSession(Session) : void
☑ prepareSession(SessionData) : void
□ prepareViewObjects(String[], String[][], LocaleCon
□ processChangeNotifications() : void

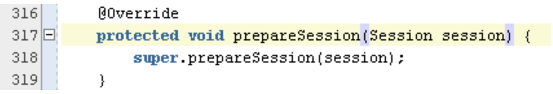
□ Copy Javadoc

6. Click **OK**.

What Happens When You Override a Built-in Framework Method

When you dismiss the Override Methods dialog, you return to the source editor with the cursor focus on the overridden method, as shown in Figure 13-27. Notice that the method appears with a single line that calls `super.prepareSession()`. This is the syntax in Java for invoking the default behavior that the base class would have normally performed for this method. By adding code before or after this line in the custom application module class, you can augment the default behavior before or after the default functionality.

Figure 13-27   Source Editor Margin Gives Visual Feedback About Overridden Methods



```
    @Override
    protected void prepareSession(Session session) {
        super.prepareSession(session);
    }
```

Also notice that when you override a method using the Override Methods dialog, the source editor inserts the JDK `@Override` annotation just before the overridden method. This causes the compiler to generate a compile-time error if the method in the application module class does not match the signature of any method in the superclass.

Be careful when you add method names to your class to override a method in the superclass; you must have the signature *exactly* the same as the base class method you want to override. Be sure to add the `@Override` annotation just before the method. This way, if your method does not match the signature of any method in the superclass, the compiler will generate a compile-time error. Also, when you write code for a method instead of calling the superclass implementation, you should have a thorough understanding of what built-in code you are suppressing or replacing.

How to Override prepareSession() to Set Up an Application Module for a New User Session

Since the `prepareSession()` method is invoked by the application module when it is used for the first time by a new user session, it's a useful method to override in your custom application module class to perform setup tasks that are specific to each new user that uses your application module. The following example illustrates an overridden `prepareSession()` method in the `AppModuleImpl` class that invokes a `setApplicationInfo()` helper method to specify database client information for the connected user.

```
public class AppModuleImpl extends SummitApplicationModuleImpl
                           implements AppModule {
...

    @Override
    protected void prepareSession(Session session) {
        super.prepareSession(session);
        setApplicationInfo ("AppModuleImpl", "prepareSession");
        String username = this.getUserPrincipalName();
        System.out.println(username);
    }

    protected void setApplicationInfo(String clientInfo, String clientIdentifier)
{
        DBTransactionImpl dbti = (DBTransactionImpl)getDBTransaction();
        CallableStatement statement =
            dbti.createCallableStatement("BEGIN "
          + "DBMS_APPLICATION_INFO.SET_CLIENT_INFO (client_info
=> :client_info);"
          + "DBMS_SESSION.SET_IDENTIFIER (:client_identifier);"
          + "END;", 0);
        try {
            statement.setString("client_info", clientInfo);
            statement.setString("client_identifier", clientIdentifier);
            statement.execute();
        } catch (SQLException sqlerr) {
            throw new JboException(sqlerr);
        } finally {
            try {
                if (statement != null) {
                    statement.close();
                }
            } catch (SQLException closeerr) {
                throw new JboException(closeerr);
            }
        }
    }
}
```

Calling a Web Service from an Application Module

Use the built-in web services wizard to create a proxy class, implement methods in this class, and use it to call web service methods in an ADF Application Module.

In a service-oriented architecture, your Oracle ADF application module may need to take advantage of functionality offered by a web service that is not based on an application module. A web service can be implemented in any programming language and can reside on any server on the network. Each web service identifies the methods in its API by describing them in a standard, language-neutral XML format. This XML document, whose syntax adheres to the Web Services Description Language (WSDL), enables JDeveloper to understand the names of the web service's methods, as well as the data types of the parameters they might expect and their eventual return value.

> **Note:**
>
> Application modules can also be exposed as web services so that they can be consumed across modules of the deployed Fusion web application. For details about reusing ADF Business Components using external services, see Creating SOAP Web Services with Application Modules.

JDeveloper's built-in web services wizards make this an easy task. Create a web service proxy class using the wizard, then call the service using method calls you add to a local Java object.

How to Call an External Service Programmatically

To call a web service from an application module, you create a web service proxy class for the service you want to invoke. A web service proxy is a generated Java class that represents the web service inside your application. It encapsulates the service URL of the web service and handles the lower-level details of making the call.

To work with a web service, you need to know the URL that identifies its WSDL document. If you have received the WSDL document as an email attachment, for example, and saved it to your local hard drive, the URL could be similar to:

```
file:///D:/temp/SomeService.wsdl
```

Alternatively, the URL could be an HTTP-based URL like:

```
http://someserver.somecompany.com/SomeService/SomeService.wsdl
```

Some web services make their WSDL document available by using a special parameter to modify the service URL. For example, a web service that expects to receive requests at the HTTP address of `http://someserver.somecompany.com/SomeService` might publish the corresponding WSDL document using the same URL with an additional parameter on the end, like this:

```
http://someserver.somecompany.com/SomeService?WSDL
```

Since there is no established standard, you will just need to know what the correct URL to the WSDL document is. With the URL information, you can then create a web service proxy class to call the service.

ADF Business Components services have URLs to the service of the following formats:

- On Integrated WebLogic Server, the URL has the format *`http://host:port/EJB-context-root/@WebService-name?WSDL`*, for example:

  ```
  http://localhost:8888/EJB-SummitService/SummitService?WSDL
  ```

- On Oracle WebLogic Server, the URL has the format *`http://host:port/context-root/@WebService-name?WSDL`*, for example:

  ```
  http://localhost:8888/SummitService/SummitService?WSDL
  ```

The web service proxy class presents a set of Java methods that correspond to the web service's public API. By using the web service proxy class, you can call any method in the web service in the same way as you work with the methods of any other local Java class.

To call a web service from an application module using a proxy class, you perform the following tasks:

1. Create a web service proxy class for the web service. To create a web service proxy class for a web service that you need to call, use the Create Web Service Client and Proxy wizard.
2. Implement the methods in the proxy class to access the desired web services.
3. Create an instance of the web service proxy class in your application module and invoke one or more methods on the web service proxy object.

Creating a Web Service Proxy Class to Programmatically Access the Service

To create a web service proxy class for a web service you need to call, use the Create Web Service Proxy wizard.

Before you begin:

It may be helpful to have an understanding of web services. For more information, see Calling a Web Service from an Application Module.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Application Modules.

To create a web service proxy class to programmatically access the service:

1. In the Applications window, right-click the project in which you want to create the web service proxy and choose **New** and then **Web Service Client and Proxy**.
2. In the Create Web Service Client and Proxy wizard, on the Select Web Service Description page, enter the URL for the WSDL of the service you want to call in your application, and then tab out of the field.

   When the wizard displays **Next** enabled, then JDeveloper has recognized and validated the WSDL document. If the **Next** button is not enabled, fix the problem after verifying the URL and repeat this step.

3. Continue through the pages of the wizard to specify details about the web service proxy.
4. Click **Finish**.

Calling the Web Service Proxy Template to Invoke the Service

After you create the web service proxy, you must implement the methods in the proxy class to access the desired web services.

Before you begin:

It may be helpful to have an understanding of web services. For more information, see Calling a Web Service from an Application Module.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Application Modules.

You will need to complete this task:

> Create the desired web service proxy class, as described in Creating a Web Service Proxy Class to Programmatically Access the Service.

To call the web service proxy template to invoke the service:

1. Open the proxy client class, called `*port_name*Client.java`, in the source editor, and locate the comment `// Add your own code to call the desired methods`, which is in the main method.
2. Add the appropriate code to invoke the web service.
3. Deploy the full set of client module classes that JDeveloper has generated, and reference this class in your application.

Calling a Web Service Method Using the Proxy Class in an Application Module

After you've generated the web service proxy class, you can use it inside a custom method of your application module, as shown in the following example. The method creates an instance of the web service proxy class and calls the web service method from the web service proxy class for the result.

```
// In YourModuleImpl.java
public void performSomeApplicationTask(String symbol) throws Exception {
  // application-specific code here
   :
  // Create an instance of the web service proxy class
  StockQuoteServiceSoapHttpPortClient svc =
            new StockQuoteServiceSoapHttpPortClient();
  // Call a method on the web service proxy class and get the result
  QuoteInfo quote = svc.quoteForSymbol(symbol);
  float currentPrice = quote.getPrice();
  // more application-specific code here
}
```

What Happens When You Create the Web Service Proxy

JDeveloper generates the web service proxy class in the package you've indicated with a name that reflects the name of the web service port it discovered in the WSDL document. The web service port name might be a human-readable name like `StockQuoteService`, or could be a less-friendly name like `StockQuoteServiceSoapHttpPort`. The port name is decided by the developer that published the web service you are using. If the port name of the service were `StockQuoteServiceSoapHttpPort`, for example, JDeveloper would generate a web proxy class named `StockQuoteServiceSoapHttpPortClient`.

The web service proxy displays in the Applications window as a single, logical node called *`WebServiceName`*`Proxy`. For example, the node for the `StockQuoteService` web service would appear in the Applications window with the name `StockQuoteServiceProxy`. As part of generating the proxy class, in addition to the main web service proxy class that you use to invoke the server, JDeveloper generates a number of auxiliary classes and interfaces. You can see these files in the Applications window under the *`WebServiceName`*`Proxy` node. The generated files are used as part of the lower-level implementation of invoking the web service.

The only auxiliary generated classes you need to reference are those created to hold structured web service parameters or return types. For example, imagine that the `StockQuoteService` web service has a `quoteForSymbol()` method that accepts one `String` parameter and returns a floating-point value indicating the current price of the stock. If the designer of the web service chose to return a simple floating-point number, then the web service proxy class would have a corresponding method like this:

```
public float quoteForSymbol(String symbol)
```

If instead the designer of the web service thought it useful to return multiple pieces of information as the result, then the service's WSDL file would include a named structure definition describing the multiple elements it contains. For example, assume that the service returns both the symbol name and the current price as a result. To contain these two data elements, the WSDL file might define a structure named `QuoteInfo` with an element named `symbol` of string type and an element named `price` of floating-point type. In this situation, when JDeveloper generates the web service proxy class, the Java method signature would instead look like this:

```
public QuoteInfo quoteForSymbol(String symbol)
```

The `QuoteInfo` return type references one of the auxiliary classes that comprises the web service proxy implementation. It is a simple bean whose properties reflect the names and types of the structure defined in the WSDL document. In a similar way, if the web service accepts parameters whose values are structures or arrays of structures, then you will work with these structures in your Java code using the corresponding generated beans.

How to Create a New Web Service Connection

After developing a web service proxy, you can generate additional connections for the proxy that you can use in testing and deployment situations. For example, you might want to create a connection that includes user name and password for testing purposes.

The connection information is stored in the `connections.xml` file along with the other connections in your application. This abstraction of the endpoint URL also allows you to edit the connection after deployment using Enterprise Manager without requiring modification to the client code.

Before you begin:

It may be helpful to have an understanding of web services. For more information, see Calling a Web Service from an Application Module.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Application Modules.

To create a new web service connection:

1. In the Applications window, right-click a web service proxy and choose **Create ADF Web Service Connection**.

   The New ADF Web Service Connection dialog displays the default settings for a connection associated with the selected proxy.

2. Modify the connection information as necessary, and click **OK**.

> **WARNING:**
>
> If you create a new web service connection with the same name as an existing connection, the existing connection will be overwritten with the new information.

After you create a new web service connection, you can modify your client to use this connection. You could use code similar to that shown in the following example to access the connection from your client.

```
Context ctx = ADFContext.getCurrent().getConnectionsContext();

WebServiceConnection wsc = (WebServiceConnection) ctx.lookup("MyAppModuleService");

MyAppModuleService proxy = wsc.getJaxWSPort(MyAppModuleService.class);
```

The argument that you pass to the `lookup()` method is the name that you gave to the web service connection. In this example, it is `MyAppModuleService`.

What Happens at Runtime: How the Web Service Proxy Handles a Web Service Invocation

When you invoke a web service from an application module, the web service proxy class handles the lower-level details of using the XML-based web services protocol described in SOAP. In particular, it does the following:

- Creates an XML document to represent the method invocation
- Packages any method arguments in XML
- Sends the XML document to the service URL using an `HTTP POST` request
- Unpackages the XML-encoded response from the web service

If the method you invoke has a return value, your code receives it as an appropriately typed object to work with in your application module code.

What You May Need to Know About Web Service Proxies

When you are implementing web service proxies in an application, you might want to use a try-catch block to handle web service exceptions or invoke an application module with a web service proxy class. The following sections contain additional information you might need to know about these and other features with regard to web service proxies.

Using a Try-Catch Block to Handle Web Service Exceptions

By using the generated web service proxy class, invoking a remote web service becomes as easy as calling a method in a local Java class. The only distinction to be aware of is that the web service method call could fail if there is a problem with the HTTP request involved. The method calls that you perform against a web service proxy should anticipate the possibility that the request might fail by wrapping the call with an appropriate `try...catch` block. The following example improves on the simpler example (shown in Calling a Web Service Method Using the Proxy Class in an Application Module) by catching the web service exception. In this case, it simply rethrows the error as a `JboException`, but you could implement more appropriate error handling in your own application.

```
// In YourModuleImpl.java
public void performSomeApplicationTask(String symbol) {
  // application-specific code here
  // :
  QuoteInfo quote = null;
  try {
    // Create an instance of the web service proxy class
    StockQuoteServiceSoapHttpPortClient svc =
               new StockQuoteServiceSoapHttpPortClient();
    // Call a method on the web service proxy class and get the result
    quote = svc.quoteForSymbol(symbol);
  }
  catch (Exception ex) {
    throw new JboException(ex);
  }
  float currentPrice = quote.getPrice();
  // more application-specific code here
}
```

Separating Application Module and Web Services Transactions

You will use some web services to access reference information. However, other services you call may modify data. This data modification might be in your own company's database if the service was written by a member of your own team or another team in your company. If the web service is outside your firewall, of course the database being modified will be managed by another company.

In either of these situations, it is important to understand that any data modifications performed by a web service you invoke will occur in their own distinct transaction, unrelated to the application module's current unit of work. For example, if you have invoked a web service that modifies data and then you later call `rollback()` to cancel the pending changes in the application module's current unit of work, this has no effect on the changes performed by the web service you called in the process. You may need to invoke a corresponding web service method to perform a compensating change to account for your rollback of the application module's transaction.

Setting Browser Proxy Information

If the web service you need to call resides outside your corporate firewall, you need to ensure that you have set the appropriate Java system properties to configure the use of an HTTP proxy server. The Java system properties to configure are:

- `http.proxyHost` — Set this to the name of the proxy server.
- `http.proxyPort` — Set this to the HTTP port number of the proxy server (often 80).
- `http.nonProxyHosts` — Optionally set this to a vertical-bar-separated list of servers not requiring the user of a proxy server (for example, `localhost|127.0.0.1|*.`*`yourcompany`*`.com`).

Within JDeveloper, you can configure an HTTP proxy server on the Web Browser and Proxy page of the Preferences dialog. When you run your application, JDeveloper includes appropriate `-D` command-line options to set these three system properties based on the settings you've indicated in this dialog.

Invoking Application Modules with a Web Service Proxy Class

If you use a web service proxy class to invoke an Oracle ADF service-based application module, you lose the ability to optimize the call when the calling component and the service you are calling are colocated. As an alternative, you can use the service interface approach described in Creating SOAP Web Services with Application Modules.

14 Sharing Application Module View Instances

This describes how to organize your ADF Business Components data model project to most efficiently share read-only data accessed from lookup tables or other static data source, such as a flat file. It describes the differences between ADF application modules that you may share at the application level and those that you may share at the session level.
This chapter includes the following sections:

About Shared Application Modules

You can create a shared ADF Business Components application module to allow multiple requests from multiple sessions to share a single application module. It helps prevent unnecessary overhead of repopulating data caches.

Web applications often utilize data that is required across sessions and does not change very frequently. An example of this type of **static data** might be displayed in the application user interface in a lookup list. Each time your application accesses the static data, you could incur an unnecessary overhead when the static data caches are repopulated from the database for each application session on every request. In order to optimize performance, a common practice when working with **ADF Business Components** is to cache the shared static data for reuse across sessions and requests. Creating a **shared application module** allows requests from multiple sessions to share a single application module instance which is managed by an application pool for the lifetime of the web server virtual machine.

Shared Application Module Use Cases and Examples

A **view accessor** in ADF Business Components is a value accessor object that points from an **entity object** attribute (or view object) to a destination **view object** or shared view instance in the same application workspace.

This ability to access view objects in different application modules makes **view accessors** particularly useful for:

- Validation rules that you set on the attributes of an entity object. For example, when the end user fills out a registration form, individual validation rules can verify the title, marital status, and contact code against lookup table data queried by view instances of the shared application module.
- List of Value (LOV) that you enable for the attribute of any view object. For example, to display a list of values to the end user at runtime.

Additional Functionality for Shared Application Modules

You may find it helpful to understand other **Oracle ADF** features before you start working with shared application modules. Following are links to other functionality that may be of interest.

- For details about configuring application module instances to improve runtime performance, see Using State Management in a Fusion Web Application and Tuning Application Module Pools.
- For a quick reference to the most common code that you will typically write, use, and override in your custom application module classes, see Most Commonly Used ADF Business Components Methods.
- For API documentation related to the `oracle.jbo` package, see the following Javadoc reference document:
  - *Java API Reference for Oracle ADF Model*

Sharing an Application Module Instance

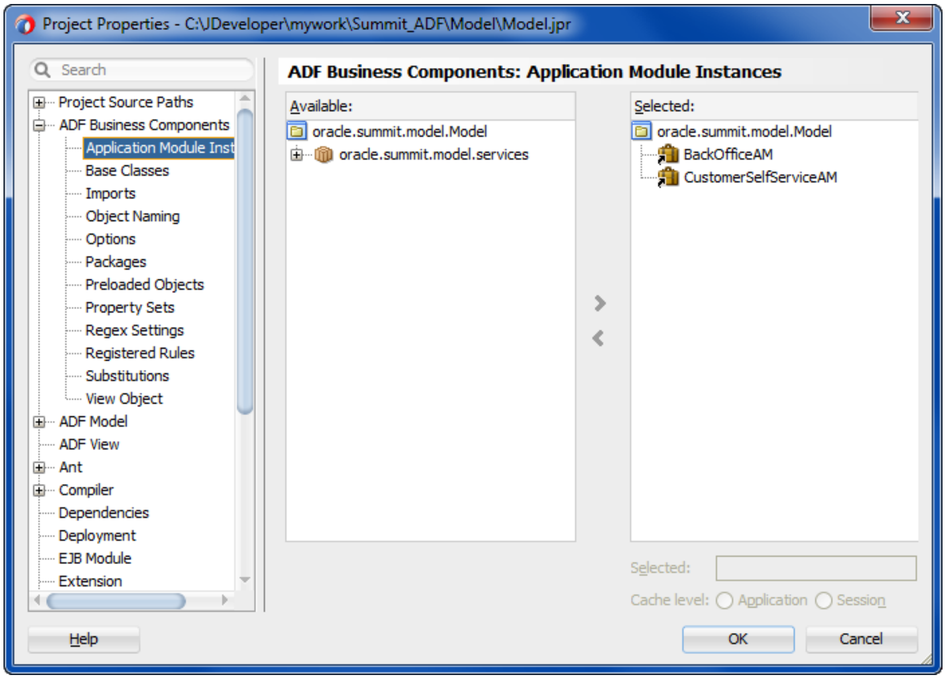
JDeveloper allows you to specify application-level or session-level ADF application module data model sharing.

Declarative support for shared data caches is available in JDeveloper through the Project Properties dialog. Creating a shared application module allows requests from multiple sessions to share a single application module instance which is managed by an application pool for the lifetime of the web server virtual machine.

> **Best Practice:**
>
> Use a shared application module to group view instances when you want to reuse lists of static data across the application. The shared application module can be configured to allow any user session to access the data or it can be configured to restrict access to just the UI components of a single user session. For example, you can use a shared application module to group view instances that access lookup data, such as a list of countries. The use of a shared application module allows all shared resources to be managed in a single place and does not require a scoped managed bean for this purpose.

As shown in Figure 14-1, the Project Properties dialog lets you specify application-level or session-level sharing of the application module's data model. In the case of **application-level sharing**, any HTTP user session will be able to access the same view instances contained in the shared application module. In contrast, the lifecycle of the **session-level shared** application module extends to an application module session (`SessionImpl`) that is in use by a single HTTP user session and applies to a single **root application module**. In this case, each distinct *root* application module used by a given HTTP user session will get its own distinct instance of a session-scoped shared application module. In other words, distinct root application modules used by the same HTTP session do not share data in a session-scoped shared application module.

Figure 14-1 Project Properties Dialog Defines Shared Application Module Instance



When you create the data model for the application module that you intend to share, be sure that the data in cached row sets will not need to be changed either at the application level or session level. For example, in the application-level shared application module, view instances should query only static data such as state codes or currency types. If a view object instance queries data that depends on the current user, then the query can be cached at the session level and shared by all components that reference the row-set cache. For example, the session-level shared application module might contain a view instance with data security that takes a manager as the current user to return the list of direct reports. In this case, the cache of direct reports would exist for the duration of the manager's HTTP user session. The ADF Business Components **application module pool** will re-create the session-scoped application module should an HTTP user session be assigned a recycled application module from the pool. This ensures that the duration of the session-scoped application module is tied to the HTTP session for as long as the HTTP session is able to continue to use the same root application module instance. Note that the cache of direct reports of the session-level shared application module cannot be accessed across distinct root application modules.

How to Create a Shared Application Module Instance

To create a shared application module instance, use the Project Properties dialog. You define a logical name for a distinct, separate root application module that will hold your application's read-only data.

Before you begin:

It may be helpful to have an understanding of lookup data. For more information, see Defining a Base View Object for Use with Lookup Tables.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Shared Application Modules.

You will need to complete this task:

> Create the application module that you will share, as described in How to Create an Application Module.

To create a shared application module instance:

1. In the Application window, right-click the project in which you want to create the shared application module and choose **Project Properties**.
2. In the Project Properties dialog, expand **ADF Business Components** and select **Application Module Instances**.
3. On the ADF Business Components: Application Module Instances page, select one of these tabs:
    - When you want to define the shared application module for the context of the application, select the **Application** tab.
    - When you want to define the shared application module for the context of the current user session, select the **Session** tab
4. In the **Available Application Modules** list, select the desired application module and shuttle it to the **Application Module Instances** list.
5. Assign the application module a unique instance name.

    The shared application module instance (of either scope) must have a unique instance name. Supplying a meaningful name will also help to clarify which shared application module instance a given usage is referencing.
6. Click **OK**.

What Happens When You Define a Shared Application Module

JDeveloper automatically creates the `*AppModuleName*Shared` configuration when you create an application module. The presence of this configuration in the `bc4j.xcfg` file informs JDeveloper that the application module is a candidate to be shared, and allows JDeveloper to display the application module in the **Available Application Modules** list of the Project Properties dialog's Application Module Usage page.

The `*AppModuleName*Shared` configuration sets these properties on the application module to enable sharing and help to maintain efficient use of the shared resource at runtime:

- `jbo.ampool.isuseexclusive` is set to `false` to specify that requests from multiple sessions can share a single instance of the application module, which is managed by the application pool for the lifetime of the web server virtual machine. When you do not enable application module sharing, JDeveloper sets the value `true` to repopulate the data caches from the database for each application session on every request.
- `jbo.ampool.maxpoolsize` is set to `1` (one) to specify that only a single application module instance will be created for the ADF Business Components application module pool. This setting enforces the efficient use of the shared application module resource and prevents unneeded multiple instances of the shared application module from being created at runtime.

You can view the shared application module's configuration by opening the application module in the overview editor and choosing **Configurations** from the navigation menu. JDeveloper saves the bc4j.xcfg file in the ./common subdirectory relative to the application module's XML document. If you remove the configuration or modify the values of the jbo.ampool runtime properties (isuseexclusive, maxpoolsize), the application module will not be available to use as a shared application module instance.

For example, if you look at the bc4j.xcfg file in the ./src/oracle/summit/model/services/common directory of the SummitADF application workspace, you will see the two named configurations for the BackOfficeAppModule application module, as shown in the following example. Specifically, the BackOfficeAppModuleShared configuration sets the jbo.ampool runtime properties on the shared application module instance. For more information about the ADF Business Components application module pooling and runtime configuration of application modules, see Tuning Application Module Pools.

```
<BC4JConfig version="11.1" xmlns="http://xmlns.oracle.com/bc4j/configuration">
...
   <AppModuleConfigBag
      ApplicationName="oracle.summit.model.services.BackOfficeAppModule">
      <AppModuleConfig
            name="BackOfficeAppModuleLocal"
            jbo.project="oracle.summit.model.Model"
            ApplicationName="oracle.summit.model.services.BackOfficeAppModule"
            DeployPlatform="LOCAL" JDBCName="summit_adf">
         <Database jbo.TypeMapEntries="Java"/>
         <Security

AppModuleJndiName="oracle.summit.model.services.BackOfficeAppModule"/>
      </AppModuleConfig>
      <AppModuleConfig
            name="BackOfficeAppModuleShared"
            jbo.project="oracle.summit.model.Model"
            ApplicationName="oracle.summit.model.services.BackOfficeAppModule"
            DeployPlatform="LOCAL" JDBCName="summit_adf">
         <AM-Pooling jbo.ampool.maxpoolsize="1"
                  jbo.ampool.isuseexclusive="false"/>
         <Database jbo.TypeMapEntries="Java"/>
         <Security

AppModuleJndiName="oracle.summit.model.services.BackOfficeAppModule"/>
      </AppModuleConfig>
   </AppModuleConfigBag>
</BC4JConfig>
```

Because the shared application module can be accessed by any data model project (based on ADF Business Components) in the same application workspace, JDeveloper maintains the scope of the shared application module in the ADF Business Components project configuration file (.jpx). This file is saved in the src directory of the project. For example, if you look at the Model.jpx file in the ./src directory of the application's Model project, you will see that the SharedLookupService application module's usage definition specifies SharedScope = 2, corresponding to application-level sharing, as shown in the following example. An application module that you set to session-level sharing will show SharedScope = 1.

```
<JboProject
   xmlns="http://xmlns.oracle.com/bc4j"
   Name="Model"
   SeparateXMLFiles="true"
   PackageName="oracle.summit.model">
   . . .
   <AppModuleUsage
      Name="SharedLookupService"
      FullName="oracle.summit.model.services.BackOfficeAppModule"
      ConfigurationName="oracle.summit.model.services.BackOfficeAppModuleShared"
      SharedScope="2"/>
</JboProject>
```

What You May Need to Know About Design Time Scope of the Shared Application Module

Defining the shared application module in the Project Properties dialog makes the application module's data model available to other data model projects of the same application workspace only. When you want to make the data model available beyond the application workspace, you can publish the data model as an ADF Library, as described in Reusing Application Components .

When viewing a **data control** usage from the `DataBindings.cpx` file in the Structure window, do not set the **Configuration** property to a shared application module configuration. By default, for an application module named ***AppModuleName***, the Property window will list the configurations named ***AppModuleName*****Shared** and ***AppModuleName*****Local**. At runtime, Oracle Application Development Framework (Oracle ADF) uses the shared configuration automatically when you configure an application as a shared application module, but the configuration is not designed to be used by an **application module data control** usage. For more information about data control usage, see Working with the DataBindings.cpx File .

What You May Need to Know About the Design Time Scope of View Instances of the Shared Application Module

You define view accessors on the business component definition for the data model project that will permit access to view instances of the shared application module. The view accessor lets you point from an entity object or view object definition in one data model project to a view object definition or view instance in a shared application module. For details about creating view accessors for this purpose, see Accessing View Instances of the Shared Service.

What You May Need to Know About Managing the Number of Shared Query Collections

Similar to the way application module pooling works in ADF Business Components, shared query collections are stored in a query collection pool. To manage the query collection pool, the ADF Business Components framework removes query collections based on a maximum idle time setting. This behavior limits the growth of the cache and prevents rarely used query collections from occupying memory space.

As in application module and connection pooling, a query collection pool monitor wakes up after a user-specified sleep interval and then initiates the cleanup operation. Any query collection that exceeds the maximum idle time (length of time since it was last used), will be removed from the pool.

You can change the default values for the maximum idle time for the shared query collection (default is 900000 ms/15 min) and the sleep period for its pool monitor (default is 1800000 ms/30 min). You use the overview editor for application module configurations (on the bc4j.xcfg file) to configure these values on the selected ***AppModuleName*****Shared** configuration.

To display the application module configuration overview editor, double-click the application module in the Applications window and, in the overview editor, select the **Configurations** navigation tab. Then, in the Configurations page of the overview editor, click the shared configuration hyperlink. In the application module configuration overview editor, select the **Properties** tab and click **Add Property** to select the following properties from the Add Property dialog and click **OK**.

- jbo.qcpool.monitorsleepinterval
- jbo.qcpool.maxinactiveage

Then, in the **Properties** list enter the desired idle time and sleep period in milliseconds:

- jbo.qcpool.monitorsleepinterval the time (ms) that the shared query collection pool monitor should sleep between pool checks.
- jbo.qcpool.maxinactiveage the maximum amount of time (ms) that a shared query collection may remain unused before it is removed from the pool.

What You May Need to Know About Shared Application Modules and Connection Pooling

The default connection behavior for all application modules is to allow each root application module to have its own database connection. When your application defines more than one shared application module, you can configure the application to nest them under the same transaction in order to use a single database connection. This optimization allows the shared application module instances to share the same connection and entity cache and reduces the database resources that the application uses. This is particularly useful for shared application modules cases because they are read only and have longer life than transactional application modules.

> **Best Practice:**
>
> Oracle recommends nesting shared application module instances under a single transaction to reduce the database resources required by the application. You can make this shared application module optimization by setting the jbo.shared.txn property to use the same transaction name (an arbitrary identifier you supply) for each shared application module configuration the application defines.

Set the jbo.shared.txn property using the overview editor for application module configurations (on the bc4j.xcfg file). To display the application module configuration overview editor, double-click the shared application module in the Applications window and, in the overview editor, select the **Configurations** navigation tab. Then, in the Configurations page of the overview editor, click the configuration hyperlink. In the application module configuration overview editor, select the **Properties** tab and click

Add Property to select the following property from the Add Property dialog and click **OK**.

- `jbo.shared.txn`

Then, in the **Properties** list enter the transaction name (where `SharedAMOptimization` is an arbitrary name) for the property:

- `jbo.shared.txn` = `SharedAMOptimization`

Repeat the `jbo.shared.txn` property setting using the same transaction name for each shared application module configuration that your application defines.

Currently, the application module configuration parameter `jbo.doconnectionpooling=true` is not supported for use with shared application modules. This feature is available to configure nonshared application modules when it is desirable to release JDBC connection objects to the database connection pool.

This feature is intentionally not supported for shared application modules to prevent decreases in performance that would result from managing state for shared access. Instead, the default use of `jbo.doconnectionpooling=false` is enforced.

The default connection pooling configuration ensures that each shared application module instance holds onto the JDBC connection object that it acquires from the pool until the application module instance is removed from the application module pool. For more information about the `jbo.doconnectionpooling` parameter and connection pool behavior, see Deployment Environment Scenarios and Pooling.

Defining a Base View Object for Use with Lookup Tables

Use an ADF application module to access static data that has been defined in a database lookup table. An ADF Business Components shared application module accesses such lookup tables to display static data.

When your application needs to display static data, you can define a shared application module with view instances that most likely will access lookup tables. A lookup table is a static, translated list of data to which the application refers. Lookup table data can be organized in the database in various ways. While it is possible to store related lookup data in separate tables, it is often convenient to combine all of the lookup information for your application within a single table. For example, a column `LOOKUP_TYPE` created for the `ORDERS_LOOKUPS` table would serve to partition one table that might contain diverse codes such as `FWK_TBX_YES_NO` for the values yes and no, `FWK_TBX_COUNTRY` for country names, and `FWK_TBK_CURRENCY` for the names of national currencies.

When your database schema organizes lookup data in a single database table, you want to avoid creating individual queries for each set of data. Instead, you will use the overview editor to define a single, base view object that maps the desired columns of the lookup table to the view object attributes you define. Since only the value of the `LOOKUP_TYPE` column will need to change in the query statement, you can add **view criteria** on the view object definition to specify a `WHERE` clause that will set the `LOOKUP_TYPE` value. In this way, your application encapsulates access to the lookup table data in a single view object definition that will be easy to maintain when a `LOOKUP_TYPE` value changes or your application needs to query additional lookup types.

How to Create a Base View Object Definition for a Lookup Table

The base view object that queries columns of the lookup table will be a read-only view object, since you do not need to handle updating data or require any of the benefits provided by entity-based view objects. (For a description of those benefits, see About View Objects.)

> **Note:**
>
> While read-only view objects you create to access lookup tables are ideal for inclusion in a shared application module, if you intend to share the view object in a shared application module instance, you must create the view object in the same package as the shared application module.

To create a read-only view object, use the Create View Object wizard, which is available from the New Gallery.

Before you begin:

It may be helpful to have an understanding of lookup data. For more information, see Defining a Base View Object for Use with Lookup Tables.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Shared Application Modules.

You will need to complete this task:

Create the application module that you will share, as described in How to Create an Application Module.

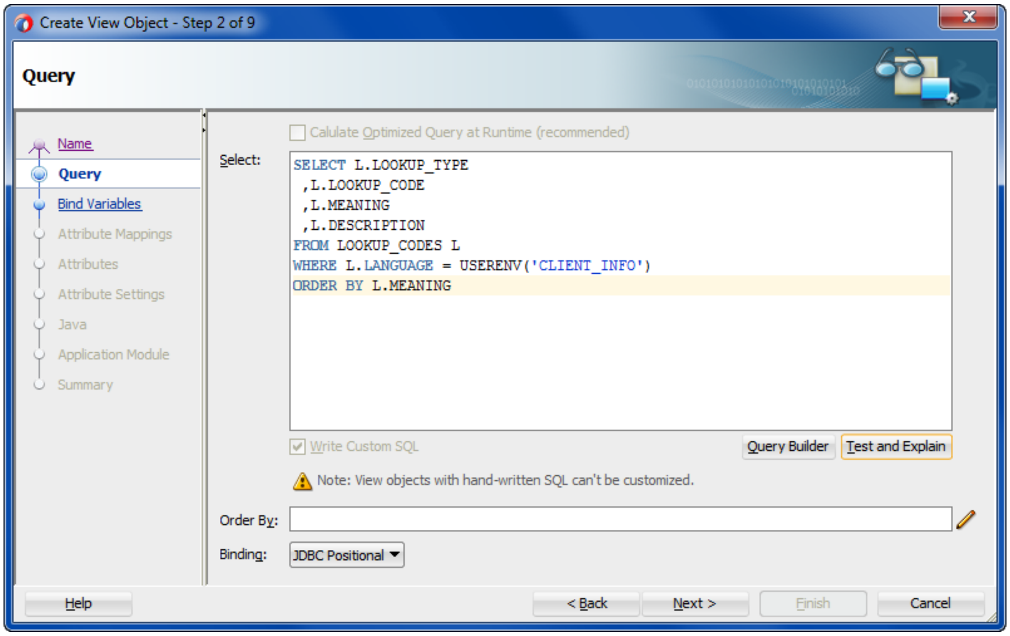
To create a base view object for a lookup table:

1. In the Application window, locate the shared application module, right-click its package node, and choose **New** and then **View Object**.
2. In the Create View Object wizard, on the Name page, enter a package name and a view object name.

   When naming the package, consider creating a separate package for the lookup.
3. Select **Custom SQL query** to indicate that you want this view object to manage data with read-only access and click **Next**.
4. On the Query page, enter your SQL statement directly into the **Select** text box.

   Your query names the columns of the lookup table, and should look similar to the SQL statement shown in Figure 14-2 which queries the `LOOKUP_CODE`, `MEANING`, and `DESCRIPTION` columns in the `LOOKUP_CODES` table.

Figure 14-2 Create View Object Wizard, SQL Query for Lookup Table



☐ Calulate Optimized Query at Runtime (recommended)

Select:

```
SELECT L.LOOKUP_TYPE
 ,L.LOOKUP_CODE
 ,L.MEANING
 ,L.DESCRIPTION
FROM LOOKUP_CODES L
WHERE L.LANGUAGE = USERENV('CLIENT_INFO')
ORDER BY L.MEANING
```

☑ Write Custom SQL

Note: View objects with hand-written SQL can't be customized.

Order By:

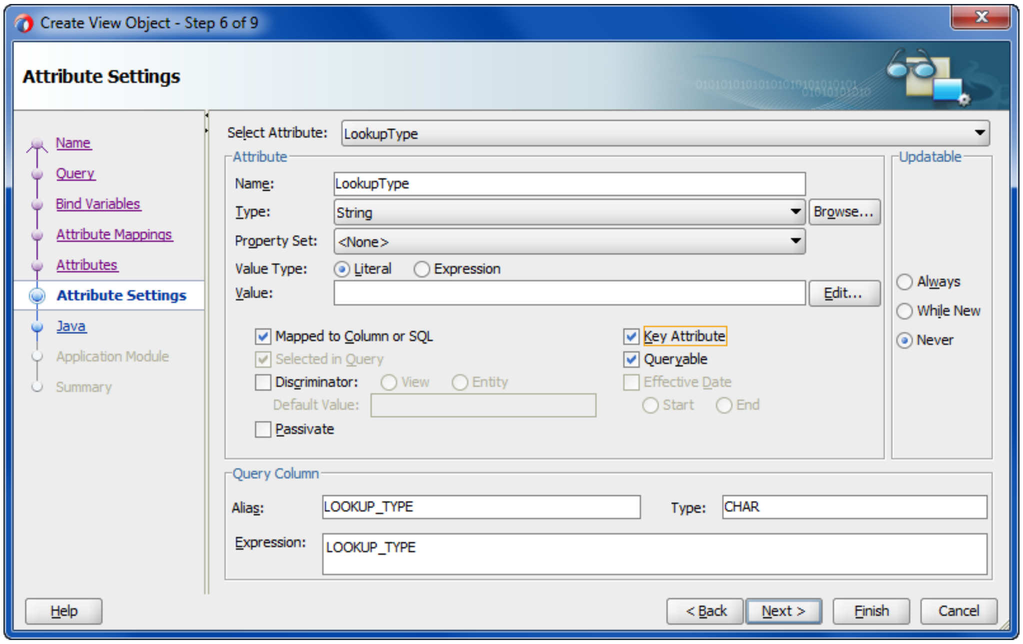
Binding: JDBC Positional

5. After entering the query statement, no other changes are required. Click **Next**.
6. On the Bind Variables page, click **Next**.
7. On the Attribute Mappings page, note the mapped view object attribute names displayed and click **Next**.

   By default, the wizard creates Java-friendly view object attribute names that correspond to the `SELECT` list column names.
8. On the Attribute Settings page, from the **Select Attribute** dropdown, select the attribute that corresponds to the primary key of the queried table and then enable the **Key Attribute** checkbox.

   Because the read-only view object is not based on an entity object, the Create View Object wizard does not define a key attribute by default. Failure to define the key attribute can result in unexpected runtime behavior for **ADF Faces** components with a data control based on the read-only view object collection. In the case of read-only view objects, define the key attribute, as shown in Figure 14-3.

Figure 14-3 Create View Object Wizard, Attribute Settings Page



Select Attribute: LookupType

Attribute

Name: LookupType

Type: String

Property Set: <None>

Value Type: ● Literal ○ Expression

Value:

☑ Mapped to Column or SQL

☑ Selected in Query

□ Discriminator: ○ View ○ Entity

Default Value:

□ Passivate

☑ Key Attribute

☑ Queryable

□ Effective Date

○ Start ○ End

Updatable

○ Always

○ While New

● Never

Query Column

Alias: LOOKUP_TYPE

Type: CHAR

Expression: LOOKUP_TYPE

9. If you want to rename individual attributes to use names that might be more appropriate, from the **Select Attributes** dropdown, choose the attribute and enter the desired name in the **Name** field. When you are finished, click **Next**.

   For example, you might rename the default attributes `LookupType` and `LookupCode` to `Type` and `Value` respectively. Changes you make to the view object definition will not change the underlying query.

10. On the Java page, click **Next**.

11. On the Application Module page, do not add an instance of the view object to the application module data model. Click **Finish**.

    The shared **application module data model** will include view instances based on view criteria that you add to the base view object definition. In this way, you do not need to create an individual view object to query each `LOOKUP_TYPE` value. For details about adding the view object instances to the data model, see Adding Master-Detail View Object Instances to an Application Module.

What Happens When You Create a Base View Object

When you create the view object definition for the lookup table, JDeveloper first describes the query to infer the following from the columns in the `SELECT` list:

- The Java-friendly view attribute names (for example, `LookupType` instead of `LOOKUP_TYPE`)

  By default, the wizard creates Java-friendly view object attribute names that correspond to the `SELECT` list column names.

- The SQL and Java data types of each attribute

JDeveloper then creates the XML document file that represents the view objects's declarative settings and saves it in the directory that corresponds to the name of its package. For example, the XML file created for a view object named `LookupsBaseVO` in the lookups package is ./lookups/LookupsBaseVO.xml under the project's source path.

To view the view object settings, expand the desired view object in the Application window, select the XML file under the expanded view object, and open the Structure window. The Structure window displays the list of definitions, including the SQL query and the properties of each attribute. To open the file in the editor, double-click the corresponding **.xml** node. As shown in the following example, the LookupsBaseVO.xml file defines one <SQLQuery> definition and one <ViewAttribute> definition for each mapped column. Without a view criteria to filter the query results, the view object query returns the LOOKUP_CODE, LOOKUP_MEANING, and LOOKUP_DESCRIPTION and maps them to view instance attribute values for Value, Name, and Description respectively. Key attributes are defined to ensure proper row set navigation when the base view object collection is bound to an ADF Faces component.

```
<ViewObject
    xmlns="http://xmlns.oracle.com/bc4j"
    Name="LookupsBaseVO"
    BindingStyle="OracleName"
    CustomQuery="true"
    PageIterMode="Full"
    UseGlueCode="false"
    FetchMode="FETCH_AS_NEEDED"
    FetchSize="500">
  <SQLQuery>
    <![CDATA[SELECT L.LOOKUP_TYPE
     ,L.LOOKUP_CODE
     ,L.MEANING
     ,L.DESCRIPTION
     FROM LOOKUP_CODES L
     WHERE L.LANGUAGE = USERENV('CLIENT_INFO')
     ORDER BY L.MEANING]]>
  </SQLQuery>
  <DesignTime>
    <Attr Name="_codeGenFlag2" Value="Access|VarAccess"/>
    <Attr Name="_isExpertMode" Value="true"/>
  </DesignTime>
  <ViewAttribute
    Name="Type"
    IsUpdateable="false"
    IsPersistent="false"
    IsNotNull="true"
    PrecisionRule="true"
    Precision="255"
    Type="java.lang.String"
    ColumnType="VARCHAR2"
    AliasName="LOOKUP_TYPE"
    Expression="LOOKUP_TYPE"
    SQLType="VARCHAR">
    <DesignTime>
      <Attr Name="_DisplaySize" Value="30"/>
    </DesignTime>
    ...
  </ViewAttribute>
  <ViewAttribute
    Name="Value"
    IsUpdateable="false"
    IsPersistent="false"
    IsNotNull="true"
    PrecisionRule="true"
    Precision="30"
    Type="java.lang.String"
    ColumnType="VARCHAR2"
    AliasName="LOOKUP_CODE"
    Expression="LOOKUP_CODE"
    SQLType="VARCHAR">
    <DesignTime>
      <Attr Name="_DisplaySize" Value="30"/>
    </DesignTime>
    ...
  </ViewAttribute>
  <ViewAttribute
    Name="Name"
    IsUpdateable="false"
    IsPersistent="false"
    IsNotNull="true"
    PrecisionRule="true"
    Precision="80"
    Type="java.lang.String"
    ColumnType="VARCHAR2"
    AliasName="MEANING"
    Expression="MEANING"
    SQLType="VARCHAR">
    <DesignTime>
      <Attr Name="_DisplaySize" Value="80"/>
    </DesignTime>
    ...
  </ViewAttribute>
  <ViewAttribute
    Name="Description"
    IsUpdateable="false"
    IsPersistent="false"
    PrecisionRule="true"
    Precision="240"
    Type="java.lang.String"
    ColumnType="VARCHAR2"
    AliasName="DESCRIPTION"
    Passivate="true"
    Expression="DESCRIPTION"
    SQLType="VARCHAR">
    <DesignTime>
      <Attr Name="_DisplaySize" Value="240"/>
    </DesignTime>
    ...
  </ViewAttribute>
  <AttrArray Name="KeyAttributes">
    <Item Value="Type"/>
    <Item Value="Value"/>
  </AttrArray>
. . .
</ViewObject>
```

How to Define the WHERE Clause of the Lookup View Object Using View Criteria

You create named view criteria definitions in the data model project when you need to filter view object results. View criteria that you define at design time can participate in UI scenarios that require filtering of data.

Use the Edit View Criteria dialog to create the view criteria definition for the lookup base view object you defined to query the lookup table. The editor lets you build a `WHERE` clause using attribute name instead of the target view object's corresponding SQL column names. The resulting definition will include:

- One view criteria row consisting of one view criteria group, with a single view criteria item used to define the lookup view object's `Type` attribute.
- The view criteria item will consist of an `Type` attribute name, the Equal operator, and the value of the `LOOKUP_TYPE` that will filter the query results.

Because a single view criteria is defined, no logical conjunctions are needed to bracket the `WHERE` clause conditions.

Before you begin:

It may be helpful to have an understanding of lookup data. For more information, see Defining a Base View Object for Use with Lookup Tables.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Shared Application Modules.

You may also find it helpful to have an understanding of view criteria. For more information, see Working with Named View Criteria.

You will need to complete this task:

Create the base view object for the lookup data, as described in How to Create a Base View Object Definition for a Lookup Table.

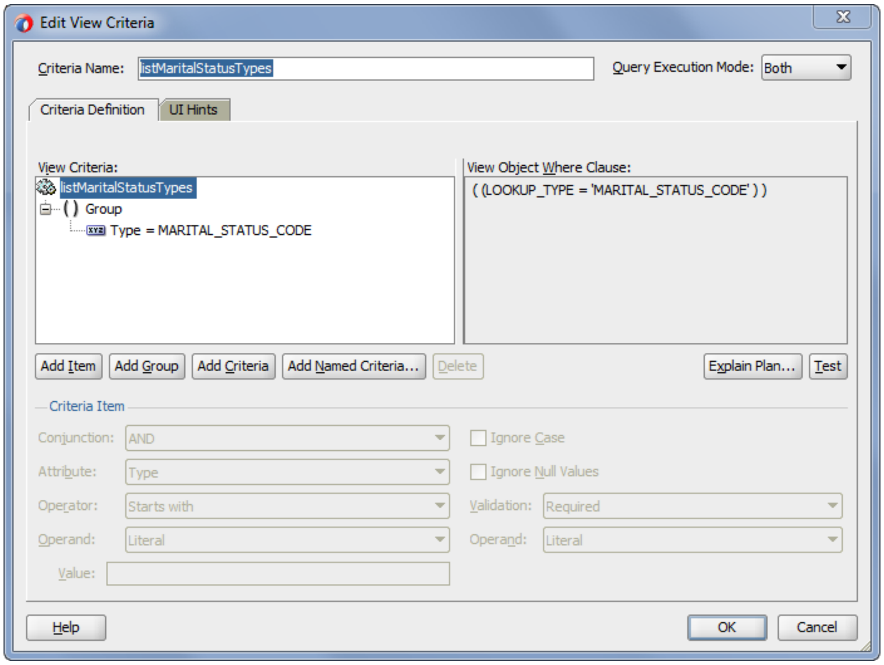
To create LOOKUP_TYPE view criteria for the lookup view object:

1. In the Application window, double-click the lookup base view object you defined.
2. In the overview editor, click the **View Criteria** navigation tab and click the **Create New View Criteria** button.
3. In the Create View Criteria dialog, on the View Criteria page, click the **Add Item** button to add a single criteria item to the view criteria group.
4. In the Criteria Item panel, define the criteria item as follows:
   - Choose **Type** as the attribute (or other name that you defined for the attribute the view object maps to the `LOOKUP_TYPE` column).
   - Choose **Equals** as the operator.
   - Keep **Literal** as the operand choice and enter the value name that defines the desired type. For example, to query the marital status codes, you might enter the value `MARITAL_STATUS_CODE` corresponding to the `LOOKUP_TYPE` column.

   Leave all other settings unchanged.

   The view object `WHERE` clause shown in the editor should display a simple criteria similar to the one shown in Figure 14-4, where the value `MARITAL_STATUS_CODE` is set to filter the `LOOKUP_TYPE` column.
5. Click **OK**.
6. Repeat this procedure to define one view criteria for each `LOOKUP_TYPE` that you wish to query.

Figure 14-4 Edit View Criteria Dialog with Lookup View Object View Criteria Specified



Criteria Name: listMaritalStatusTypes
Query Execution Mode: Both

View Criteria:
listMaritalStatusTypes
( ) Group
Type = MARITAL_STATUS_CODE

View Object Where Clause:
((LOOKUP_TYPE = 'MARITAL_STATUS_CODE'))

Conjunction: AND
Attribute: Type
Operator: Starts with
Operand: Literal
Value:
□ Ignore Case
□ Ignore Null Values
Validation: Required
Operand: Literal

What Happens When You Create a View Criteria with the Editor

The Create View Criteria dialog in JDeveloper lets you easily create view criteria and save them as named definitions. These named view criteria definitions add metadata to the target view object's own definition. Once defined, named view criteria appear by name in the View Criteria page of the overview editor for the view object.

JDeveloper then creates the XML document file that represents the view objects's declarative settings and saves it in the directory that corresponds to the name of its package. For example, the XML file created for a view object named `LookupsBaseVO` in the `lookups` package is `./lookups/LookupsBaseVO.xml` under the project's source path.

To view the view criteria, expand the desired view object in the Application window, select the XML file under the expanded view object, open the Structure window, and expand the View Criteria node. As shown in the following example, the `LookupsBaseVO.xml` file specifies the `<ViewCriteria>` definition that allows the `LookupsBaseVO` to return only the marital types. Other view criteria added to the `LookupsBaseVO` are omitted from this example for brevity.

```
<ViewObject
    xmlns="http://xmlns.oracle.com/bc4j"
    Name="LookupsBaseVO"
    BindingStyle="OracleName"
    CustomQuery="true"
    PageIterMode="Full"
    UseGlueCode="false">
  <SQLQuery>
    <![CDATA[SELECT L.LOOKUP_TYPE
     ,L.LOOKUP_CODE
     ,L.MEANING
     ,L.DESCRIPTION
     FROM LOOKUP_CODES L
     WHERE L.LANGUAGE = SYS_CONTEXT('USERENV','LANG')
     ORDER BY L.MEANING]]>
  </SQLQuery>
      ...
  <ViewCriteria
     Name="listMaritalStatusTypes"
     ViewObjectName="oracle.summit.model.lookups.LookupsBaseVO"
     Conjunction="AND"
     Mode="3"
     <Properties>
       <CustomProperties>
        <Property
          Name="autoExecute"
          Value="false"/>
        <Property
          Name="showInList"
          Value="true"/>
        <Property
          Name="mode"
          Value="Basic"/>
       </CustomProperties>
     </Properties>
     <ViewCriteriaRow
       Name="vcrow24">
       <ViewCriteriaItem
         Name="Type"
         ViewAttribute="Type"
         Operator="="
         Conjunction="AND"
         Value="MARITAL_STATUS_CODE"
         Required="Optional"/>
     </ViewCriteriaRow>
</ViewCriteria>
```

What Happens at Runtime: How a View Instance Accesses Lookup Data

When you create a view instance based on a view criteria, the next time the view instance is executed it augments its SQL query with an additional WHERE clause predicate corresponding to the view criteria that you've populated in the view criteria rows.

Accessing View Instances of the Shared Service

Use ADF Business Components view accessors to access view instances across shared application modules. You can set up validation rules or LOVs and use view accessors to access shared view instances.

View accessors in ADF Business Components are value accessor objects that point from an entity object attribute (or view object) to a destination view object or shared view instance in the same application workspace. The view accessor returns a row set that by default contains all rows from the destination view object. You can optionally filter this row set by applying view criteria to the view accessor. The base entity object or view object on which you create the view accessor and the destination view object need not be in the same project or application module, but they must be in the same application workspace.

Because view accessors give you the flexibility to reach across application modules to access the queried data, they are ideally suited for accessing view instances of shared application modules. For details about creating a data model of view instances for a shared application module, see How to Create a Shared Application Module Instance.

This ability to access view objects in different application modules makes view accessors particularly useful for:

- Validation rules that you set on the attributes of an entity object. In this case, the view accessor derives the validation rule's values from lookup data corresponding to a view instance attribute in the shared application module.
- List of Value (LOV) that you enable for the attribute of any view object. In this case, the view accessor derives the list of values from lookup data corresponding to a view instance attribute in the shared application module.

Validation rules with accessors are useful when you do not want the UI to display a list of values to the user, but you still need to restrict the list of valid values. Alternatively, consider defining an LOV for view object attributes to simplify the task of working with list controls in the user interface. Because you define the LOV on the individual attributes of business components, you can customize the LOV usage for an attribute once and expect to see the list control in the form wherever the attribute appears.

How to Create a View Accessor for an Entity Object or View Object

View accessors provide the means to access a data source independent of the application module. View accessors can be defined at the level of the entity object or individual view objects. However, because at runtime view accessor results are often filtered depending on the usage involved, it is recommended that you create unique view accessors for each usage in your application.

> **Best Practice:**
>
> Oracle recommends creating unique view accessors whenever your application needs to expose an LOV-enabled attribute. Reusing view accessors to create multiple list of values is discouraged because LOV results are often filtered at runtime. For example, the results of a saved search will filter the row set of the target view object until the end user unapplies the search criteria. Consequently, view accessors that get applied to this same destination view object will have their results filter too. To ensure the view accessor always returns the intended row set at runtime, create unique view accessors for each usage.

Defining view accessors on the entity object should be used carefully since view objects that you create based on the entity object will inherit the view accessors of their base entity objects. While defining the view accessor once on the entity object itself allows you to reuse the same view accessor, the view accessor must not be used in different application scenarios. If you intend to define validation rules for the entity object attributes and create LOV-enabled attributes for that entity object's view object, it is recommended that you create separate view accessors.

For example, assume the AddressEO entity object defines the Shared_CountriesVA view accessor and the AddressesVO view object inherits this view accessor. In this case, defining the view accessor on the entity object is useful: the accessor for AddressEO defines a validation rule on the CountryId attribute. But a different view accessor for AddressesVO should be used to enable an LOV on its CountryId attribute.

When you create a view accessor that accesses a view instance from a shared application module, you may want to use a prefix like Shared_ to name the view accessor. This naming convention will help you identify the view accessor when you need to select it for the entity object or view object.

Before you begin:

It may be helpful to have an understanding of view accessors. For more information, see Accessing View Instances of the Shared Service.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Shared Application Modules.

You will need to complete these tasks:

- Create the entity object or view object that you want to access, as described in Creating a Business Domain Layer Using Entity Objects and Defining SQL Queries Using View Objects.
- Enable application module sharing, as described in How to Create a Shared Application Module Instance.
- Create the base view object for the lookup data, as described in How to Create a Base View Object Definition for a Lookup Table.
- You can optionally refine the list returned by a view accessor by applying view criteria that you define on the view object. To create view criteria for use with a view accessor, see How to Define the WHERE Clause of the Lookup View Object Using View Criteria.

To create the view accessor:

1. In the Application window, double-click the entity object or view object on which you want to define the view accessor.

   Whether you create the view accessor on the entity object or on the view object will depend on the view accessor's intended usage. Generally, creating view accessors on the entity object ensures the widest possible usage.

2. In the overview editor, click the **Accessors** navigation tab and then, in the View Accessors section, click the **Create New View Accessors** button to add the accessor to the entity object or view object definition you are currently editing.

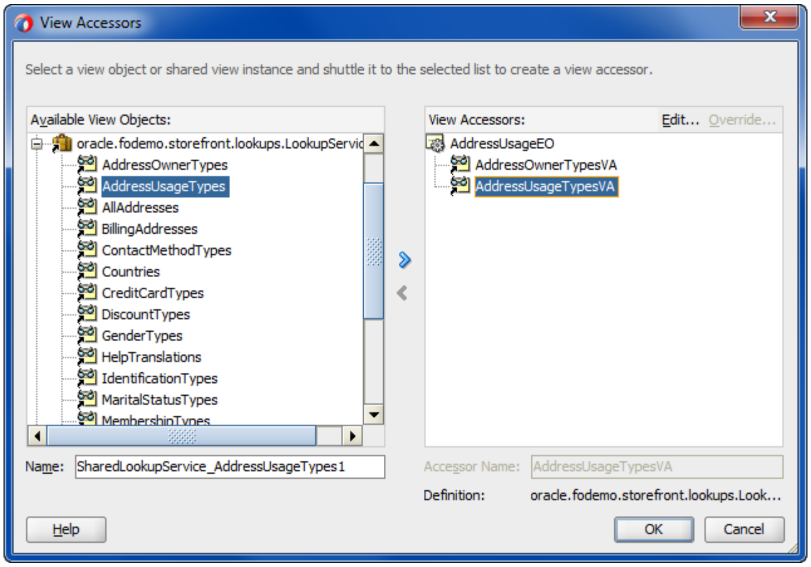
3. In the View Accessors dialog, select the view instance name you created for your lookup table from the shared application module node and shuttle it to the view accessors list.

   The dialog will display all view objects and view instances from your application. For example, the View Accessors dialog in Figure 14-5 shows the shared application module LookupServiceAM with the list of view instances, as shown .

   By default, the view accessor you create will display the same name as the view object instance (or will have an integer appended when it is necessary to distinguish it from a child view object of the same name). You can edit **Accessor Name** to give it a unique name.

For example, the View Accessors dialog in Figure 14-5 shows the view accessor `AddressUsageTypesVA` for the `AddressUsageTypes` view instance selection in the shared application module `LookupServiceAM`. This view accessor is created on the base entity object `AddressUsagesEO` and accesses the row set of the `AddressUsageTypes` view instance.

Figure 14-5 Defining a View Accessor on an Entity Object



View Accessors

Select a view object or shared view instance and shuttle it to the selected list to create a view accessor.

Available View Objects:

- oracle.fodemo.storefront.lookups.LookupServic
  - AddressOwnerTypes
  - AddressUsageTypes
  - AllAddresses
  - BillingAddresses
  - ContactMethodTypes
  - Countries
  - CreditCardTypes
  - DiscountTypes
  - GenderTypes
  - HelpTranslations
  - IdentificationTypes
  - MaritalStatusTypes
  - MembershipTypes

View Accessors:

- AddressUsageEO
  - AddressOwnerTypesVA
  - AddressUsageTypesVA

Name: SharedLookupService_AddressUsageTypes1

Accessor Name: AddressUsageTypesVA

Definition: oracle.fodemo.storefront.lookups.Look...

4. If you want to apply an existing view criteria to filter the accessor, with the view accessor selected in the overview editor, click the **Edit** icon.

   In the Edit View Accessor dialog, click **Edit** and perform the following steps to apply the view criteria:

   a. Select the view criteria that you want to apply and shuttle it to the **Selected** list.

      You can add additional view criteria to apply multiple filters (a logical AND operation will be performed at runtime).

   b. Enter the attribute name for the bind variable that defines the controlling attribute for the view accessor row set.

      Unlike view criteria that you set directly on a view object (to create a view instance, for example), the controlling attribute of the view accessor's view criteria derives the value from the view accessor's base view object.

   c. Click **OK**.

5. In the View Accessors dialog, click **OK**.

How to Validate Against the Attribute Values Specified by a View Accessor

View accessors that you create to access the view rows of a destination view object may be used to verify data that your application solicits from the end user at runtime. For example, when the end user fills out a registration form, individual validation rules can verify the title, marital status, and contact code against lookup table data queried by view instances of the shared application module.

You can apply view accessors you have defined on the entity object to these built-in declarative validation rules:

- The Compare validator performs a logical comparison between an entity attribute and a value. When you specify a view accessor to determine the possible values, the compare validator applies the `Equals`, `NotEquals`, `GreaterThan`, `LessThan`, `LessOrEqualTo`, `GreaterOrEqualTo` operator you select to compare against the values returned by the view accessor.
- The List validator compares an entity attribute against a list of values. When you specify a view accessor to determine the valid list values, the List validator applies an `In` or `NotIn` operator you select against the values returned by the view accessor.
- The Collection validator performs a logical comparison between an operation performed on a collection attribute and a value. When you specify a view accessor to determine the possible values, the Collection validator applies the Sum, Average, Count, Min, Max operation on the selected collection attribute to compare against the values returned by the view accessor.

Validation rules that you define to allow runtime validation of data for entity-based view objects are always defined on the attributes of the entity object. You use the editor for the entity object to define the validation rule on individual attributes. Any view object that you later define that derives from an entity object with validation rules defined will automatically receive attribute value validation.

Before you begin:

It may be helpful to have an understanding of view accessors. For more information, see Accessing View Instances of the Shared Service.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Shared Application Modules.

You will need to complete this task:

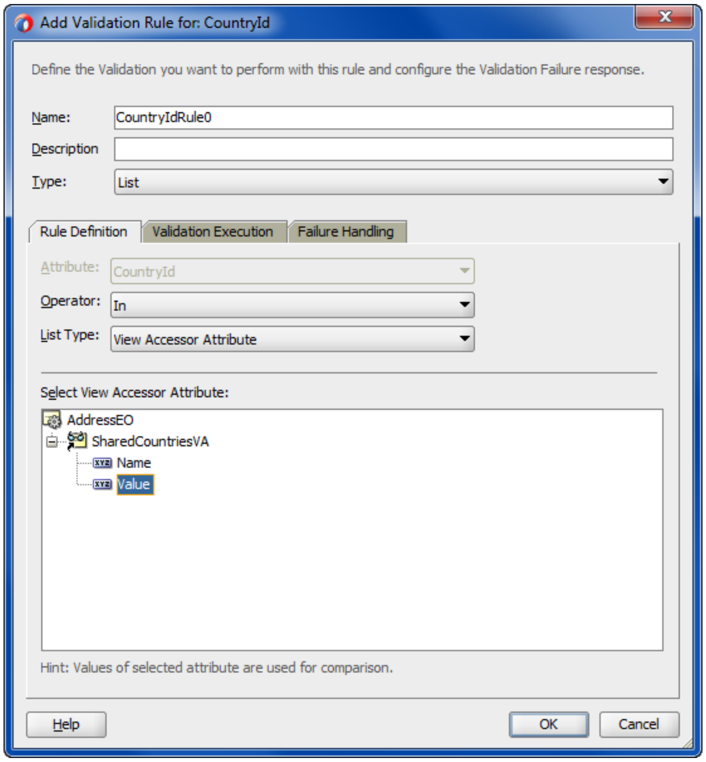
> Create the desired entity object, as described in How to Create Multiple Entity Objects and Associations from Existing Tables.

To validate against a view accessor comparison, list, or collection type:

1. In the Application window, double-click the desired entity object.
2. In the overview editor, click the **Business Rules** navigation tab.
3. In the Business Rules page, expand the entity object, select the attribute to be validated, and then click the **Create new validator** button to add the validation rule to the entity object attribute.
4. In the Add Validation Rule dialog, in the **Rule Type** dropdown list, select **Compare**, **List**, or **Collection**.
5. Make the selections required by the validator selection.
6. In the **Compare With** or **List Type** dropdown list, select **View Accessor Attribute**.
7. In the **Select View Accessor Attribute** group box, expand the desired view accessor from the shared service and select the attribute you want to provide as validation.

Figure 14-6 shows what the dialog looks like when you use a List validator to select a view accessor attribute.

Figure 14-6 List Validator Using a View Accessor



Add Validation Rule for: CountryId

Define the Validation you want to perform with this rule and configure the Validation Failure response.

Name: CountryIdRule0
Description:
Type: List

Attribute: CountryId
Operator: In
List Type: View Accessor Attribute

Select View Accessor Attribute:

Hint: Values of selected attribute are used for comparison.

8. Click the **Failure Handling** tab and enter a message that will be shown to the user if the validation rule fails.
9. Click **OK**.

What Happens When You Define a View Accessor Validator

When you use a List validator, a `<ListValidationBean>` tag is added to an entity object's XML file. The following example shows the XML code for the `CountryId` attribute in the `Address` entity object. A List validator has been used to validate the user's entry against the list of country ID values as retrieved by the view accessor from the `Countries` view instance.

```
<Attribute
   Name="CountryId"
   IsNotNull="true"
   Precision="2"
   ColumnName="COUNTRY_ID"
   Type="java.lang.String"
   ColumnType="CHAR"
   SQLType="VARCHAR"
   TableName="ADDRESSES">
   RetrievedOnUpdate="true"
   RetrievedOnInsert="true">
   <DesignTime>
         <Attr Name="_DisplaySize" Value="2"/>
   </DesignTime>
   <ListValidationBean
         xmlns="http://xmlns.oracle.com/adfm/validation"
         Name="CountryId_Rule_1"
         ResId="CountryId_Rule_0"
         OnAttribute="CountryId"
         OperandType="VO_USAGE"
         Inverse="false"
         ViewAccAttrName="Value"
         ViewAccName="SharedCountriesVA">
         <ResExpressions>
            <Expression
               Name="0"><![CDATA[SharedCountriesVA.Value]]>
            </Expression>
         </ResExpressions>
   </ListValidationBean>
   <Properties>
      <SchemaBasedProperties>
        <LABEL
          ResId="CountryId_LABEL"/>
      </SchemaBasedProperties>
   </Properties>
</Attribute>
```

What You May Need to Know About Dynamic Filtering with View Accessors

The View Object API setWhereClause() method allows you to add a dynamic WHERE clause to a view instance whose view accessor may already have view criteria specified at design time. At runtime, when your view accessor and its view criteria is applied to the view instance and you call the setWhereClause() method on the view instance to add an extra WHERE clause, the programmatically set WHERE clause is AND-ed with the WHERE clause of any already applied view criteria.

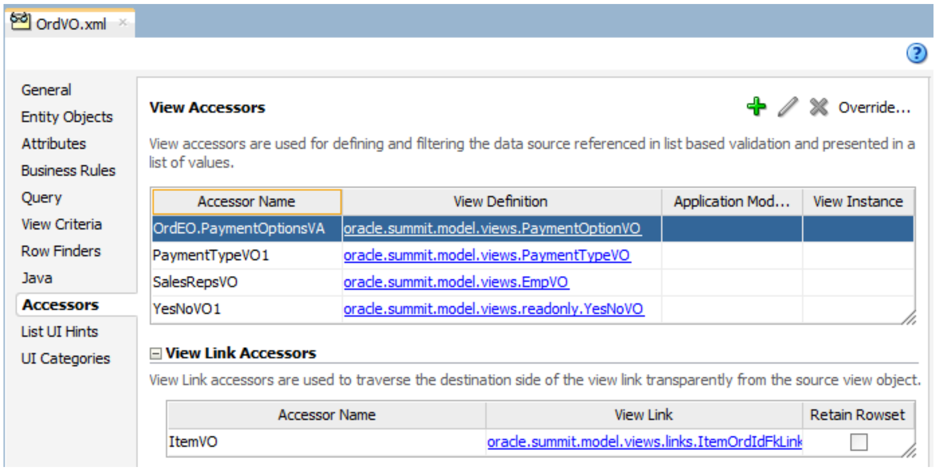
How to Create an LOV Based on a Lookup Table

View accessors that you create to access the view rows of a destination view object may be used to display a list of values to the end user at runtime. You first create a view accessor with the desired view instance as its data source, and then you can add the view accessor to an LOV-enabled attribute of the displaying view object. You will edit the view accessor definition for the LOV-enabled attribute so that it points to the specific lookup attribute of the view instance. Because you want to populate the row set cache for the query with static data, you would locate the destination view instance in a shared application module.

While the list usage is defined on the attribute of a view object bound to a UI list control, the view accessor definition exists on either the view object or the view object's base entity object. If you choose to create the view accessor on the view object's entity object, the Accessors page of the overview editor for the view object will display the inherited view accessor, as shown in Figure 14-7. Alternatively, if you choose to create the view accessor on the attribute's view object, you can accomplish this from either the editor for the LOV definition or from the Accessors page of the overview editor.

To ensure that the view accessor for the LOV always queries the latest data from the database table, you can enable the auto-refresh feature on the view accessor's destination view object.

Figure 14-7 Accessors Page of the Overview Editor



View Accessors

View accessors are used for defining and filtering the data source referenced in list based validation and presented in a list of values.

Accessor Name	View Definition	Application Mod...	View Instance
OrdEO.PaymentOptionsVA	oracle.summit.model.views.PaymentOptionVO		
PaymentTypeVO1	oracle.summit.model.views.PaymentTypeVO		
SalesRepsVO	oracle.summit.model.views.EmpVO		
YesNoVO1	oracle.summit.model.views.readonly.YesNoVO		

⊟ View Link Accessors

View Link accessors are used to traverse the destination side of the view link transparently from the source view object.

Accessor Name	View Link	Retain Rowset
ItemVO	oracle.summit.model.views.links.ItemOrdIdFkLink	□

Before you begin:

It may be helpful to have an understanding of view accessors. For more information, see Accessing View Instances of the Shared Service.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Shared Application Modules.

You may also find it helpful to understand additional examples of how to work with LOV-enabled attributes. For more information, see Working with List of Values (LOV) in View Object Attributes.

You will need to complete these tasks:

- Create the view object that is the data source for the view accessor, as described in How to Create an Entity-Based View Object, and How to Create a Custom SQL Mode View Object.
- Create the view object that is the data source for the view accessor, as described in How to Create an Entity-Based View Object, and How to Create a Custom SQL Mode View Object.
- Optionally, enable auto refresh for the view object, as described in How to Automatically Refresh the View Object of the View Accessor.

  If you need to ensure that your view accessor always queries the latest data from the lookup table, you can set the **Auto Refresh** property on the destination view object. This property allows the view object instance to refresh itself after a change in the database. The typical use case is when you define a view accessor for the destination view object.

To create an LOV that displays values from a lookup table:

1. In the Application window, double-click the view object that contains the desired attribute.
2. In the overview editor, click the **Accessors** navigation tab.
3. In the Accessors page, in the View Accessors list, check to see whether the view object inherited the desired view accessor from its base entity object. If no view

accessor is present, either create the view accessor on the desired entity object or click the **Create New View Accessors** button to add the accessor to the view object you are currently editing.

Validation rules that you define are always defined on the attributes of the view object's base entity object. It may therefore be convenient to define view accessors at the level of the base entity objects when you know that you will also validate entity object attributes using a view accessor list.

4. In the overview editor, click the **Attributes** navigation tab.
5. In the Attributes page, select the attribute that is to display the LOV, and then click the **List of Values** tab and click the **Add List of Values** button.
6. In the Create List of Values dialog, select the view accessor from the **List Data Source** dropdown list.

   The view accessor you select, will be the one created for the lookup table view object instances to use as the data source.
7. Select the attribute from this view accessor from the **List Attribute** dropdown list that will return the list of values for the attribute you are currently editing.

   The editor creates a default mapping between the view object attribute and the LOV-enabled attribute. In this use case, the attributes are the same. For example, the attribute `OrderId` from the `OrdersView` view object would map to the attribute `OrderId` from the `Shared_OrdersVA` view accessor.
8. If you want to specify supplemental values that your list returns to the base view object, click the **Add** button in **List Return Values** and map the desired view object attributes to the same attributes accessed by the view accessor.

   Supplemental attribute return values are useful when you do not require the user to make a list selection for the attributes, yet you want those attributes values, as determined by the current row, to participate in the update.

   For example, to map the attribute `StartDate` from the `OrdersView` view object, you would choose the attribute `StartDate` from the `Shared_OrdersVA` view accessor. Do not remove the default attribute mapping for the attribute for which the list is defined.
9. Click **OK**.

What Happens When You Define an LOV for a View Object Attribute

When you add an LOV to a view object attribute, JDeveloper updates the view object's XML file with an `LOVName` property in the `<ViewAttribute>` element. The definition of the LOV appears in a new `<ListBinding>` element. The metadata in the following example shows that the `PaymentTypeId` attribute refers to the `LOV_PaymentTypeId` LOV and sets the `choice` control type to display the LOV. The LOV definition for `LOV_PaymentTypeId` appears in the `<ListBinding>` element.

```
<ViewObject
  xmlns="http://xmlns.oracle.com/bc4j"
  Name="CustomerRegistrationVO"
  ...
   <ViewAttribute Name="PaymentTypeId"
                  LOVName="LOV_PaymentTypeId"
                  PrecisionRule="true"
                  EntityAttrName="PaymentTypeId"
                  EntityUsage="OrdEO"
```

```
                    AliasName="PAYMENT_TYPE_ID">
      <Properties>
         <SchemaBasedProperties>
            <CONTROLTYPE Value="choice"/>
         </SchemaBasedProperties>
      </Properties>
   </ViewAttribute>
  ...
  <ListBinding
    Name="LOV_PaymentTypeId"
    ListVOName="PaymentTypeVA"
    ListRangeSize="-1"
    NullValueFlag="start"
    NullValueId="LOVUIHints_NullValueId"
    MRUCount="0">
    <AttrArray Name="AttrNames">
      <Item Value="PaymentTypeId"/>
    </AttrArray>
    <AttrArray Name="ListAttrNames">
      <Item Value="Id"/>
    </AttrArray>
    <AttrArray Name="ListDisplayAttrNames">
      <Item Value="Payment Type"/>
    </AttrArray>
    <DisplayCriteria/>
  </ListBinding>
  ...
</ViewObject>
```

What Happens at Runtime: How the Attribute Displays the List of Values

The ADF Business Components runtime adds functionality in the attribute setters of the view row and entity object to facilitate the LOV-enabled attribute behavior. In order to display the LOV-enabled attribute values in the user interface, the LOV facility fetches the data source, and finds the relevant row attributes and mapped target attributes.

What You May Need to Know About Displaying List of Values From a Lookup Table

Unlike entity-based view objects, read-only, custom SQL view objects that you create, will not define a key attribute by default. While it is possible to create a read-only view object without defining its key attribute, in custom SQL mode it is a best practice to select the attribute that corresponds to the queried table's primary key and mark it as the key attribute. The presence of a key attribute ensure the correct runtime behavior for row set navigation. For example, the user interface developer may create an LOV component based on the read-only view object collection. Without a key attribute to specify the row key value, the LOV may not behave properly and a runtime error can result.

What You May Need to Know About Programmatically Invoking Database Change Notifications

When you create a databound UI component in a web page, you can enable the auto-refresh feature on the corresponding view object, as described in How to Automatically Refresh the View Object of the View Accessor. In this case, the **ADF Model** layer and ADF Business Components will handle processing database change notifications at runtime and the shared application module cache will be updated for you. However, when you create a method to programmatically refresh a view instance of a shared application module, your method needs to invoke the `processChangeNotification()` method on the shared application module before you refresh the view instance. Calling the `processChangeNotification()` method before refreshing the view instance ensures that the shared application module cache gets updated if the corresponding queried data has changed in the database.

To programmatically refresh a view instance of a shared application module, follow these steps (as illustrated in the following example from the `processChangeTestClient.java` example in the `SummitADF_Examples` workspace):

1. Call the `processChangeNotification()` method.
2. Get the view instance from the shared application module.
3. Refresh the view instance.

```
public class processChangeTestClient {
    public static void main(String[] args) {
        processChangeTestClient processChangeTestClient = new
processChangeTestClient();

        ApplicationModuleHandle handle = null;
        String amDef = "oracle.summit.model.services.BackOfficeAppModule";
        String config = "BackOfficeAppModuleLocal";
        try {
            handle = Configuration.createRootApplicationModuleHandle(amDef,
config);
            ApplicationModule am = handle.useApplicationModule();
            // 1. Update the shared application module cache with changed data.
            am.processChangeNotifications();
            // 2. Get the view instance to refresh.
            ViewObject vo = am.findViewObject("Inventory");
            // 3. Refresh the view instance with updated data.
            ((ViewObjectImpl)vo).refreshCollection(null, false, false);
            vo.reset();
            while (vo.hasNext()) {
                Row r = vo.next();
                System.out.println((String)r.getAttribute("Name"));
            }
        } finally {
            if (handle != null)
              Configuration.releaseRootApplicationModuleHandle(handle, false);
        }

    }
}
```

What You May Need to Know About Inheritance of AttributeDef Properties

When one view object extends another, you can create the LOV-enabled attribute on the base object. Then when you define the child view object in the overview editor, the LOV definition will be visible on the corresponding view object attribute. This inheritance mechanism allows you to define an LOV-enabled attribute once and apply it later across multiple view objects instances for the same attribute. For details about extending a view object from another view object definition, see How To Extend a Component After Creation.

You can also use the overview editor to extend the inherited LOV definition. For example, you may add extra attributes already defined by the base view object's query to display in selection list. Alternatively, you can create a view object instance that uses a custom WHERE clause to query the supplemental attributes not already queried by the base view object. For information about customizing entity-based view objects, see Working with Bind Variables.

What You May Need to Know About Using Validators

If you have created an LOV-enabled attribute for a view object, there is no need to validate the attribute using a List validator. You use an attribute validator only when you do not want the list to display in the user interface but still need to restrict the list of valid values. A List validator may be a simple static list or it may be a list of possible values obtained through a view accessor you define. Alternatively, you might prefer to use a Key Exists validator when the attribute displayed in the UI is one that references a key value (such as a primary, foreign, or alternate key). For information about declarative validation in ADF Business Components, see Defining Validation and Business Rules Declaratively.

Testing View Object Instances in a Shared Application Module

Use the interactive ADF Business Components application module testing tool in JDeveloper to test the data model. It helps you conducts tests without the need of an application user interface or test client program.

JDeveloper includes an interactive application module testing tool that you can use to test all aspects of its data model without having to use your application user interface or write a test client program. Running the Oracle ADF Model Tester can often be the quickest way of exercising the data functionality of your business service during development.

How to Test the Base View Object Using the Oracle ADF Model Tester

The application module is the transactional component that the Oracle ADF Model Tester (or UI client) will use to work with application data. The set of view objects used by an application module defines its data model, in other words, the set of data that a client can display and manipulate through a user interface. You can use the Oracle ADF Model Tester to test that the accessors you defined yield the expected validation result and that they display the correct LOV attribute values.

To test the view objects you added to an application module, use the Oracle ADF Model Tester, which is accessible from the Application window.

Before you begin:

It may be helpful to have an understanding of Oracle ADF Model Tester. For more information, see Testing View Object Instances in a Shared Application Module.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Shared Application Modules.

You may also find it helpful to understand the diagnostic messages specific to ADF Business Components debugging. For more information, see How to Enable ADF Business Components Debug Diagnostics.

You will need to complete this task:

Create the application module with view instances, as described in Creating and Modifying an Application Module.

To test view objects in an application module configuration:

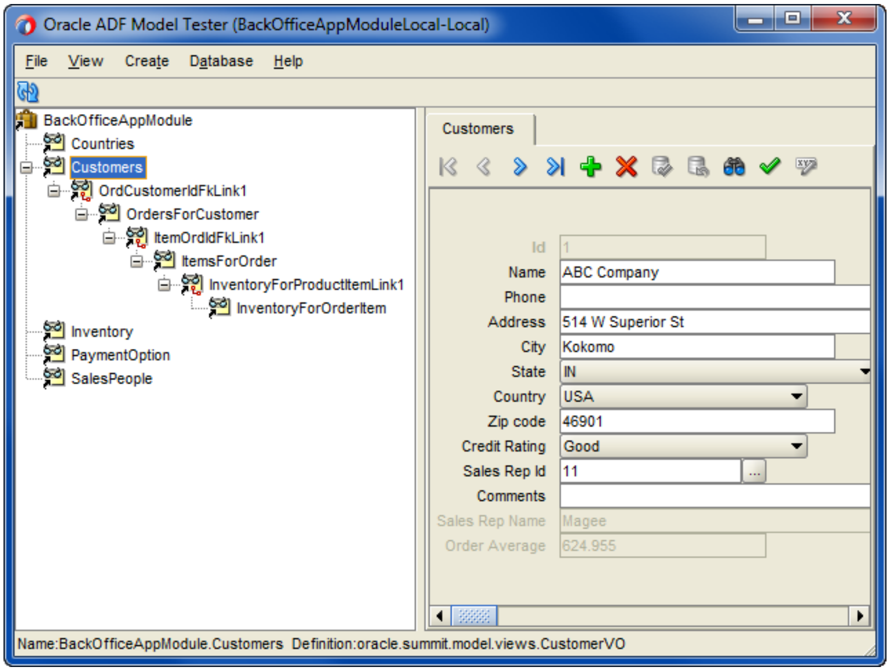
1. In the Application window, right-click the application module and choose **Run**.

   Alternatively, choose **Debug** when you want to run the application in the Oracle ADF Model Tester with debugging enabled. For example, when debugging using the Oracle ADF Model Tester, you can view status message and exceptions, step in and out of source code, and manage breakpoints. The debugger process panel opens in the Log window and the various debugger windows.

2. In the Oracle ADF Model Tester, expand the tree list and double-click the desired view object node.

   Note that the view object instance may already appear executed in the testing session. In this case, the tester panel on the right already displays query results for the view object instance, as shown in Figure 14-8. The fields in the tester panel of a read-only view object will always appear disabled since the data it represents is not editable.

Figure 14-8 Testing the Data Model in the Oracle ADF Model Tester



Id: 1
Name: ABC Company
Phone:
Address: 514 W Superior St
City: Kokomo
State: IN
Country: USA
Zip code: 46901
Credit Rating: Good
Sales Rep Id: 11
Comments:
Sales Rep Name: Magee
Order Average: 624.955

Name:BackOfficeAppModule.Customers Definition:oracle.summit.model.views.CustomerVO

How to Test LOV-Enabled Attributes Using the Oracle ADF Model Tester

To test the LOV you created for a view object attribute, use the Oracle ADF Model Tester, which is accessible from the Application window. For details about displaying the tester and the supported control types, see How to Test LOV-Enabled Attributes Using the Oracle ADF Model Tester.

What Happens When You Use the Oracle ADF Model Tester

When you launch the Oracle ADF Model Tester, JDeveloper starts the tester tool in a separate process and the Oracle ADF Model Tester appears. The tree at the left of the dialog displays all of the view object instances in your application module's data model. Figure 14-8 shows just one instance in the expanded tree, called `ProductImages`. After you double-click the desired view object instance, the Oracle ADF Model Tester will display a panel to inspect the query results, as shown in Figure 14-8.

The test panel will appear disabled for any read-only view objects you display because the data is not editable. But even for the read-only view objects, the tool affords some useful features:

- You can validate that the UI hints based on the Label Text **control hint** and format masks are defined correctly.
- You can also scroll through the data using the toolbar buttons.

The Oracle ADF Model Tester becomes even more useful when you create entity-based view objects that allow you to simulate inserting, updating, and deleting rows, as described in How to Test Entity-Based View Objects Interactively.

What Happens at Runtime: How Another Service Accesses the Shared Application Module Cache

When a shared application module with application scope is requested by an LOV, then the ADF Business Components runtime will create an `ApplicationPool` object for that usage. There is only one `ApplicationPool` created for each shared usage that has been defined in the ADF Business Components project configuration file (`.jpx`). The runtime will then use that `ApplicationPool` to acquire an application module instance that will be used like a user application module instance, to acquire data. The reference to the shared application module instance will be released once the application-scoped application module is reset. The module reference is released whenever you perform an unmanaged release or upon session timeout.

Since multiple threads will be accessing the data caches of the shared application module, it is necessary to partition the iterator space to prevent race conditions between the iterators of different sessions. This will help ensure that the next request from one session does not change the state of the iterator that is being used by another session. The runtime uses ADF Business Components support for multiple iterators on top of a single `RowSet` to prevent these race conditions. So, the runtime will instantiate as many iterators as there are active sessions for each `RowSet`.

An application-scoped shared application module lifecycle is similar to the lifecycle of any application module that is managed by the `ApplicationPool` object. For example, once all active sessions have released their shared application module, then the application module may be garbage-collected by the `ApplicationPool` object. The shared pool may be tuned to meet specific application requirements.

Session-scoped shared application modules are simply created as nested application module instances within the data model of the root, user application module. For details about nested application modules, Defining Nested Application Modules.

15

Creating SOAP Web Services with Application Modules

This chapter describes how to publish ADF application modules and how to define a SOAP service interface connection to make them available as external web services in a Fusion web application. It also describes how to incorporate the published application module as an external service in a Fusion web application.
This chapter includes the following sections:

About Service-Enabled Application Modules

Service-enabled ADF Business Components application modules are made available to SOAP service customers through a service interface.

A **service-enabled application module** is an ADF application module that you advertise through a service interface to service consumers. There are three scenarios for service consumers to consume a published service-enabled application module: web service access, Service Component Architecture (SCA) composite access, and access by another ADF application module.

Service Component Architecture (SCA) provides an open, technology-neutral model for implementing remotable services that are defined in terms of business functionality and that make middleware functions more accessible to application developers. **ADF Business Components** supports an SCA-compliant solution through application modules you can publish with a service interface.

When you service-enable your application module, JDeveloper generates the necessary artifacts comprising: 1) The Java interface defining the service, 2) an EJB 3.0 session bean that implements this Java interface, 3) a WSDL file that describes the service's operations, and (4) XML schema documents (XSD) that defines the service's data structures. The service interface is described for Fusion web application clients in a language-neutral way by the combination of WSDL and XSD.

SCA defines two kinds of service:

- Remotable services, typically coarse-grained and designed to be published remotely in a loosely coupled SOA architecture
- Local services, typically fine-grained and designed to be used locally by other implementations that are deployed concurrently in a tightly coupled architecture

ADF Business Components services fall into the first category, and should only be used as remotable services.

ADF Business Components services, including data access and method calls, defined by the remote application modules are interoperable with any other application module. This means the same application module can support interactive web user interfaces using ADF **data controls** and web service clients.

Service-Enabled Application Module Use Cases and Examples

Any development team can publish a service-enabled application module to contribute to the Fusion web application. The Fusion web application assembled from remote web services also does not require the participating services to run on a single application server.

Although the web applications may run on separate application servers, the appearance that SCA provides is one of a unified application. Consuming client projects use the ADF service factory lookup mechanism to access the data and any business methods encapsulated by the service-enabled application module. At runtime, the calling client and the ADF web service may or may not participate in the same transaction, depending on the protocol used to invoke the service (either SOAP or RMI). Only the RMI protocol and a Java Transaction API (JTA) managed transaction support the option to call the service in the same transaction as the calling client. By default, to support the RMI protocol, the ADF web service is configured to participate in the same transaction.

Additional Functionality for Service-Enabled Application Modules

You may find it helpful to understand other **Oracle ADF** features before you start working with ADF Business Components services. Following are links to other functionality that may be of interest.

- For information about the SCA and service data object (SDO) standards, see the Specifications section of the website for the Organization for the Advancement of Structured Information Standards (OASIS) through the Open Composite Services Architecture (CSA) at `http://www.oasis-opencsa.org`.
- For further background about web services and Oracle WebLogic Server support for web services, see Overview of Web Services in *Understanding Web Services*
- For details about the predefined authorization policies supported by Oracle Web Services Manager (OWSM), including `binding_permission_authorization_policy` used to enable authorization for RMI clients. For more information, see the "Predefined Policies" appendix in *Securing Web Services and Managing Policies with Oracle Web Services Manager*.
- For local service support, use the `ApplicationModule` interface and `ViewObject` interface support described in Working Programmatically with an Application Module's Client Interface.
- For API documentation related to the `oracle.jbo` package, see the following Javadoc reference document:
  - *Java API Reference for Oracle ADF Model*

Publishing Service-Enabled Application Modules

Use the ADF Application Module to enable a SOAP-based web service and make available rows of view object data.

The application module is ADF Business Components framework component that encapsulates business logic as a set of related business functions. Application modules are mapped to services. You use the overview editor for your application module to enable a web service interface and publish rows of **view object** data as service data object (SDO) components.

The SDO framework upon which the SDO components are based abstracts the data of the view object and standardizes the way that data structures are passed between Java and XML. This data abstraction simplifies working with heterogeneous data sources in a service-oriented architecture (SOA) and lets you selectively service-enable view objects using the same view object to support interactive web user interfaces and web service clients.

JDeveloper allows you to expose application modules as web services which use SDO components based on view instance that your application module defines to standardize the way that data structures are passed between Java and XML. JDeveloper also generates the WSDL service description that is used by the web service client in the consuming application.

> **Note:**
>
> JDeveloper only supports generating SDO components for view objects of a service-enabled application module. Currently, no other ADF Business Components may be defined as SDO components.

The service-enabled application module exposes the view objects, custom methods, built-in data manipulation operations, and specialized find methods based on named **view criteria** to be used by the client. Once you have enabled the application module service interface, you will need to create an ADF Business Components Service Interface deployment profile and deploy it to the target application server.

> **Note:**
>
> It is important to note that you don't implement methods with SDO parameters directly. The SDO framework is used to wrap the view row types during runtime only.

How to Enable the Application Module Service Interface

You edit the application module in JDeveloper to create a web service interface that exposes the top-level view objects and defines the available service operations it supports. Top-level view objects that you may service-enable and make accessible by the service client are:

- View instances of standalone view objects (does not participate in a hierarchical relationship)
- Master view instances of master-detail related view objects (does not include the child view instance)

- View instances of view objects that extend a parent view object (must not include a subtype discriminator attribute to define a polymorphic view object)

View objects that do not automatically participate in the service interface are:

- Child view instances of master-detail related view objects

  You can add child view objects individually to the service interface. For information on how to create SDO classes for child view objects, see How to Service-Enable Individual View Objects.

- View instances of polymorphic view objects (as defined by a subtype discriminator attribute)

  In JDeveloper, you can create a web service proxy client to access and manipulate for polymorphic view objects. For an example of a JAX-WS client, see Accessing Polymorphic Collections in the Consuming Application.

The primary purpose of the standard service operations is to expose data manipulation operations on the view objects. Any business logic that you have defined on the underlying framework objects (for example, business rule validation) will be applied when you invoke a standard service operation.

Table 15-1 shows the list of standard operations that service view instances support.

Table 15-1 Standard View Instance Data Manipulation Operations

Operation	Method Name	Operation Description
`Create`	`create<`*`VOName`*`>`	Creates an ADF Business Components view row.
`Update`	`update<`*`VOName`*`>`	Updates an ADF Business Components view row.
`Delete`	`delete<`*`VOName`*`>`	Deletes an ADF Business Components view row.
`Merge`	`merge<`*`VOName`*`>`	Updates an ADF Business Components view row if one exists; otherwise, creates a new one.
`GetByKey`	`get<`*`VOName`*`>`	Gets a single ADF Business Components view row by primary key.
`Find` (by view criteria applied to view object query statement)	`find<`*`VOName`*`>`	Finds and returns a list of ADF Business Components view rows using an SDO-based view criteria that is applied to the selected view object's query statement. Note that the query must not specify a bind variable defined as required for the query to execute. The service interface does not expose required bind variables at runtime. For details about creating a find method for this scenario, see How to Expose a Declarative Find Operation Filtered By a Required Bind Variable.

Table 15-1 (Cont.) Standard View Instance Data Manipulation Operations

Operation	Method Name	Operation Description
`Find` (by view criteria)	`find<VOName><VCName>`	Finds and returns a list of single ADF Business Components view rows by named view criteria and values for the required bind variables. This is the preferred way to filter the ADF Business Components view rows that rely on a required bind variable.
`Process`	`process<VOName>`	Performs a Create, Update, Delete, or Merge operation on a list of ADF Business Components view rows. The specified operation is applied to all objects in the given list.
`ProcessChangeSummary`	`processCS<VOName>`	Performs a Create, Update, or Delete operation on a list of ADF Business Components view rows. Different operations may be applied to different objects, depending on what is specified in the `ChangeSummary` object.

Additionally, several built-in methods can be optionally added to the client service interface of the application module (`AppModuleService.java`). Table 15-2 shows the list of built-in methods that provide runtime support for service view instances that you enable.

Table 15-2 Built-In Service Interface Methods

Operation	Method Name and Async Method Name	Method Description

`Generate Object and Attribute Control Hint Operations`	`getDfltObjAttrHints()`
	`getDfltObjAttrHintsAsync()`
		Returns the base UI hints, including object and attribute display name labels, for an object based on a specific locale. The method takes the name of the object, at the service, to get the hints and the locale name (in ISO 639-1 format) to drive localization of base hints.
		This method may be used by the web service consuming application to generate a user interface with the correct labels for the service view instance and its attributes. For details about specifying display name UI hints, see How to Set Display Names for Service View Instances and Attributes.

Table 15-2 (Cont.) Built-In Service Interface Methods

Operation	Method Name and Async Method Name	Method Description

`Generate Last Update Time Operations`	`getServiceLastUpdateTime()`
	`getServiceLastUpdateTimeAsyncResponse()`
		Returns the last modified time for the service schema files at the SOAP service endpoint.
		This method may be used by the web service consuming application that caches the service view instance schema files (XSD) to determine when the object has been updated.

`Generate Entity List Operation`	`getEntityList()`
	`getEntityListAsync()`
		Returns the list of service view instance names and flags that indicate whether each service view object at the SOAP service endpoint supports Create, Update, Merge, and Delete operations.
		This method may be used by the web service consuming application that uses custom object services to determine the available operations. Custom object services are not created using the ADF Business Components web service design time and are not described in this documentation.

Before you begin:

It may be helpful to have an understanding of how the SDO framework supports service-enabled ADF application modules and enables web service clients to access rows of data and perform service operations. For more information, see Publishing Service-Enabled Application Modules.

When you create the service-enabled application module, do not create it in a project that already contains a standard web service (a Java class or interface with the `@WebService` annotation). JDeveloper deployment profiles do not support deploying a standard web service and an ADF Business Service web service from the same project. If you attempt to deploy the ADF Business Service web service from the same project as a standard web service, the deployment will fail with an Oracle WebLogic Server exception error.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Service-Enabled Application Modules.

You will need to complete these tasks:

- Create the desired view objects, as described in How to Create an Entity-Based View Object and How to Create a Custom SQL Mode View Object.
- Create the desired application module, as described in How to Create an Application Module.
- Optionally, set JDeveloper preferences to specify a default suffix for the names of generated SDO classes, modify the default subpackage where the service interface and classes reside, and set the default namespace prefix for the

generated SDO schema and web service, as described in How to Set Preferences for Generating the ADF Web Service Interface.

- If you want to expose custom find operations on the service, create declarative view criteria to specify the custom query, as described in How to Create Named View Criteria Declaratively.

To create the web service interface:

1. In the Applications window, double-click the application module.
2. In the overview editor, click the **Service Interface** navigation tab and then click the **Enable support for Service Interface** button.

   Use the Create Service Interface wizard to configure the desired options.
3. In Create Service Interface wizard, on the Service Interface page, enter the name and target namespace for the web service.

   The target namespace is a URI for the service that you can assign to group similar services together by entering the same URI. The default target namespace is the package name of the application module combined with package names that you specify in the Preferences dialog.

   > **Tip:**
   >
   > You can define JDeveloper preferences that will shape the URI and allow you to define a unique target namespace for your service. For details about namespace URI shaping preferences, see How to Set Preferences for Generating the ADF Web Service Interface.
4. To expose the methods of the application module as asynchronous service methods and enable both synchronous and asynchronous operations on the web service, select **Generate Asynchronous Web Service Methods**.

   By default, the web service supports synchronous service methods. This forces the invoking client application to wait for the response to return before it can continue with its work. In cases where the response returns immediately, this method of invoking the web service is common. However, because request processing can be delayed, it is often useful for the client application to continue its work and to handle the response later on.
5. The **Service Categories** list will appear empty unless role shaping has been defined based on previously defined customization roles. Leave the Selected list empty unless your organization is classifying services.

   This feature has no impact on the service WSDL file and has no relevance to the operation of web services.
6. On the Service Built-in Methods page, optionally, select **Generate Object and Attribute Control Hints Operation** to generate a method that will return the static control hints (UI hints) defined on a view instance and its attributes.

   When you enable this option, the wizard adds the `getDfltObjAttrHints()` method to the client service interface (`AppModuleService.java`). The web service consuming application can invoke this method to resolve UI hints on the server without requiring a database roundtrip. The method takes a service view object name and a locale name and returns the base UI hints, including object and attribute display name labels, for an object based on a specific locale. If the locale name is not specified, the method uses the default locale.

7. Optionally, select **Generate Last Update Time Operation** to generate a method that will return the last modified time for the service schema files.

   When you enable this option, the wizard adds the `getServiceLastUpdateTime()` method to the client service interface (`AppModuleService.java`). Before any customization occurs, this method returns the file timestamp of the last updated schema file. When a schema is customized at runtime, the last-update timestamp is updated, and this latest timestamp will be returned. This feature is primarily useful in web service consuming applications that cache the service XSD.

8. Optionally, select **Generate Entity List Operation** to generate a method that will return the list of service view instances.

   When you enable this option, the wizard adds the `getEntityList()` method to the client service interface (`AppModuleService.java`). The method returns the list of service view instance names, XSD type, and flags that indicate whether the service view supports Create, Update, Merge, and Delete operations. This feature is primarily useful in web service consuming applications that require customization.

9. On the Service Custom Methods page, add the custom methods you want to expose in the service interface and define the data types of each method's parameters and return value.

   The parameters and non-void return value of the custom service methods you enable must be one of the service-interface supported data types, such as a Java primitive type, or a list of the service-interface supported data types (including `oracle.jbo.server.ViewRowImpl`, `java.util.List<`*`ViewRowImpl`*`>`, `oracle.jbo.AttributeList`, `java.util.List<`*`AttributeList`*`>`, or `java.util.List<`*`PrimitiveType`*`>`).

   Note that although both `ViewRowImpl` and `AttributeList` data types expose the identical row structure to the web service client, at runtime there will be a fundamental difference. For a description of the supported data types, see What You May Need to Know About Method Signatures on the ADF Web Service Interface.

   After selecting a qualifying custom method to appear in the service interface, for each parameter and return value using the `ViewRowImpl` or `AttributeList` data type, you must in turn select the name of the view object instance corresponding to the row structure:

   a. In the Selected list, expand **return** or **parameters** and select the item.

   b. Enter the Java element data type in **Element Java Type**.

   c. In the case where the Java element type is `ViewRowImpl` or `AttributeList`, enter the view object instance name to identify the row structure in **Element View Object**.

      For example, if you define a custom method to return a single row of the `CustomerInfo` view object instance, you would need a custom method signature like this:

      ```
      public ViewRowImpl findCustomerInfo(int id)
      ```

      Then, after selecting the `findCustomerInfo()` custom method to appear in the service interface, you would select its return value in the tree and configure its

View Object property to be `CustomerInfo`, the view instance name whose row structure should be used at runtime.

10. To expose service information messages for a custom method or warnings for process operation methods, select **Include Warnings in Return Service Data Object**.

    For example, you might want to display an informational message when a method returns the total employee compensation and the total is outside of the desired range.

    If **Include Warnings** is not selected, no informational messages will be returned with the service response.

    This option is only enabled when the method does not return a view row or a list of view rows. When the method returns view rows, the underlying view object determines whether the method supports warnings, as described How to Service-Enable Individual View Objects.

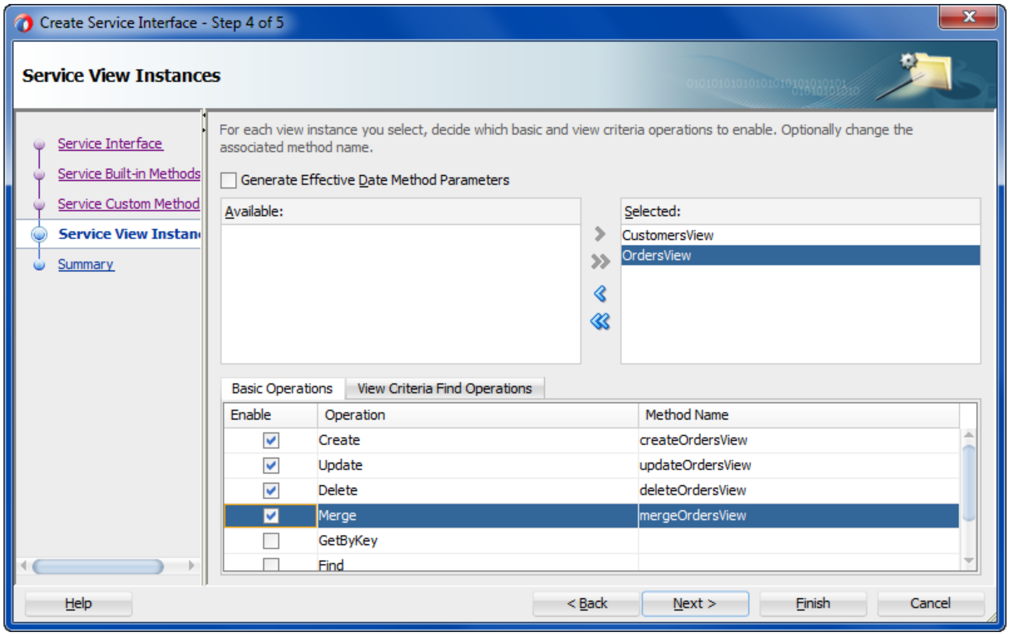
    The informational messages (and warnings) are reported as part of the return object. JDeveloper generates appropriate wrappers as the return objects, and the wrappers contain the actual method return and the informational messages.

11. On the Service View Instances page, select the top-level view instances in the application module that you want to expose in the service interface.

    View object subtypes of the top-level view instance will automatically be service-enabled.

    Also, on this page, you can enable the available data manipulation operations supported on the exposed methods, as shown in Figure 15-1.

Figure 15-1   View Instances and CRUD Operation Selection



Create Service Interface - Step 4 of 5

Service View Instances

For each view instance you select, decide which basic and view criteria operations to enable. Optionally change the associated method name.

☐ Generate Effective Date Method Parameters

Available:

Selected:
CustomersView
OrdersView

Enable	Operation	Method Name
☑	Create	createOrdersView
☑	Update	updateOrdersView
☑	Delete	deleteOrdersView
☑	Merge	mergeOrdersView
☐	GetByKey	
☐	Find	

12. In the Basic Operations tab, select the data manipulation operations for the currently selected view instance and in the **Method Name** field, change the names of the selected service operations to the names that you prefer to expose in the service interface.

The primary purpose of the standard service operations is to expose data manipulation operations on the view objects. Any business logic that you have defined on the underlying framework objects (for example, business rule validation) will be applied when you invoke the service operations. For a description of the operations that service view instances support, see Table 15-1.

In the case of the find method operation that you can select, the find method must not reference a required bind variable in the view object's query statement. A required bind variable is one that makes the query execution dependent on the availability of a valid value for the bind variable. The service interface does not expose required bind variables at runtime. For details about defining a find operation for this scenario, see How to Expose a Declarative Find Operation Filtered By a Required Bind Variable.

13. To expose declarative find operations, select the **View Criteria Find Operations** tab and click the **Add View Criteria** icon.

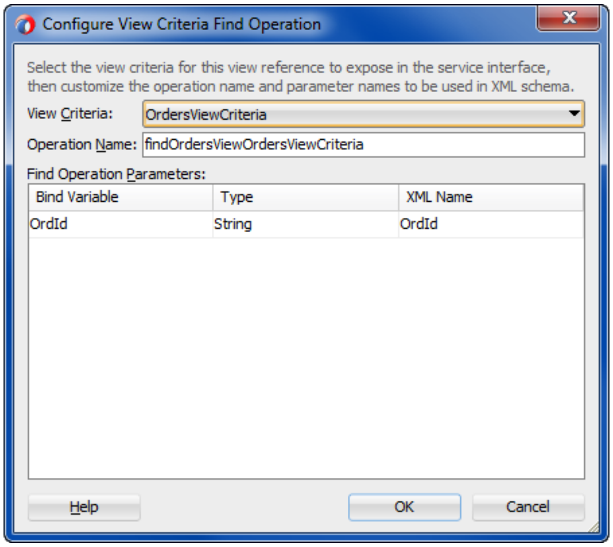
    You can define custom find operations when you want the service to support executing a predefined query. For information about defining a named view criteria, see Working with Named View Criteria.

    > **Caution:**
    >
    > The service interface find operations are based on specific view criteria that your project defines. This means that the bind variables of the view criteria must match the parameters of the corresponding find operation method. If you change the number or order of the bind variables after the find operation is defined and the service interface generated, the corresponding method will not execute at runtime. Therefore, after changing the underlying view criteria, you must regenerate the service interface.

    a. In the Configure View Criteria Find Operation dialog, choose the named view criteria for the find operation.

       The dialog displays the list of view criteria exposed by the referenced view object. For example, `OrdersView` defines `OrdersViewCriteria` with a bind variable `OrdId` that specifies the order ID, as shown in Figure 15-2.

Figure 15-2 Specialized Find Methods Based on Named View Criteria


Configure View Criteria Find Operation

Select the view criteria for this view reference to expose in the service interface, then customize the operation name and parameter names to be used in XML schema.

View Criteria: OrdersViewCriteria

Operation Name: findOrdersViewOrdersViewCriteria

Find Operation Parameters:

Bind Variable	Type	XML Name
OrdId	String	OrdId

b. If the view criteria uses a bind variable, you can double-click the XML name to customize the name as it will appear in the XML definition for the service.

14. Click **Next** to review the custom methods that your service view instances will expose.

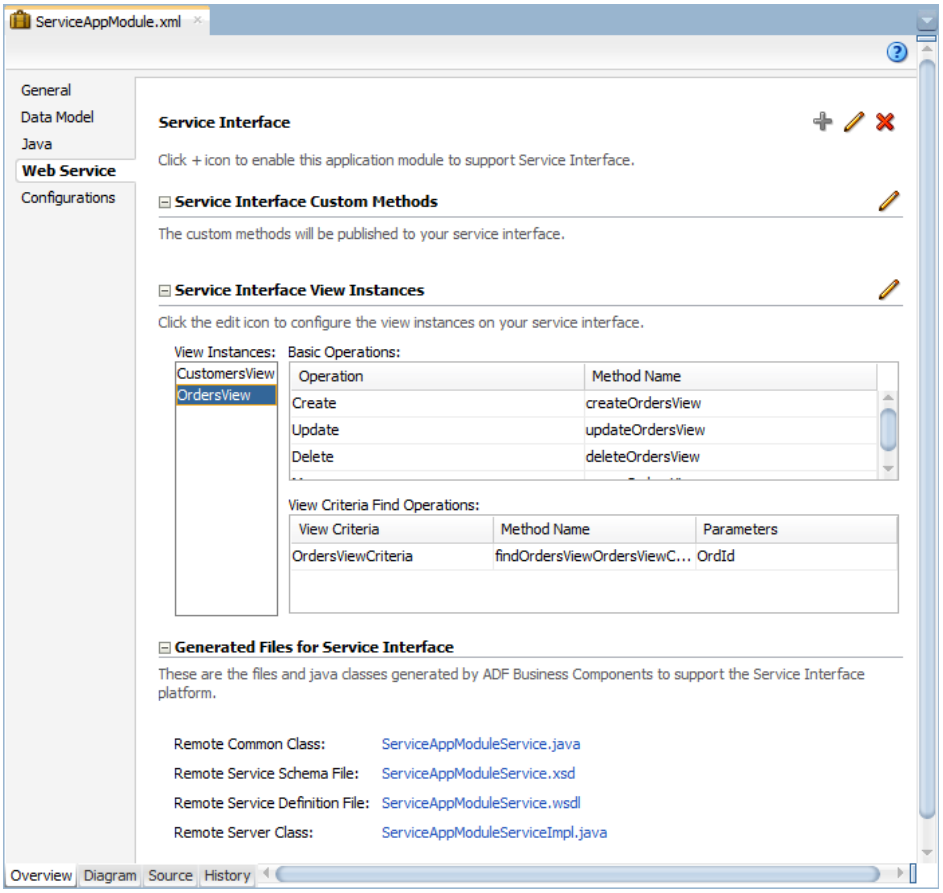
15. Click **Finish**.

What Happens When You Create an Application Module Service Interface

JDeveloper generates the service interface class and enables any view instance options you have chosen, as shown in Figure 15-3.

Do not modify the generated files for the service-enabled application module. The generated files implement required methods of the service interface. An exception to this are use cases that require adding Java annotations to the service implementation class. For example, annotations that you add in the service implementation class let you attach security policies, as described in How to Secure the ADF Web Service for Access By SOAP Clients.

Figure 15-3 Service Interface Page of the Overview Editor for an Application Module



Service Interface

Click + icon to enable this application module to support Service Interface.

Service Interface Custom Methods

The custom methods will be published to your service interface.

Service Interface View Instances

Click the edit icon to configure the view instances on your service interface.

View Instances:

- CustomersView
- OrdersView

Basic Operations:

Operation	Method Name
Create	createOrdersView
Update	updateOrdersView
Delete	deleteOrdersView

View Criteria Find Operations:

View Criteria	Method Name	Parameters
OrdersViewCriteria	findOrdersViewOrdersViewC...	OrdId

Generated Files for Service Interface

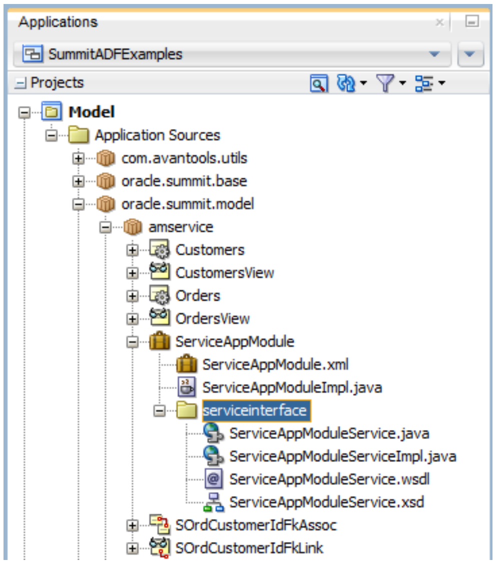
These are the files and java classes generated by ADF Business Components to support the Service Interface platform.

Remote Common Class:	ServiceAppModuleService.java
Remote Service Schema File:	ServiceAppModuleService.xsd
Remote Service Definition File:	ServiceAppModuleService.wsdl
Remote Server Class:	ServiceAppModuleServiceImpl.java

The following types of files are generated and are listed in the Applications window in the Projects panel, under the application module's **serviceinterface** node, as shown in Figure 15-4.

- Web service interface, for example, `ServiceAppModuleService.java`
- Web service schema file, for example, `ServiceAppModuleService.xsd`
- Web service definition file, for example, `ServiceAppModuleService.wsdl`
- Web service implementation class, for example, `ServiceAppModuleServiceImpl.java`

Figure 15-4 Service Interface Files Appear Below Application Module



In addition, the `connections.xml` file is created when you first create an ADF Business Components service. Although this file is only used by the web service client (the consuming application), it is generated with the service-enabled application module as a convenience. This file appears in the Applications window in the Application Resources panel, under the **Descriptors** and **ADF META-INF** nodes.

Annotations Generated in the Web Service Interface

The web service interface uses metadata annotations specified by the web service specification (JSR-181) to indicate how the interface should be exposed as a web service. This example shows part of `ServiceAppModuleService.java`, which is the web service interface class for the `ServiceAppModule` application module in the `oracle.summit.model.amservice` package in the `SummitADF_Examples` workspace.

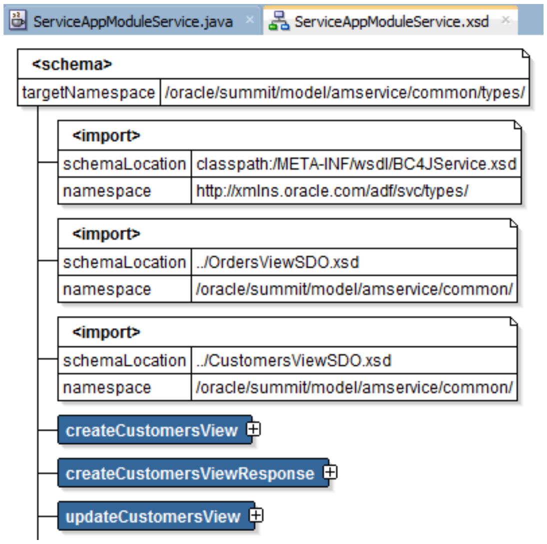
```
package oracle.summit.model.amservice.common.serviceinterface;
...
import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebResult;
import javax.jws.soap.SOAPBinding;
...
import oracle.summit.model.amservice.common.CustomerViewSDO;
import oracle.summit.model.amservice.common.OrderViewSDO;
...
import oracle.webservices.annotations.PortableWebService;
import oracle.webservices.annotations.SDODatabinding;
...
@SOAPBinding(parameterStyle = SOAPBinding.ParameterStyle.WRAPPED,
style=SOAPBinding.Style.DOCUMENT)
@PortableWebService(targetNamespace="http://www.globalcompany.com/
ServiceAppModuleService",
    name="ServiceAppModuleService",
    wsdlLocation=
            "oracle/summit/model/amservice/common/serviceinterface/
ServiceAppModuleService.wsdl")
```

```
@SDODatabinding(schemaLocation=
            "oracle/summit/model/amservice/common/serviceinterface/
ServiceAppModuleService.xsd")
public interface ServiceAppModuleService
{
    public static final String NAME =
("http://www.globalcompany.com/ServiceAppModuleService")

    @WebMethod(action="www.globalcompany.example.com/createCustomersView",
        operationName="createCustomersView")
    @RequestWrapper(targetNamespace="www.globalcompany.example.com/types/",
        localName="createCustomersView")
    @ResponseWrapper(targetNamespace="www.globalcompany.example.com/types/",
        localName="createCustomersViewResponse")
    @WebResult(name="result")
    CustomersViewSDO createCustomersView(@WebParam(mode =
WebParam.Mode.IN, name="customerView")
        CustomersViewSDO customersView) throws ServiceException;
    ...
}
```

Web Service Schema Generated in the Web Service Schema File

The web service schema file is an XML schema file which represents the web service schema, as shown in Figure 15-5.

Figure 15-5 Web Service Schema File


ServiceAppModuleService.java
ServiceAppModuleService.xsd

<schema>
targetNamespace /oracle/summit/model/amservice/common/types/

<import>
schemaLocation classpath:/META-INF/wsdl/BC4JService.xsd
namespace http://xmlns.oracle.com/adf/svc/types/

<import>
schemaLocation ../OrdersViewSDO.xsd
namespace /oracle/summit/model/amservice/common/

<import>
schemaLocation ../CustomersViewSDO.xsd
namespace /oracle/summit/model/amservice/common/

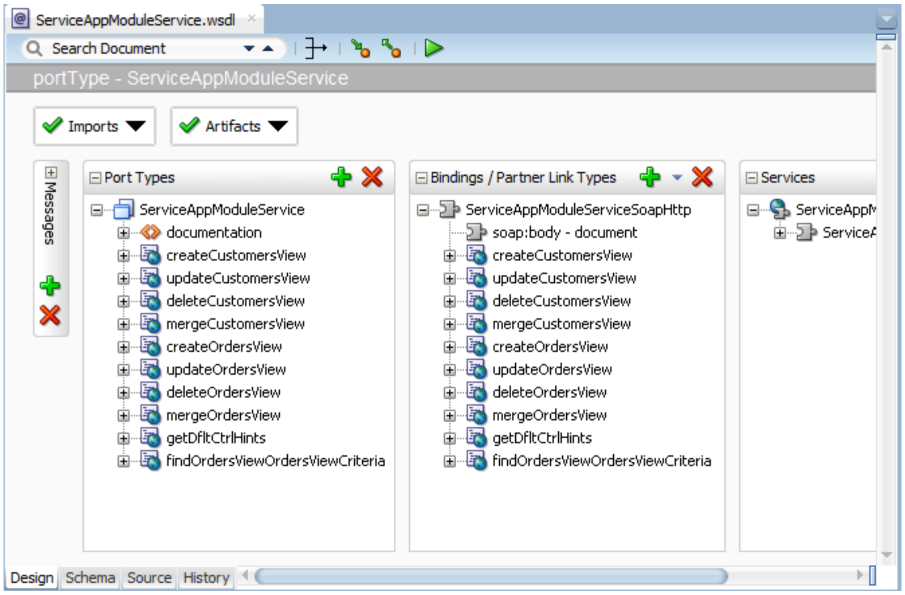
createCustomersView
createCustomersViewResponse
updateCustomersView

WSDL Generated in the Web Service Definition File

The web service definition file is a XML-structured document file that conforms to the Web Service Definition Language (WSDL) specification that describes the generated web service as a collection of endpoints, or ports. A port is defined by associating a network address with a reusable binding. The client application that connects to the web service reads the WSDL to determine what functions are available on the server. The WSDL also specifies the endpoint for the service itself, which you can use to locate and test your deployed service.

Figure 15-6 shows the WSDL for the web service generated for the `ServiceAppModule` application module in the WSDL visual editor. You can see the WSDL as an XML document by selecting the **Source** tab.

Figure 15-6 WSDL Document



Stateless Session Bean Specified by the Service Implementation Class

The service implementation class is an EJB 3.0 stateless session bean that implements the web service interface and extends the `oracle.jbo.server.svc.ServiceImpl` class, the generic service implementation for ADF Business Components. The following example shows part of `ServiceAppModuleServiceImpl.java`, which is the service implementation class for the `ServiceAppModule` application module in the `oracle.summit.model.amservice` package in the `SummitADF_Examples` workspace.

```
package oracle.summit.model.amservice.server.serviceinterface;
...
import oracle.summit.model.amservice.common.CustomersViewSDO;
import oracle.summit.model.amservice.common.OrdersViewSDO;
import
oracle.summit.model.amservice.common.serviceinterface.ServiceAppModuleService;

...
import oracle.webservices.annotations.PortableWebService;

@Interceptors({ ServiceContextInterceptor.class })
@Stateless(name="oracle.summit.model.amservice.common.ServiceAppModuleServiceBean
",
                    mappedName="ServiceAppModuleServiceBean")
@Remote(ServiceAppModuleService.class)
@PortableWebService(targetNamespace="http://www.globalcompany.com/
ServiceAppModuleService",
    serviceName="ServiceAppModuleService",
portName="ServiceAppModuleServiceSoapHttpPort",
    endpointInterface="oracle.summit.model.amservice.common.
```

```
                                     serviceinterface.ServiceAppModuleService")
public class ServiceAppModuleServiceImpl extends ServiceImpl implements
ServiceAppModuleService
{
    ...
    public CustomersViewSDO createCustomersView(CustomersViewSDO customersView)
        throws ServiceException {
        return (CustomersViewSDO) create(customersView, "CustomersView");
    }

    public CustomersViewSDO updateCustomersView(CustomersViewSDO customersView)
        throws ServiceException {
        return (CustomersViewSDO) update(customersView, "CustomersView");
    }
...
}
```

Lookup Defined in the connections.xml File

The connections.xml file allows the web service client to look up the service with the factory class oracle.jbo.client.svc.ServiceFactory. The ADF Business Components service factory provides the mechanism that allows the service client to look up the service. The service factory relies on ADF connection architecture and the connections.xml file to manage service endpoint locations. The connections.xml file is created when you first create an ADF Business Components service. This file appears in the Applications window in the Application Resources panel, under the **Descriptors** and **ADF META-INF** nodes.

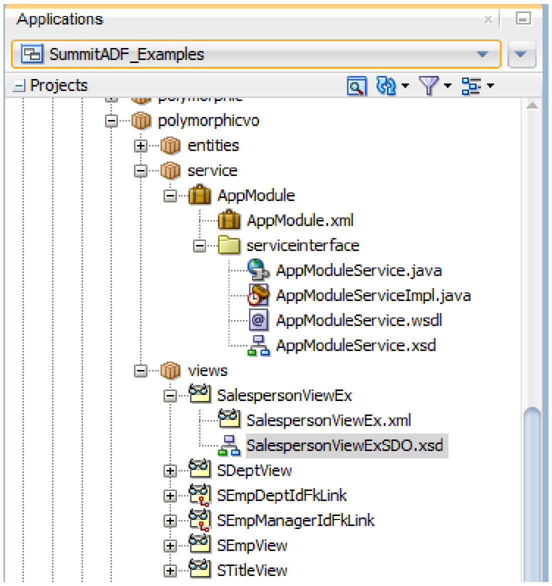
The following example shows the initial connections.xml entry created by JDeveloper when you first create an ADF Business Components service.

```
<Reference name="{http://www.globalcompany.com}ServiceAppModuleService"
                    className="oracle.jbo.client.svc.Service" xmlns="">
   <Factory className="oracle.jbo.client.svc.ServiceFactory"/>
   <RefAddresses>
      <StringRefAddr addrType="serviceInterfaceName">
         <Contents>oracle.summit.model.amservice.common.serviceinterface.
                          ServiceAppModuleService</Contents>
      </StringRefAddr>
      <StringRefAddr addrType="serviceEndpointProvider">
         <Contents>ADFBC</Contents>
      </StringRefAddr>
      <StringRefAddr addrType="jndiName">
         <Contents>ServiceAppModuleServiceBean#oracle.summit.model.amservice.
                       common.serviceinterface.ServiceAppModuleService</Contents>
      </StringRefAddr>
      <StringRefAddr addrType="serviceSchemaName">
         <Contents>ServiceAppModuleService.xsd</Contents>
      </StringRefAddr>
      <StringRefAddr addrType="serviceSchemaLocation">
         <Contents>oracle/summit/model/amservice/common/
                                        serviceinterface/</Contents>
      </StringRefAddr>
   </RefAddresses>
</Reference>
```

What Happens When You Create an Application Module Service Interface With Polymorphic View Objects

When your model project defines subtype view objects that extend a base view object, the generated SDO service interface will be strongly-typed for the base view objects, but not for the polymorphic subtypes. For example, assume `SalespersonViewEx` extends the `SEmpView` and adds an attribute `CommissionPct` to define the subtype. When you generate the service interface for the service-enabled view instances, the Applications window displays the service interface and service objects, as shown in Figure 15-7.

Figure 15-7 Service Object Generated for Polymorphic Subtype



> **Note:**
>
> After generating the SDO service interface from a ADF Model project that defines base view objects and subtype view objects, the generated service schema definition will be strongly-typed for the base view object, but not for the polymorphic subtype. Therefore, web service clients that need to access polymorphic subtype view objects use the approach described in Accessing Polymorphic Collections in the Consuming Application.

Because the polymorphic subtype is specified in application module data model as an import on the base service-enabled view instance, JDeveloper automatically generates the subtype service object from the base service object. The following example shows the `SalespsersonViewExSDO.xsd` subtype schema definition with the additional attribute `CommissionPct` defined. The subtype schema includes the attributes of base service object `SEmpViewSDO`.

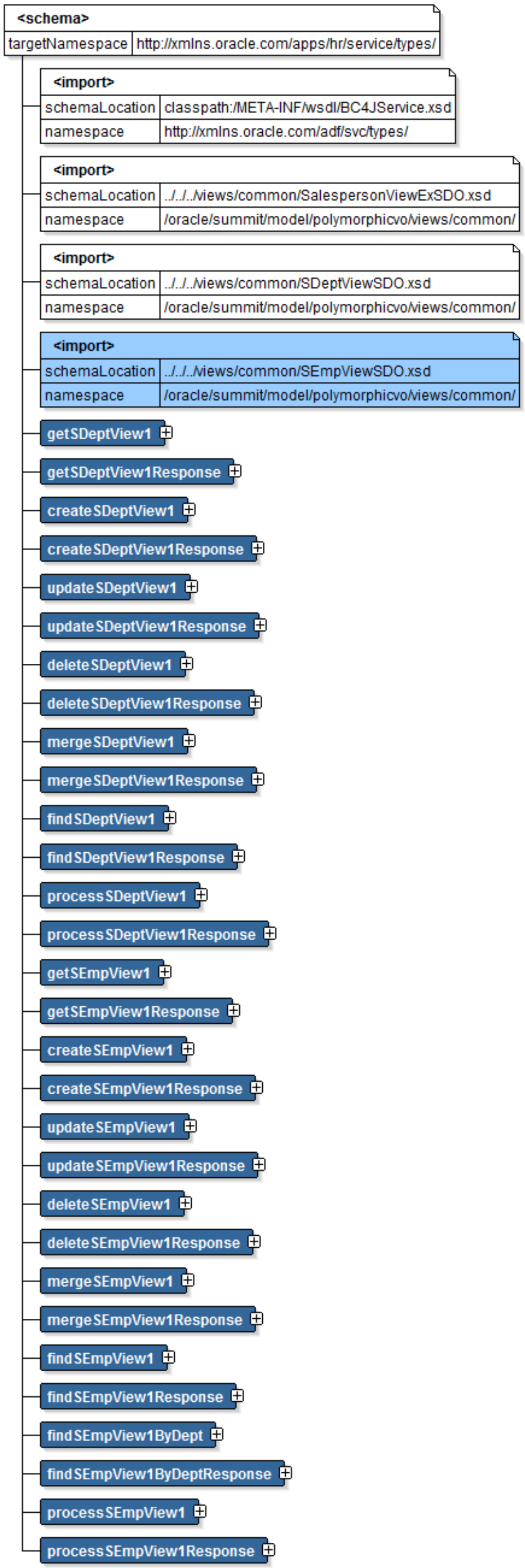
```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" targetNamespace="/oracle/model/
polymorphicvo/views/common/"
     xmlns="/oracle/model/polymorphicvo/views/common/" elementFormDefault="qualified">
    <xsd:include schemaLocation="SEmpViewSDO.xsd"/>
    <xsd:complexType name="SalespersonViewExSDO">
        <xsd:annotation>
            <xsd:appinfo source="http://xmlns.oracle.com/adf/svc/metadata/">
                <key xmlns="http://xmlns.oracle.com/adf/svc/metadata/">
                    <attribute>Id</attribute>
                </key>
            </xsd:appinfo>
        </xsd:annotation>
        <xsd:complexContent>
            <xsd:extension base="SEmpViewSDO">
                <xsd:sequence>
                    <xsd:element name="CommissionPct" type="xsd:decimal"
                                 minOccurs="0" nillable="true"/>
                </xsd:sequence>
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>
    <xsd:element name="salespersonViewExSDO" type="SalespersonViewExSDO"/>
</xsd:schema>
```

The generated service interface uses the XSD `<import>` element to expose the polymorphic subtype. The following example shows the service interface XSD `AppModuleService.xsd` with the import definition for the `SalespersonViewExSDO` subtype.

```
<schema xmlns="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
                    targetNamespace="http://xmlns.oracle.com/apps/hr/service/types/"
xmlns:tns="http://xmlns.oracle.com/apps/hr/service/types/"
     xmlns:ns0="/oracle/model/polymorphicvo/views/common/"     xmlns:ns1="http://
xmlns.oracle.com/adf/svc/types/">     <import namespace="http://xmlns.oracle.com/adf/svc/types/"
                    schemaLocation="classpath:/META-INF/wsdl/BC4JService.xsd"/>     <import
namespace="/oracle/model/polymorphicvo/views/common/"
                    schemaLocation="../../../../views/common/SalespersonViewExSDO.xsd"/>
     <import namespace="/oracle/model/polymorphicvo/views/common/"
                    schemaLocation="../../../../views/common/SDeptViewSDO.xsd"/>     <import
namespace="/oracle/model/polymorphicvo/views/common/"
                    schemaLocation="../../../../views/common/SEmpViewSDO.xsd"/>
```

Figure 15-8 shows how the web service schema represents the various strongly-typed service objects, as well as the import definition for the polymorphic subtype `SalespersonViewExSDO`.

Figure 15-8 Web Service Schema Imports Polymorphic View Usage



<schema>

targetNamespace	http://xmlns.oracle.com/apps/hr/service/types/

<import>

schemaLocation	classpath:/META-INF/wsdl/BC4JService.xsd
namespace	http://xmlns.oracle.com/adf/svc/types/

<import>

schemaLocation	../../../views/common/SalespersonViewExSDO.xsd
namespace	/oracle/summit/model/polymorphicvo/views/common/

<import>

schemaLocation	../../../views/common/SDeptViewSDO.xsd
namespace	/oracle/summit/model/polymorphicvo/views/common/

<import>

schemaLocation	../../../views/common/SEmpViewSDO.xsd
namespace	/oracle/summit/model/polymorphicvo/views/common/

What You May Need to Know About Method Signatures on the ADF Web Service Interface

You can define two different kinds of interfaces for an application module: the client interface and the service interface. The client interface is used by the **ADF Model** layer for UI clients. The service interface is for application integration and is used by external web services or other application services (either programmatically or automatically using the service-enabled entity feature).

An application module can support no interface at all, only client interfaces, only service interfaces, or both client interfaces and service interfaces combined. However, be aware that the two kinds of interfaces differ in the data types that are supported for the parameters and/or return values of your custom methods that you define for the respective interfaces. The types supported on the client interface are described in What You May Need to Know About Method Signatures on the Client Interface.

The service interface, in contrast to the client interface, supports a more narrow set of data types for custom method parameters and return values and is limited to:

- Java primitive types and their object wrapper types (for example, `int`, `Integer`, and `Long`)
- `java.lang.String`
- `java.math.BigDecimal`
- `java.math.BigInteger`
- `java.sql.Date`
- `java.sql.Time`
- `java.sql.Timestamp`
- `java.util.Date`
- `oracle.jbo.AttributeList`
- `oracle.jbo.domain.BlobDomain`
- `oracle.jbo.domain.Char`
- `oracle.jbo.domain.ClobDomain`
- `oracle.jbo.domain.DBSequence`
- `oracle.jbo.domain.Date`
- `oracle.jbo.domain.NClobDomain`
- `oracle.jbo.domain.Number`
- `oracle.jbo.domain.Timestamp`
- `oracle.jbo.domain.TimestampLTZ`
- `oracle.jbo.domain.TimestampTZ`
- `oracle.jbo.server.ViewRowImpl` or any subtype
- `java.util.List<`*aType*`>`, where *aType* is any of the service-interface supported data types, including Java primitive type

> **Note:** The service interface specifically does not support Java `Map` collection. This means it is not possible to return a collection of objects that are of different types. However, a collection is not limited to view row attributes, a return type can be defined as a list of any service-interface supported data type. For example, `List<DataObject>`, `List<AttributeList>`, and `List<String>` are all valid types.

You can define a custom method that returns a type of `AttributeList` when you want to allow the client developer to work with the list of attributes and perform custom operations without the need to involve framework behavior before the custom method executes. As an alternative, when the client developer wants the framework to manage rows (create, find, and populate), define custom methods that return `ViewRowImpl` or `List<ViewRowImpl>` instead. In summary, if your method signature defines `ViewRowImpl` or `List<ViewRowImpl>` as the data type, then the application automatically:

1. Looks up the row in the corresponding view object instance by primary key and/or alternate key
2. If the row is not found, then creates a new row
3. Applies the attribute changes in the found or new row

Whereas, if your method signature defines the `AttributeList` data type, then no automatic behavior is provided, and the actions performed and data modified by the custom method will be limited to your custom method's code.

What You May Need to Know About Row Finders and the ADF Web Service Operations

When you create a row finder for a view object, the service operations of the exposed service view instance support row lookup using non-key attributes defined by the row finder. For example, the service can invoke the row finder to locate the employee by their email address instead of the employee ID (a row key attribute). When the row finder is defined on a non-key attribute of the view object and the end user supplies a value for this attribute in the web service payload, the service will automatically invoke the row finder to identify the matching rows. Where the email address is the row finder's only required value, the service update operation would allow the end user to update the record of an employee who is identified only by their email address. For details about defining a row finder for use with the service view instance, see Working with Row Finders.

How to Service-Enable Individual View Objects

As a result of enabling the web service interface using the overview editor for the application module, JDeveloper automatically enables your parent view instance selections as Service Data Object (SDO) components. The generated SDO components for each view instance will reference the same namespace and will be configured with the same settings for options such as whether or not warnings are supported. You can use the Java page of the overview editor to customize the SDO definition of these existing service-enabled view objects. You can also use the Java page to service-enable view objects that were not added already to the service interface. For example, if you selected a parent view object that represents the master in a **master-detail relationship**, the child view object will not be automatically service-enabled. You can use the Java page of the overview editor for the child view object to individually add it to the service interface.

You use the Java page of the overview editor for the view object to configure the SDO name and namespace for a view object, or to selectively service-enable child view objects.

Before you begin:

It may be helpful to have an understanding of how the SDO framework supports service-enabled ADF view objects and enables web service clients to access rows of data and perform service operations. For more information, see Publishing Service-Enabled Application Modules.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Service-Enabled Application Modules.

You will need to complete these tasks:

1. Create the desired view objects, as described in How to Create an Entity-Based View Object, and How to Create a Custom SQL Mode View Object.
2. Enable the service interface for the application module, as described in How to Enable the Application Module Service Interface.
3. Optionally, set JDeveloper preferences to specify a default suffix for the names of generated SDO classes, modify the default subpackage where the service interface and classes reside, and set the default namespace prefix for the generated SDO schema and web service, as described in How to Set Preferences for Generating the ADF Web Service Interface.

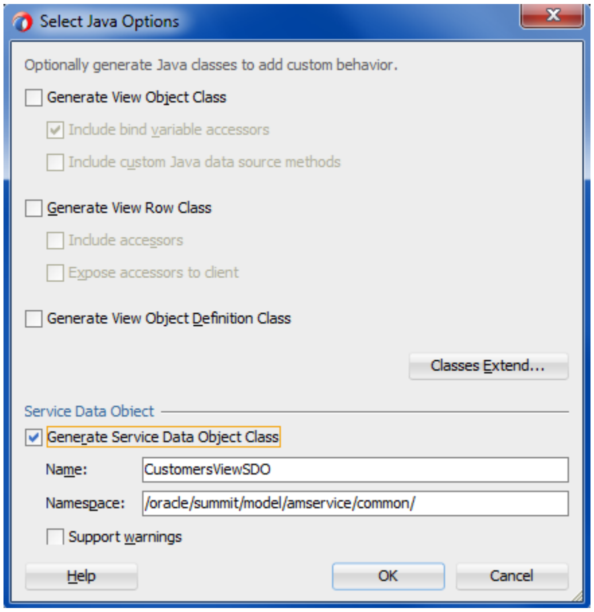
To set the SDO name and namespace for a view object:

1. In the Applications window, double-click the view object.
2. In the overview editor, click the **Java** navigation tab and then click the **Edit java options** button.
3. In the Select Java Options dialog, in the **Service Data Object** section, select **Generate Service Data Object Class**. Enter a value for the service data object name and the service data object's target namespace.

   The target namespace is a URI for the SDO component that you can assign to group similar SDO components together by entering the same URI.

   A default SDO namespace is created for you based on the SDO component's package name with periods replaced by "/". If you have defined a prefix for the namespace in the Service page of the Preferences dialog, the prefix will be added at runtime to the beginning of the namespace. For example, Figure 15-9 shows the default namespace based on the package name.

Figure 15-9 Service Data Object Name and Namespace Options



Optionally generate Java classes to add custom behavior.

☐ Generate View Object Class

☑ Include bind variable accessors

☐ Include custom Java data source methods

☐ Generate View Row Class

☐ Include accessors

☐ Expose accessors to client

☐ Generate View Object Definition Class

Service Data Object

☑ Generate Service Data Object Class

Name: CustomersViewSDO

Namespace: /oracle/summit/model/amservice/common/

☐ Support warnings

4. When you want to be able to extract warnings associated with the view rows of the service interface object, select **Support Warnings**.

   For example, your view object might have a range validator defined on the `Salary` attribute and failure handling for the validator is specified as Informational Warning.

   If **Support Warnings** is not selected, no informational messages will be returned with the service response.

   When enabled and a warning is generated from the underlying ADF business component object for one of the standard service operations or custom operations, the warning information will be captured by the response object. You can use the methods generated for the service object result class to extract the messages from the view rows, as described in Container Object Implemented by SDO Result Class and Interface.

5. Click **OK**.

How to Customize the SDO Properties of Service-Enabled View Objects

You can use the overview editor for the view object to customize the SDO component definition of the service-enabled view object. By default, all attributes of the service-enabled view object will be exposed as SDO properties. By customizing the view object definition, you can exclude individual SDO properties from participating in the service interface. In the case of SDO properties that define numeric values, you can associate two properties so they appear as a single complex type in the service interface. For example, you can associate one property that defines a currency code or unit of measure with another property that displays the numeric value. Currently, only the complex service types `AmountType` (a currency code) and `MeasureType` (a unit of measure) are supported.

Excluding Individual SDO Properties in a Generated SDO Component

As a result of enabling the web service interface using the overview editor for the application module, JDeveloper automatically enables your parent view instance selections as SDO components. Additionally, you can selectively service-enable individual child view objects and generate SDO components. By default, generated SDO components expose all attributes of their base view object definition as SDO properties. You can hide any attribute that you do not want the service interface to return as an SDO property.

You use the Attributes page of the overview editor to select the view object attribute that you want to exclude from the service interface. You then use the Details tab in the overview editor for the view object to hide the selected attribute from the SDO component.

Before you begin:

It may be helpful to have an understanding of how the SDO framework supports service-enabled ADF view objects and enables web service clients to access rows of data and perform service operations. For more information, see Publishing Service-Enabled Application Modules.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Service-Enabled Application Modules.

You will need to complete these tasks:

1. Create the desired view objects, as described in How to Create an Entity-Based View Object, and How to Create a Custom SQL Mode View Object.
2. Service-enable the desired view object, as described in How to Service-Enable Individual View Objects.

To exclude an SDO property from a service-enabled view object:

1. In the Applications window, double-click the view object.
2. In the overview editor, click the **Attributes** navigation tab.
3. In the Attributes page, select the attribute corresponding to the property that you want to exclude and then click the **Details** tab and deselect **SDO Property**.

Associating Related SDO Properties Using Complex Data Types

As a result of service-enabling the view object, JDeveloper automatically exposes SDO properties as XSD-defined service types that correspond to the data types of the underlying view object's attributes. In the case of attributes that define numeric values, you can change the SDO property's service type to associate a related property using one of these predefined service types:

- `AmountType` service type, for use with any property that defines a currency code
- `MeasureType` service type, for use with any property that defines a unit of measure

When you change the service type of an SDO property to either of these complex types, the service interface associates the two properties together and returns them as a single XML element. Both properties of the SDO component must be defined by attributes in the base service-enabled view object.

For example, suppose that your view object defines the `OrderTotal` attribute and a `CurrencyCode` attribute to specify the currency code of allowed countries. By default, the service interface exposes these attributes as SDO properties and returns each property as a separate XML element:

```
<OrderTotal>100.00</Price>
<CurrencyCode>USD</CurrencyCode>
```

If you change the type of the `OrderTotal` property (assume that the XSD file defines this property as a `decimal` type) to the complex type `AmountType` and then associate the `CurrencyCode` property, the service interface will return them as one XML element:

```
<OrderTotal CurrencyCode="USD">123.00</OrderTotal>
```

Also, when you generate a web service proxy, as described in Calling a Web Service Method Using the Proxy Class in an Application Module, the class treats the two values as one object:

```
AmountType price;
...
price.setValue(123.00);
price.setCurrencyCode("USD");
```

You use the Attributes page of the overview editor to select the view object attribute whose service type you want to customize. You use the Edit Attribute dialog that you display from the Attributes page of the overview editor to associate SDO properties for the selected attribute and select the predefined complex service type.

Before you begin:

It may be helpful to have an understanding of how the SDO framework supports service-enabled ADF view objects and enables web service clients to access rows of data and perform service operations. For more information, see Publishing Service-Enabled Application Modules.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Service-Enabled Application Modules.

You will need to complete these tasks:

1. Create the desired view objects, as described in How to Create an Entity-Based View Object, and How to Create a Custom SQL Mode View Object.
2. Service-enable the desired view object, as described in How to Service-Enable Individual View Objects.

To associate SDO properties in a service-enabled view object:

1. In the Applications window, double-click the view object.
2. In the overview editor, click the **Attributes** navigation tab.
3. In the Attributes page, select the attribute corresponding to the property that you will associate with another SDO property.

   The attribute you select must define a numeric type. For example, to associate a currency code with the attribute that displays the amount paid by a customer, you might select the `OrderTotal` attribute in the `Orders` service-enabled view object.
4. With the attribute selected, click the **Details** tab and then from the **XSD Type** dropdown list, choose the desired service type.

If the **XSD Type** dropdown is not enabled, return to the attribute list and select an attribute of type numeric. Attributes whose values are not a numeric type cannot be associated with the available complex service types.

The SDO framework supports the complex service types AmountType and MeasureType. Choose AmountType when the property you want to associate specifies currency information. Choose MeasureType when the property you want to associate specifies a unit of measure.

5. In the **currencyCode** or **unitCode** dropdown list, select the view object attribute to define the complex type.

   The dialog changes to display the dropdown list appropriate to the XSD type selection. Choose the attribute that is used to determine the currency code or unit of measure.

How to Support Nested Processing in Service-Enabled Master-Detail View Objects

When your data model defines master-detail relationships between parent and child view objects, the service operations that you enable for the master view object may not automatically be executed on the detail view object. Post operations on the detail view object are supported by default when the primary source **entity object** of the master view object is composed with the primary destination entity object of the detail view object. This master-detail relationship is known as a composition association and is created in JDeveloper when the source entity object contains the destination entity object as a logical, nested part, as described in What You May Need to Know About Composition Associations.

To support create/merge/update/process methods that post child details along with the parent, you will need to create a **view link** to define the master-detail relationship according to one of these scenarios:

- The view link uses the composition association, then post operations on the detail view object are supported by default.
- The view link is based on an association, and the association has the destination accessor generated, and the association has a custom property SERVICE_PROCESS_CHILDREN=true defined.
- The view link is not based on an association but has a custom property SERVICE_PROCESS_CHILDREN=true defined.

The custom property provides an alternative to using a composition association that makes it convenient to support nested processing for any view objects with a view link defined. You can define SERVICE_PROCESS_CHILDREN as a custom property in the overview editor for either the view link or the view link's association (when present).

To support get and find methods that retrieve child details along with the parent, the view link between the master and detail view objects must have the destination accessor generated. The destination accessor permits traversal of the hierarchy from the master to the detail view object.

Before you begin:

It may be helpful to have an understanding of how the SDO framework supports service-enabled ADF view objects and enables web service clients to access rows of data and perform service operations. For more information, see Publishing Service-Enabled Application Modules.

You may also find it helpful to have an understanding of how master-detail relationships are defined using view links or **entity associations**:

- For more information about view links, see Working with Multiple Tables in a Master-Detail Hierarchy.
- For more information about associations, see Creating and Configuring Associations.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Service-Enabled Application Modules.

You will need to complete these tasks:

1. Create the desired view objects and service-enable the child view object in the master-detail hierarchy, as described in How to Service-Enable Individual View Objects.
2. If the view link is not based on an association, confirm that a destination accessor exists for the view link by opening the view link in the overview editor and viewing the Relationship page. To generate the accessor so it appears in the Relationship page, click the **Edit Accessors** button, and then, in the View Link Properties dialog, select **Generate Accessor in View Object** for the destination accessor.

   If the view link is based on an association, then the destination accessor must exist for the association's destination entity object. To generate one, use the Relationship page of the overview editor for the association.

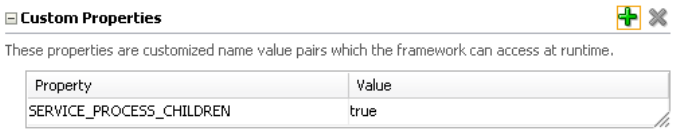
To support nested processing in a master-detail hierarchy:

1. If the view link for the master-detail hierarchy is not based on an association, then in the Applications window, double-click the view link; otherwise, if the view is based on an association, then in the Applications window, double-click the association.

   You can confirm how the view link was created in the Relationship page of the overview editor. The Attributes section names the source and destination attributes. When the view link is based on an association, the attribute hyperlinks will contain the names of the association. Otherwise, the hyperlinks will contain the names of the base entity objects.
2. In the overview editor, click the **General** navigation tab.

   The overview editor for the view link and the association display similar selections.
3. In the General page, expand the **Custom Properties** section, and then click the **Add Custom Property** icon and enter `SERVICE_PROCESS_CHILDREN` for the property and `true` for the property value, as shown in Figure 15-10.

Figure 15-10   Custom Property to Support Nested Processing



These properties are customized name value pairs which the framework can access at runtime.

Property	Value
SERVICE_PROCESS_CHILDREN	true

What Happens When You Create SDO Classes

When you create SDO classes, the following files are generated and appear in the Applications window under the owning view object:

- Service data object interface
- Service data object class
- Service data object schema file
- Service data object result class and Interface, generated when **Support Warnings** is enabled in the Select Java Options dialog

Do not modify the files generated for service-enabled view objects. The generated files implement required methods of the view object SDO interface.

Property Accessors Generated in the SDO Interface

The view object SDO interface contains strongly typed accessors for the SDO properties, as shown in the following example.

```
package oracle.summit.model.amservice.common;
  public interface OrdersViewSDO extends java.io.Serializable {
  public java.math.Integer getId();
  public void setId(java.lang.Integer value);
...}
```

View Object Interface Implemented by SDO Class

The view object SDO class implements the view object SDO interface and extends the `SDODataObject` class, which is Oracle's implementation of the SDO specification.

At runtime an instance of an SDO object represents a row in memory.

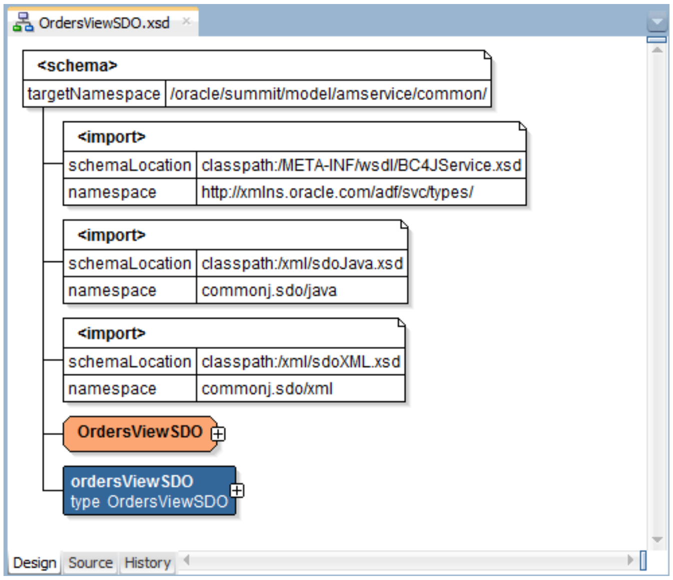
The SDO class is similar to the view row class, as shown in the following example.

```
package oracle.summit.model.amservice.common;
import org.eclipse.persistence.sdo.SDODataObject;
public class OrdersViewSDOImpl extends SDODataObject implements OrdersViewSDO
  {
  ...
  }
```

View Object Schema Generated in the SDO Schema File

The view object SDO schema file, as shown in Figure 15-16, is an XML Schema file which represents the SDO schema.

Figure 15-11 Generated SDO Schema



<schema>
targetNamespace /oracle/summit/model/amservice/common/

<import>
schemaLocation classpath:/META-INF/wsdl/BC4JService.xsd
namespace http://xmlns.oracle.com/adf/svc/types/

<import>
schemaLocation classpath:/xml/sdoJava.xsd
namespace commonj.sdo/java

<import>
schemaLocation classpath:/xml/sdoXML.xsd
namespace commonj.sdo/xml

Container Object Implemented by SDO Result Class and Interface

The view object SDO result class is a container object that allows a service operation to return a list of view rows (wrapped in service data objects) and a list of warnings associated with these view rows. Specifically, the service get operation returns the original object, while the create/update/merge/find/process operations return a wrapper object that contains a list of the original object and a list of information messages, and the delete operation returns only the informational message. If you have enabled the **Support Warnings** option for the service-enabled view object, you can use the generated method result interface to extract warnings.

The view object SDO result class, as shown in the following example, is similar to the view row class.

```
package oracle.summit.model.amservice.common;
import oracle.sdo.SDODataObject;
public class OrdersViewSDOResultImpl extends
   oracle.jbo.common.service.types.MethodResultImpl implements OrdersViewResult {

   public static final int START_PROPERTY_INDEX =
          oracle.jbo.common.service.types.MethodResultImpl.END_PROPERTY_INDEX +
1;
   public static final int END_PROPERTY_INDEX = START_PROPERTY_INDEX + 0;
   public OrdersViewResultImpl() {}
   public java.util.List getValue() {
      return getList(START_PROPERTY_INDEX + 0);
   }

   public void setValue(java.util.List value) {
      set(START_PROPERTY_INDEX + 0 , value);
   }
}
```

How to Expose a Declarative Find Operation Filtered By a Required Bind Variable

The ADF service interface framework allows you to expose declarative find operations to execute the query define by a view object you select. However, when that query uses a bind variable to filter the query results, the bind variable must not be specified as **Required** and **Updatable**. Because the service interface does not expose required, updatable bind variables, a find operation that you execute for such a view object would fail to return any result.

When you want to filter a query result using bind parameters, use the view criteria and expose it as a find operation on the service interface. A service interface find operation based on a view criteria that you create can specify required bind variables.

Before you begin:

It may be helpful to have an understanding of how the SDO framework supports service-enabled ADF view objects and enables web service clients to access rows of data and perform service operations. For more information, see Publishing Service-Enabled Application Modules.

You may also find it helpful to have an understanding of how to filter view object queries using bind variables. For more information, see Working with Bind Variables.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Service-Enabled Application Modules.

You will need to complete these tasks:

1. Create the desired application module, as described in How to Create an Application Module.
2. Create the view criteria, as described in How to Create Named View Criteria Declaratively. In the Edit View Criteria dialog, set the criteria item as a bind variable and set the **Validation** field to **Required**. This selection ensures that the query will not execute without a valid value.

To expose a find operation for a view criteria with required bind variable:

1. In the Applications window, double-click the application module.
2. In the overview editor, click the **Service Interface** navigation tab and then click the **Edit attributes of Service Interface** icon.

   Alternatively, you can select **Edit Service Custom Methods** if you have already defined the service interface.
3. In the Edit Service Interface dialog, select **Service View Instances** from the navigation list and add the view object that you want to filter with its named view criteria to the **Selected** list.
4. To expose the find operation, select the view instance, click the **View Criteria Find Operations** tab and then click the **Add View Criteria** button.
5. In the Configure View Criteria Find Operation dialog, choose the named view criteria for the find operation.

   The dialog displays the bind variable for the selected view criteria.

6. If you want to customize the bind variable name shown in the XML definition for the service, in the **Find Operations Parameters** section, double-click the XML name and edit the name.
7. Click **OK**.

How to Expose a Custom Find Method Filtered By a Required Bind Variable

As an alternative to exposing a declarative find operation that relies on a view criteria, you can define a service method in your data model project's application module implementation class. The class you create for this purpose allows you to encapsulate business service functionality into a single method that you implement. For details about the purpose of the custom application module implementation class, see Customizing an Application Module with Service Methods.

The following example shows a custom find method implemented in the *AppModuleName*`Impl.java` file to set the bind variable and execute the view object instance query. It uses `setNamedWhereClauseParam()` on the view object instance to set the bind variable. Before executing the query, the find method sets the view object in forward-only mode to prevent caching the view rows that the find method iterates over.

```
public class AppModuleImpl extends ApplicationModuleImpl
{
   public List<ViewRowImpl> findProducts(String location)
   {
      List<ViewRowImpl> result = new ArrayList<ViewRowImpl>();
      ViewObjectImpl vo = getProductsView1();
      vo.setNamedWhereClauseParam("TheLocation", location);
      vo.setForwardOnly(true);
      vo.executeQuery();
      while (vo.hasNext()) {
        result.add((ViewRowImpl)vo.next());
      }
      return result;
   }
}
```

Before you begin:

It may be helpful to have an understanding of how the SDO framework supports service-enabled view objects and enables web service clients to access rows of data and perform service operations. For more information, see Publishing Service-Enabled Application Modules.

You may also find it helpful to have an understanding of how to filter view object queries using bind variables. For more information, see Working with Bind Variables.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Service-Enabled Application Modules.

You will need to complete these tasks:

1. Create the custom application module class, as described in How to Generate a Custom Class for an Application Module.

2. Create the custom find method to programmatically filter a query result and set the required bind variable, as described in How to Add a WHERE Clause with Named Bind Variables at Runtime.

To expose a find method that sets a required bind variable:

1. In the Applications window, double-click the application module.
2. In the overview editor, click the **Service Interface** navigation tab and then click the **Edit attributes of Service Interface** icon.

   Alternatively, you can click the **Edit Service Custom Methods** icon if you have already defined the service interface.
3. In the Edit Service Interface dialog, select **Service View Instances** from the navigation list and add the find method that you defined to the **Selected** list.
4. Click **OK**.

How to Generate Asynchronous ADF Web Service Methods

By default, the web service supports synchronous service methods. This forces the invoking client application to wait for the response to return before it can continue with its work. In cases where the response returns immediately, this method of invoking the web service is common. However, because request processing can be delayed, it is often useful for the client application to continue its work and to handle the response later on.

Before you begin:

It may be helpful to have an understanding of how the SDO framework supports service-enabled view objects and enables web service clients to access rows of data and perform service operations. For more information, see Publishing Service-Enabled Application Modules.

You may also find it helpful to have an understanding of invoking web services using asynchronous request-response. For more information, see the "Overview of Asynchronous Web Services" section of the "Developing Asynchronous Web Services" chapter in *Developing Oracle Infrastructure Web Services*.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Service-Enabled Application Modules.

You will need to complete this task:

> Before you can deploy an asynchronous web service, you must configure the queues used to store the request and response. For information about configuring the request and response queues, see the "Creating the Request and Response Queues" section of the "Developing Asynchronous Web Services" chapter in *Developing Oracle Infrastructure Web Services*.

To expose asynchronous web service methods:

1. In the Applications window, double-click the application module.
2. In the overview editor, click the **Service Interface** navigation tab and then click the **Edit attributes of Service Interface** icon.
3. In the Edit Service Interface dialog, in the Service Interface page, select **Generate Asynchronous Web Service Methods**.

4. Click **OK**.

What Happens When You Generate Asynchronous ADF Web Service Methods

JDeveloper generates the service interface for the web service and enables the asynchronous service operation. As shown in the following example, the class annotation `@AsyncWebService` declares the `EmpService` service interface asynchronous and for each synchronous method in the interface, the service exposes an asynchronous method with the same method name and "`Async`" appended.

Exposing both synchronous and asynchronous methods in the same interface allows the web service client developer to decide how to invoke the operation through a web service proxy: by calling the appropriately named method. Note that developers must not invoke asynchronous methods through the ADF Business Components service proxy that is returned from `oracle.jbo.client.svc.ServiceFactory.getServiceProxy()` method.

In this example, because the `EmpService` service is enabled for asynchronous operation, the interface exposes the `getEmployeeAsync()` method and declares the `getEmployee()` method synchronous using the method annotation `@CallbackMethod(exclude=true)` to override the default operation (it is the `exclude=true` part that declares a method in the asynchronous service as synchronous). No annotation is required to declare the asynchronous service methods when the class annotation `@AsyncWebService` is present.

```
import javax.xml.ws.Action;
...
import oracle.webservices.annotations.async.AsyncWebService;
import oracle.webservices.annotations.async.CallbackMethod;

@SOAPBinding(parameterStyle=SOAPBinding.ParameterStyle.WRAPPED,
style=SOAPBinding.Style.DOCUMENT)
@PortableWebService(targetNamespace="http://xmlns.example.com/apps/service/",
name="EmpService",
                    wsdlLocation="oracle/apps/service/EmpService.wsdl")
@SDODatabinding(schemaLocation="oracle/apps/service/EmpService.xsd")
@AsyncWebService
public interface EmpService
{
   ...

   @WebMethod(action="http://xmlns.example.com/apps/service/getEmployee",
                       operationName="getEmployee")
   @RequestWrapper(targetNamespace="http://xmlns.example.com/apps/service/
types/",
                       localName="getEmployee")
   @ResponseWrapper(targetNamespace="http://xmlns.example.com/apps/service/
types/",
                       localName="getEmployeeResponse")
   @WebResult(name="result")
   @CallbackMethod(exclude=true)
   Emp getEmployee(@WebParam(mode = WebParam.Mode.IN, name="empno") Integer
empno)
      throws ServiceException;

   @WebMethod(action="http://xmlns.example.com/apps/service/getEmployeeAsync",
                        operationName="getEmployeeAsync")
   @RequestWrapper(targetNamespace="http://xmlns.example.com/apps/service/
types/",
                        localName="getEmployeeAsync")
   @ResponseWrapper(targetNamespace="http://xmlns.example.com/apps/service/
types/",
                        localName="getEmployeeAsyncResponse")
   @WebResult(name="result")
   @Action(input="http://xmlns.example.com/apps/service/getEmployeeAsync",
           output="http://xmlns.example.com/apps/service/
getEmployeeAsyncResponse")
   Emp getEmployeeAsync(@WebParam(mode = WebParam.Mode.IN, name="empno") Integer
empno);
}
```

The duplicate asynchronous methods delegate to the synchronous methods in the service implementation, as shown in the following example. This ensures that the underlying business logic is the same for operations declared as either synchronous or asynchronous.

```
...
import oracle.webservices.annotations.async.AsyncWebService;

@Stateless(name="oracle.apps.service.EmpServiceBean",
mappedName="EmpServiceBean")
@Remote(EmpService.class)
@PortableWebService(targetNamespace="http://xmlns.oracle.com/apps/service/",
                     serviceName="EmpService", portName="EmpServiceSoapHttpPort",
                     endpointInterface="oracle.apps.service.EmpService")
@Interceptors(ServiceContextInterceptor.class)
@AsyncWebService
public class EmpServiceImpl extends ServiceImpl implements EmpService
{
   ...

   /**
    * getEmployee: generated method. Do not modify.
    */
   public Emp getEmployee(Integer empno)
      throws ServiceException
   {
      return (Emp) get(new Object[] { empno }, "Employee", Emp.class);
   }

   /**
    * getEmployeeAsync: generated method. Do not modify.
    */
   public Emp getEmployeeAsync(Integer empno)
      throws ServiceException
   {
      return getEmployee(empno);
   }
}
```

What Happens at Runtime: How the Asynchronous Call Is Made

From the client's point of view, an asynchronous call consists of two one-way message exchanges. The sequence diagram in Figure 15-12 depicts the following flow:

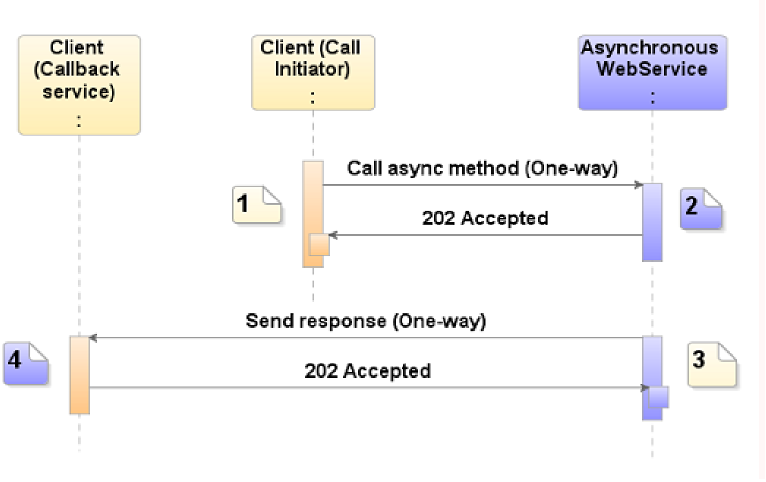
1. The client calls for the asynchronous operation. (In the figure, Step 1.)

2. The asynchronous service receives the request and returns the HTTP acknowledgement back to the client without actually processing the request. (In the figure, Step 2)
3. Eventually the asynchronous operation will complete and the module on the server side will send the response to the client side. (In the figure, Step 3.)

   To receive the response at the client side, the client must have some kind of HTTP listener, for example, a servlet or a web service.
4. The client side-generated web service (the Callback Service) receives the asynchronous responses. (In the figure, Step 4.)

   The module in Step 3 on the server side acts like a client to the callback service and so is referred as the callback client.

Figure 15-12 Asynchronous Call Sequence


Client (Callback service) :
Client (Call Initiator) :
Asynchronous WebService :
Call async method (One-way)
1
202 Accepted
2
Send response (One-way)
4
202 Accepted
3

How to Set Preferences for Generating the ADF Web Service Interface

You have additional control of the service generated by JDeveloper. You can set JDeveloper preferences to perform the following tasks:

- Specify a default suffix for the names of generated SDO classes
- Modify the default subpackage where the service interface and classes reside
- Shape the URI of the generated SDO schema and web service target namespace by setting a default namespace prefix and by specifying the names of packages that you want to exclude from the URI.

To set the SDO class name suffix:

1. In the main menu, choose **Tools** and then **Preferences**.
2. In the Preferences dialog, expand **ADF Business Components** and choose **Class Naming**.
3. In the **View Object** suffix list, enter a suffix for **SDO**, for example, `SDO`.

To set the default subpackage for the generated service interface:

1. In the main menu, choose **Tools** and then **Preferences**.
2. In the Preferences dialog, expand **ADF Business Components** and choose **Packages**.
3. In the **Relative Package Specification for Classes** list, specify the default package names:
   - To set the Service Interface package name, enter a value for the **Client Interface**. (The **Service Interface** displays the same package name you specify for the client interface). The default package name is `common`.
   - Enter a value for the **Service Interface Subpackage** of the **Service Interface**. The default subpackage name is `serviceinterface`.

   For example, if you enter `common` for **Service Interface** and `serviceinterface` for **Service Interface Subpackage** (the defaults), service interfaces for data model components in the data model package `oracle.summit.model.amservice` will be placed in the subpackage `oracle.summit.model.amservice.common.serviceinterface`.

To shape the URI of the generated SDO schema and service interface target namespace:

1. In the main menu, choose **Tools** and then **Preferences**.
2. In the Preferences dialog, expand **ADF Business Components** and choose **Service**.
3. Enter a value for the **Default Namespace Prefix** to be added to the beginning of the target namespace of the generated SDO schema and web service.

   For example, when you enable **Generate Service Data Object Class** in the Java page of the Create View Object wizard, a namespace prefix `www.globalcompany.example.com` would be added to the package name for the SDO to create the target namespace `www.globalcompany.example.com/oracle/summit/model/amservice/common/`.
4. Enter a semi-colon separated list of package names for the **Packages to Exclude** to shape the URI of the default target namespace.

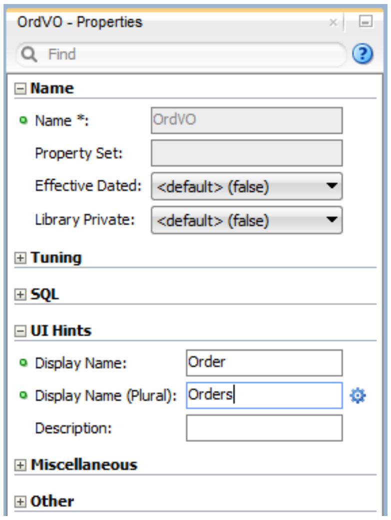
   For example, when you enter a namespace prefix `www.globalcompany.example.com` and the package name of the SDO is `globalcompany.summit.model.amservice/` the generated target namespace would incorrectly duplicate the package name `globalcompany`. In this situation, you can exclude the package name `globalcompany` to shape the URI for the target namespace as `www.globalcompany.example.com/summit/model/amservice/common/`.

   More than one package name can be excluded in a semi-colon separated list, as `a;a.b` to yield `<`*`namespace-prefix`*`>/c/d`. In the case of subpackage names, be sure to specify fully-qualified package names. For example to exclude package `c` in the package name `a.b.c.d`, enter `a.b.c`. In this case, the default target namespace yields `<`*`namespace-prefix`*`>/a/b/d/`.

How to Set Display Names for Service View Instances and Attributes

Display names are labels that support the web service consuming application to programmatically generate a user interface and to display the service view instance and its attributes with the correct identifying labels. To enable access to these display names at runtime by the consuming application, you use the Create Service Interface wizard to add the built-in method `getDfltObjAttrHints()` to the remote client service interface. This method lets the consuming application developer programmatically retrieve the **Display Name** UI hint associated with the service view instance and its attributes. You define the UI hints in the Property window for the view object, where you can define a singular label and a plural label for each view instance name and attribute name. For example, as Figure 15-13 shows, you may define UI hints for the `OrdersVO` view object with the labels `Order` and `Orders`.

Figure 15-13 Display Name UI Hints for View Objects



Name *: OrdVO
Property Set:
Effective Dated: <default> (false)
Library Private: <default> (false)
Display Name: Order
Display Name (Plural): Orders
Description:

Before you begin:

You will need to complete this task:

> Define the built-in method `getDfltObjAttrHints()` in the remote client service interface, as described in How to Enable the Application Module Service Interface.

To set UI Hint labels for the view instance and its attributes:

1. In the Applications window, double-click the application module that defines the view instance for the SDO.
2. In the Data Model list, select the view instance name and, below the list, click the **View Definition** link.
3. In the overview editor for the view object, click the **General** navigation tab.
4. In the main menu, choose **View** and then **Property Inspector**.
5. In the Property Inspector, to set the label for the view object instance, expand **UI Hints** and enter the labels for the **Display Name** and **Display Name (Plural)**.
6. In the overview editor for the view object, click the **Attributes** navigation tab and select the attribute that you want to label.
7. In the Property Inspector, to set the label for the view object attribute, expand **UI Hints** and enter the labels for the **Label** and **Display Name (Plural)**.

How to Secure the ADF Web Service for Access By SOAP Clients

At runtime, the web service client will invoke the service-enable methods of the application module through the SOAP protocol. You can configure an Oracle Web Service Manager (OWSM) security policy to enable authentication and authorization on the service. The security policy that you select will require the SOAP client call to provide credential information (or SAML token) as part of the SOAP header. You can also configure other policies to enable message protection (integrity and confidentiality) for inbound SOAP requests, for instance.

To secure the web service for SOAP clients:

1. Configure an OWSM authentication policy.
2. Configure an OWSM authorization policy.

You can enable authentication to require users to supply credentials before they have access to the service methods on the service interface. The type of authentication required is configured on the service implementation class using an OWSM authentication policy annotation.

You can enable permission checking to enable only users with sufficient privileges to invoke a service method on the service interface. Permission checking is configured on the service implementation class using this OWSM authorization annotation:

- `binding_permission_authorization_policy`

  This policy provides simple permission-based authorization for the request based on the authenticated Subject at the SOAP binding level. This policy ensures that the Subject has permission to perform the operation. This policy should follow an authentication policy where the Subject is established and can be attached to any SOAP-based endpoint.

As an alternative to the permission checking policy, you can configure one of these role-based OWSM security policies:

- `binding_authorization_denyall_policy`

  This policy provides simple role-based authorization for the request based on the authenticated Subject at the SOAP binding level. This policy denies all users with any roles. It should follow an authentication policy where the Subject is established and can be attached to any SOAP-based endpoint.
- `binding_authorization_permitall_policy`

  This policy provides a simple role-based authorization for the request based on the authenticated Subject at the SOAP binding level. This policy permits all users with any roles. It should follow an authentication policy where the Subject is established and can be attached to any SOAP-based endpoint.

For further details about the authorization policies, see Security Policies—Authorization Only in *Securing Web Services and Managing Policies with Oracle Web Services Manager*.

ADF service interface framework also provides an EJB interceptor `ServicePermissionCheckInterceptor` which does the same permission check as `oracle/binding_permission_authorization_policy`, and additionally covers RMI invocation to services. If your services will always be invoked through SOAP, you should attach the OWSM `oracle/binding_permission_authorization_policy` authorization policy. If your services will also be invoked through RMI, you should attach the EJB interceptor `ServicePermissionCheckInterceptor`.

Before you begin:

It may be helpful to have an understanding of how the SDO framework supports service-enabled ADF view objects and enables web service clients to access rows of data and perform service operations. For more information, see Publishing Service-Enabled Application Modules.

You may also find it helpful to have an understanding of the predefined authentication policies supported by OWSM. For more information, see Security Policies-Authentication Only in *Securing Web Services and Managing Policies with Oracle Web Services Manager*.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Service-Enabled Application Modules.

You will need to complete this task:

> Grant users access to the service, as described in How to Grant Test Users Access to the Service.

To configure an authentication and authorization:

1. In the Applications window, expand the application module, expand the **serviceinterface** node, and then double-click the service implementation class (*`AppModule`*`ServiceImpl.java`) node.

   In the web service generated from the `ServiceAppModule` application module in the `oracle.summit.model.amservice` package in the `SummitADF_Examples` workspace, the service implementation class is `ServiceAppModuleServiceImpl.java`.

2. If your services will always be invoked exclusively through SOAP (and not RMI), you should use an OWSM authorization policy as follows:

   a. In the source for the service implementation class, place your cursor on the `@PortableWebService` annotation.

      For example, `ServiceAppModuleServiceImpl.java` shows the annotation for the service as follows:

```
...
@PortableWebService(
  targetNamespace="/oracle/summit/model/amservice/common/",
  serviceName="ServiceAppModuleService",
  portName="ServiceAppModuleServiceSoapHttpPort",
  endpointInterface=
   "oracle.summit.model...common.serviceinterface.ServiceAppModuleService")
```

   b. In the Properties window, expand the **Policies** section, and click the button next to the **Multiple Policies** field.

   c. In the Edit Property: Multiple Policies dialog, select the desired security policy and click **OK**.

      For details about the security policies supported by OWSM, see Security Policies—Authorization Only in *Securing Web Services and Managing Policies with Oracle Web Services Manager*.

   d. Return to the source file and note that annotation `@PolicySet` is configured.

The @PolicySet annotation that you define for the service implementation class specifies the security requirements to potential clients. In this example, the annotation shows both the permission-checking authorization policy (oracle/binding_permission_authorization_policy) and an authentication policy:

```
...
@PortableWebService(
  targetNamespace="/oracle/summit/model/amservice/common/",
  serviceName="ServiceAppModuleService",
  portName="ServiceAppModuleServiceSoapHttpPort",
  endpointInterface=
   "oracle.summit.model...common.serviceinterface.ServiceAppModuleService")
@PolicySet(references = {
   @PolicyReference(value =
                        "oracle/wss_username_token_service_policy"),
   @PolicyReference(value =
                        "oracle/binding_permission_authorization_policy")
})
```

3. If your services will be invoked through SOAP and RMI, you should use ServicePermissionCheckInterceptor only (there is no need to use an OWSM authorization policy) as follows.

   In the source editor, place your cursor in the @Interceptors annotation and add ServicePermissionCheckInterceptor.class to enable permission checking at runtime.

```
...
@Interceptors({ ServiceContextInterceptor.class,
                ServicePermissionCheckInterceptor.class })
@Stateless(name=
            "oracle.summit.model.amservice.common.ServiceAppModuleServiceBean",
           mappedName="ServiceAppModuleServiceBean")
```

4. Save the service implementation class file.

How to Secure the ADF Web Service for Access By RMI Clients

Because the ADF web service is implemented as an EJB and deployed on Oracle WebLogic Server as Oracle Web Service's PortableWebService, the client application can invoke the service-enable methods of the application module through the RMI protocol.

To secure the web service for RMI clients:

1. Configure JNDI context properties to enable authentication.
2. Enable permission checking to configure an authorization policy.

Before you begin:

It may be helpful to have an understanding of the predefined authorization policies supported by OWSM. For more information, see Security Policies—Authorization Only in *Securing Web Services and Managing Policies with Oracle Web Services Manager*.

Enabling Authentication for RMI Clients

When the ADF web service is invoked through RMI, authentication is handled with the common JAAS login module. The login module can be passed the principal and credential as part of the JNDI initial context for the EJB in the calling application. If you do not define the JNDI context properties, the login module will attempt to obtain the caller's current security context.

When you choose to define remote JNDI context information, then these four JNDI context properties need to be added to the `connections.xml` file.

> **Note:**
>
> When you intend to test the service in JDeveloper using Integrated WebLogic Server, before deploying the service you can edit the JNDI context properties in the `connections.xml` file directly. However, when you deploy the service to standalone Oracle WebLogic Server, you will use Oracle Enterprise Manager to configure the JNDI context properties.

- `jndiFactoryInitial` should be set to `weblogic.jndi.WLInitialContextFactory`.
- `jndiProviderURL` is the JNDI provider URL that indicates the location of the JNDI server. The URL should be composed as `t3://<`*`hostname`*`>:<`*`server port`*`>`.

  When you test the service in JDeveloper, and your service is deployed to Integrated WebLogic Server, specify the JNDI provider URL of Integrated WebLogic Server: `t3://<`*`hostname`*`>`:7101.

  When you deploy the service to remote Oracle WebLogic Server, specify a URL like: `t3://localhost:8888`, where `t3` is the Oracle WebLogic protocol, `localhost` is the host name that the remote Oracle WebLogic Server instance runs in, `8888` is the port number.
- `jndiSecurityPrincipal` specifies the principal (user name) with permission to access the remote JNDI.

  When you test the service in JDeveloper Integrated WebLogic Server, you should omit this context property since no security is configured for the JNDI server on Integrated WebLogic Server.

  When you deploy the service to standalone Oracle WebLogic Server, the user name can be read from the file.
- `jndiSecurityCredentials` specifies the credentials (password) to be used for the security principal.

  When you test the service in JDeveloper Integrated WebLogic Server, you should omit this context property since no security is configured for the JNDI server on Integrated WebLogic Server.

  When you deploy the service to standalone Oracle WebLogic Server in a test environment, you can specify credentials in plain text for the JNDI provider. For example, you can specify `weblogic/weblogic1`, which are the default administrator user name/password credentials with sufficient privileges to access JNDI provider for Oracle WebLogic Server.

  When you deploy the service to a production environment, you must remove the plain text password to avoid creating a security vulnerability, and the `connections.xml` file must contain `<SecureRefAddr addrType="jndiSecurityCredentials"/>` with no password. To configure the service password for standalone Oracle WebLogic Server, you must use Oracle

Enterprise Manager, which will store the encrypted password in Oracle's credential store.

Before you begin:

It may be helpful to have an understanding of how the SDO framework supports service-enabled ADF view objects and enables web service clients to access rows of data and perform service operations. For more information, see Publishing Service-Enabled Application Modules.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Service-Enabled Application Modules.

To configure JNDI context properties to handle authentication:

1. In the Application Resources panel, expand the **Descriptors** and **ADF META-INF** nodes, and then double-click **connections.xml**.
2. If the source editor, use the JNDI context properties to specify the principal and credentials.

   If you are testing the service in JDeveloper's Integrated WebLogic Server, you only need to specify the `jndiProviderURL` property, as shown in the following example.

   ```
   <References xmlns="http://xmlns.oracle.com/adf/jndi">
     <Reference name="{www.globalcompany.com}ServiceAppModuleService"
                className="oracle.jbo.client.svc.Service" xmlns="">
        <Factory className="oracle.jbo.client.svc.ServiceFactory"/>
        <RefAddresses>
          ...
          <StringRefAddr addrType="jndiFactoryInitial">
             <Contents>weblogic.jndi.WLInitialContextFactory</Contents>
          </StringRefAddr>
          <StringRefAddr addrType="jndiProviderURL">
              <Contents>t3://a_hostname:7101</Contents>
          </StringRefAddr>
          ...
        </RefAddresses>
     </Reference>
     ...
   </References>
   ```

   If you are deploying the service for testing purposes to standalone Oracle WebLogic Server, you can use the `connections.xml` file to specify credentials for the JNDI provider. As the following example shows, you can specify `weblogic/weblogic1`, which are the default administrator user name/password credentials with sufficient privileges to access JNDI provider for Oracle WebLogic Server.

   ```
   <References xmlns="http://xmlns.oracle.com/adf/jndi">
     <Reference name="{www.globalcompany.com}ServiceAppModuleService"
                className="oracle.jbo.client.svc.Service" xmlns="">
        <Factory className="oracle.jbo.client.svc.ServiceFactory"/>
        <RefAddresses>
          ...
          <StringRefAddr addrType="jndiFactoryInitial">
             <Contents>weblogic.jndi.WLInitialContextFactory</Contents>
          </StringRefAddr>
          <StringRefAddr addrType="jndiProviderURL">
              <Contents>t3://localhost:8888</Contents>
          </StringRefAddr>
          <StringRefAddr addrType="jndiSecurityPrincipal">
   ```

```
            <Contents>weblogic</Contents>
        </StringRefAddr>
        <SecureRefAddr addrType="jndiSecurityCredentials">
            <Contents>weblogic1</Contents>
        </SecureRefAddr>
        ...
      </RefAddresses>
   </Reference>
   ...
</References>
```

If you are deploying the service to production Oracle WebLogic Server, you can use the connections.xml file to specify the user name. As shown in the following example, you must not specify the password.

```
<References xmlns="http://xmlns.oracle.com/adf/jndi">
  <Reference name="{www.globalcompany.com}ServiceAppModuleService"
             className="oracle.jbo.client.svc.Service" xmlns="">
     <Factory className="oracle.jbo.client.svc.ServiceFactory"/>
     <RefAddresses>
       ...
       <StringRefAddr addrType="jndiFactoryInitial">
          <Contents>weblogic.jndi.WLInitialContextFactory</Contents>
       </StringRefAddr>
       <StringRefAddr addrType="jndiProviderURL">
           <Contents>t3://localhost:8888</Contents>
       </StringRefAddr>
       <StringRefAddr addrType="jndiSecurityPrincipal">
           <Contents>a_username</Contents>
       </StringRefAddr>
       <SecureRefAddr addrType="jndiSecurityCredentials/">
       ...
     </RefAddresses>
  </Reference>
  ...
</References>
```

3. Save the file.

Configuring Authorization for RMI Clients

You can enable permission checking to allow only users with sufficient privileges to invoke a service method on the service interface. In order to enable permission checking, the ADF service interface framework provides an EJB interceptor named ServicePermissionCheckInterceptor. This EJB interceptor ensures permission checking is enforced at runtime. Currently, the interceptor is coded with the same logic that OWSM implements for the authorization policy binding_permission_authorization_policy.

Before you begin:

It may be helpful to have an understanding of how the SDO framework supports service-enabled ADF view objects and enables web service clients to access rows of data and perform service operations. For more information, see Publishing Service-Enabled Application Modules.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Service-Enabled Application Modules.

You will need to complete these tasks:

1. Configure the authentication policy for the service in the `connections.xml` file of the client application (the one invoking the service), as described in Enabling Authentication for RMI Clients.
2. Grant users access to the service, as described in How to Grant Test Users Access to the Service.

To configure a permission-based authorization policy:

1. In the Applications window, expand the **META-INF** node of the web service project and double-click the **ejb-jar.xml** node.
2. In the source editor, add the following `JpsInterceptor` definition required by the EJB for application roles evaluation.

```
<?xml version = '1.0' encoding = 'windows-1252'?>
<ejb-jar xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
         http://java.sun.com/xml/ns/j2ee/ejb-jar_3_0.xsd" version="3.0"
         xmlns="http://java.sun.com/xml/ns/javaee">
  <enterprise-beans>
    ...
  </enterprise-beans>
  <interceptors>
     <interceptor>
        <interceptor-class>
            oracle.security.jps.ee.ejb.JpsInterceptor
        </interceptor-class>
           <env-entry>
              <env-entry-name>application.name</env-entry-name>
              <env-entry-type>java.lang.String</env-entry-type>
              <env-entry-value>ApplicationName</env-entry-value>
              <injection-target>
                  <injection-target-class>
                      oracle.security.jps.ee.ejb.JpsInterceptor
                  </injection-target-class>
                  <injection-target-name>
                      application_name
                  </injection-target-name>
              </injection-target>
           </env-entry>
     </interceptor>
     ...
  <interceptors>
  <assembly-descriptor>
     <interceptor-binding>
        <ejb-name>*</ejb-name>
        <interceptor-class>
             oracle.security.jps.ee.ejb.JpsInterceptor
        </interceptor-class>
     </interceptor-binding>
  </assembly-descriptor>
</ejb-jar>
```

3. In the Applications window, expand the application module, expand the **serviceinterface** node, and then double-click the service implementation class (*`AppModule`*`ServiceImpl.java`) node.

4. In the source editor, place your cursor at the end of your annotations section and add the annotation named `ServicePermissionCheckInterceptor` to enable permission checking at runtime.

```
...
@Stateless(name="oracle.summit.model.amservice.common.
                                        ServiceAppModuleServiceBean")
@Remote(ServiceAppModuleService.class)
@PortableWebService(targetNamespace="http://www.globalcompany.com/
                       ServiceAppModuleService",
                       serviceName="ServiceAppModuleService",
                       portName="ServiceAppModuleServiceSoapHttpPort",
                       endpointInterface="oracle.summit.model.amservice.
                            common.serviceinteface.ServiceAppModuleService")
@CallByReference
@Interceptors({ServiceContextInterceptor.class,
                        ServicePermissionCheckInterceptor.class})
```

5. Save the files.

How to Grant Test Users Access to the Service

After you have configured the authorization policy for the service, you must configure the Oracle Platform Security Services (OPSS) security provider to specify which users can invoke method on the service. At design time, you perform this task by editing the `jazn-data.xml` configuration file to create application roles and make an invoke permission grant to the desired application roles. Then when you deploy the service, the administrator for the target Oracle WebLogic Server will associate enterprise users with the application roles you specify. This allows you to confer the right to invoke a service method to any user who is a member of that application role. Users who are members of a role that has not been granted the invoke permission, will denied access to the service method.

The invoke permission for Oracle Web Services is defined by the `oracle.wsm.security.WSFunctionPermission` class. You can grant the invoke permission to the application roles you define for all the methods of the service or just to individual methods.

Before you begin:

It may be helpful to have an understanding of how the SDO framework supports service-enabled ADF view objects and enables web service clients to access rows of data and perform service operations. For more information, see Publishing Service-Enabled Application Modules.

You may also find it helpful to have an understanding of application roles and the OPSS security provides. For more information, see the "Introduction to Oracle Platform Security Services" chapter in *Securing Applications with Oracle Platform Security Services*.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Service-Enabled Application Modules.

You will need to complete these tasks:

1. Configure authentication and authorization policies for your service, as described in How to Secure the ADF Web Service for Access By SOAP Clients, or How to Secure the ADF Web Service for Access By RMI Clients.

2. Add the OPSS security provider configuration file to your project by creating a `jazn-data.xml` deployment descriptor. For more information about deployment descriptors, see the "Deploying Applications" chapter in *Developing Applications with Oracle JDeveloper*.

   Note that in JDeveloper you open the New Gallery, expand **General**, select **Deployment Descriptors** and then **Oracle Deployment Descriptors**, and click **OK**.

3. Create the desired application roles that you want to make grants to, as described in Creating Application Roles.
4. For the purpose of testing your web service in JDeveloper using Integrated WebLogic Server, you can populate the application roles with test users, as described in Creating Test Users.

To grant the web service permission to application roles in the jazn-data.xml file:

1. In the main menu, choose either **Application > Secure > Resource Grants** or **Application > Secure > Entitlement Grants**.

   You can grant multiple privileges to an application role as an entitlement set or you can grant individual privileges to a resource. Create an entitlement grant to aggregate privileges that together secure a specific end user duty. For details about entitlement grants, see How to Aggregate Resource Grants as Entitlement Grants.

2. In the editor window for the `jazn-data.xml` file, click the **Source** tab.
3. In the source for the `jazn-data.xml` file, expand the `<policy-store>` element to view all ADF security policies that you already defined for your application.

   Currently, this release does not provide an editor to create an application security policy; you will need to manually create the policy in the source for the `jazn-data.xml` file.

4. Inside the `<jazn-policy>` element, create a `<grant>` element that defines the `<grantee>` with the desired application role and the `<permission>` with the fully qualified class name of the OWSM permission class (`oracle.wsm.security.WSFunctionPermission`), the permission target name that uniquely identifies the service method, and the invoke method action that you want to grant to the application role principal.

   Your finished source should look similar to this:

```
<grant>
  <grantee>
    <principals>
      <principal>
         <class>oracle.security.jps.service.policystore.ApplicationRole</
class>
         <name>customers</name>
      </principal>
     </principals>
  </grantee>
  <permissions>
     <permission>
       <class>oracle.wsm.security.WSFunctionPermission</class>
      <name>www.globalcompany.example.com/
                                 ServiceAppModuleService#CreateAccount</name>
       <actions>invoke</actions>
     </permission>
```

```
   </permissions>
</grant>
```

The `<principal>` element is defined by the application role class name `oracle.security.jps.service.policystore.ApplicationRole` and an application role name that you already created. For example, if you created an application role `customers` and you want to grant invoke service method permission to the members of that role, then enter `customers`.

The `<permission>` element is defined by the OWSM class name `oracle.wsm.security.WSFunctionPermission` and the permission target name. The permission target name is formed by appending */serviceInterfaceName* and *#serviceMethodName* (or wildcard character) to the service target namespace.

> **Tip:**
>
> You can find the target namespace and service name from the WSDL definition file for the service. In the Applications window, double-click the WSDL file under the **serviceinterface** node to view the `name` and `targetNamespace` definitions.

For example, the WSDL definition file for the Summit ADF sample application might define the following name and namespace:

```
<wsdl:definitions
     name="ServiceAppModuleService"
     targetNamespace="www.globalcompany.example.com"
```

Assume that you want to grant a permission to allow authorized users to invoke a `CreateOrder` service method on the service interface with these `SummitADF_Examples` workspace name and namespace, you would enter the target name like this:

```
www.globalcompany.example.com/ServiceAppModuleService#CreateOrder
```

Alternatively, you can enter the target name using the wildcard character * to grant all operations of the service interface in a single permission:

```
www.globalcompany.example.com/ServiceAppModuleService#*
```

The actions that you can enter are defined by the permission class. In this case, `oracle.wsm.security.WSFunctionPermission` defines the single action `invoke`.

5. Save the changes to the `jazn-data.xml` file.

How to Enable Support for Binary Attachments for SOAP Clients

The ADF service interface framework supports using Message Transmission Optimization Mechanism (MTOM) to handle sending binary data in any service method that operates on a `ViewRow` with a `BlobDomain/ClobDomain` attribute. This permits binary data to accompany XML messages, for example when images are required to document an insurance claim. The SDO data objects of the service-enabled application module maps `BlobDomain/ClobDomain` to `javax.activation.DataHandler`. These `DataHandler` properties could be passed as attachments during SDO data object marshalling/unmarshalling when the web service is called using the SOAP protocol.

To enable MTOM support for your SOAP protocol, you must add the `@MTOM` annotation to the service implementation class (for example, `ServiceAppModuleServiceImpl.java`) and your method must operate on a view row with `BlobDomain`/`ClobDomain` attribute.

Before you begin:

It may be helpful to have an understanding of how the SDO framework supports service-enabled ADF view objects and enables web service clients to access rows of data and perform service operations. For more information, see Publishing Service-Enabled Application Modules.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Service-Enabled Application Modules.

You will need to complete this task:

> Configure a character encoding format that the ADF service interface framework can use to convert data defined by the view object as ClobDomain attribute, as described in How to Specify Character Encoding for ClobDomain Type Attributes.

To enable support for sending binary data attachments:

1. In the Applications window, expand the application module, expand the **serviceinterface** node, and then double-click the service implementation class (`*AppModule*ServiceImpl.java`) file.

   In the web service generated from the `ServiceAppModule` application module in the `oracle.summit.model.amservice` package in the `SummitADF_Examples` workspace, the service implementation class is `ServiceAppModuleServiceImpl.java`.

2. In the source for the service implementation class, place your cursor anywhere in the annotation section.

   For example, in the `ServiceAppModuleServiceImpl.java` the annotation section for the service is:

```
...
@Stateless(name="oracle.summit.model....common.ServiceAppModuleServiceBean",
    mappedName="ServiceAppModuleServiceBean")
@Remote(ServiceAppModuleService.class)
@PortableWebService(targetNamespace="www.globalcompany.example.com",
    serviceName="ServiceAppModuleService",
    portName="ServiceAppModuleServiceSoapHttpPort",
    endpointInterface=
       "oracle.summit.model...common.serviceinterface.ServiceAppModuleService")
...
```

3. In the Properties window, expand the **Web Services** section and select **Enable MTOM**.

   JDeveloper adds the `@MTOM` annotation to the annotations section of the file.

```
...
@Stateless(name="oracle.summit.model....common.ServiceAppModuleServiceBean",
    mappedName="ServiceAppModuleServiceBean")
@Remote(ServiceAppModuleService.class)
```

```
@PortableWebService(targetNamespace="www.globalcompany.example.com",
    serviceName="ServiceAppModuleService",
    portName="ServiceAppModuleServiceSoapHttpPort",
    endpointInterface=
       "oracle.summit.model...common.serviceinterface.ServiceAppModuleService")
@MTOM
...
```

How to Specify Character Encoding for ClobDomain Type Attributes

A consequence of working with large string data, defined in the database as CLOB type, is the character encoding used when converting string data to and from binary data. When you work with `ClobDomain` attributes in your service data objects, the ADF service interface framework streams the data by mapping the attribute value to a base64-encoded SDO property. This mapping to base64 binary stream does not include the character set of the original data.

To support converting string data to and from binary data, the ADF service interface framework allows you to configure the encoding character set for individual `ClobDomain` attributes of the view object. Alternatively, you can configure the encoding character set at the level of the view object definition, when you want all `ClobDomain` attributes to be converted using the same character encoding scheme.

When the framework needs to convert the binary stream to and from characters for `ClobDomain` attributes, it can apply the character encoding specified as a value for either of these custom properties that you have defined on the view object:

- `SERVICE_USE_SERVER_DEFAULT_CHARSET`
- `SERVICE_CLOB_CONVERSION_CHARSET`

The custom property `SERVICE_USE_SERVER_DEFAULT_CHARSET` is useful when you do not know the specific character encoding used by the database. Since the character set of the server JVM will be locale dependent, you should take that into account when defaulting to the server's encoding character set.

The ideal configuration for `ClobDomain` attributes relies on the custom property `SERVICE_CLOB_CONVERSION_CHARSET` and the character encoding that you specify. This configuration assumes that you can verify the character encoding of the target data to ensure that the ADF service interface framework does not convert using a different encoding character set.

> **Note:**
>
> When no custom property is defined on the view object to configure character encoding, at runtime, the ADF service interface framework will default to the UTF-8 character encoding scheme to handle conversion of `ClobDomain` data. UTF-8 is a Unicode encoding that provides multilingual support for CLOB data that is supported by Oracle Database.

Before you begin:

It may be helpful to have an understanding of how the SDO framework supports service-enabled ADF view objects and enables web service clients to access rows of data and perform service operations. For more information, see Publishing Service-Enabled Application Modules.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Service-Enabled Application Modules.

You will need to complete this task:

> Enable MTOM support by adding the `@MTOM` annotation to the service implementation class to allow service methods to operate on a view row with `ClobDomain` attributes, as described in How to Enable Support for Binary Attachments for SOAP Clients.

To configure character encoding for individual ClobDomain attributes of the view object:

1. In the Applications window, double-click the view object that contains ClobDomain attributes.
2. In the overview editor, click the **Attributes** navigation tab.
3. In the Attributes page, select the attribute with **Clob** displayed in the **Type** column.
4. Click the **Custom Properties** tab, click the **Add** icon and choose **Non-Translatable Property** from the dropdown list and then enter one of the following predefined custom properties in the **Property** name field:

   `SERVICE_USE_SERVER_DEFAULT_CHARSET` - when you want the character encoding of the JVM to be used to convert the data. Enter `true` for the **Value**.

   `SERVICE_CLOB_CONVERSION_CHARSET` - when you want to specify a known character encoding that corresponds to the character encoding of the data. Enter the desired character encoding for the **Value**. For example, you might enter `UTF-16`. Note, when no value is supplied, the default UTF-8 will be used.

To configure character encoding for all ClobDomain attributes of the view object:

1. In the Applications window, double-click the view object that contains ClobDomain attributes.
2. In the overview editor, click the **General** navigation tab.
3. In the General page, expand the **Custom Properties** section, click the **Add** icon and choose **Non-Translatable Property** from the dropdown list and then enter one of the following predefined custom properties in the **Property** name field:

   `SERVICE_USE_SERVER_DEFAULT_CHARSET` - when you want the character encoding of the JVM to be used to convert the data. Enter `true` for the **Value**.

   `SERVICE_CLOB_CONVERSION_CHARSET` - when you want to specify a known character encoding that corresponds to the character encoding of the data. Enter the desired character encoding for the **Value**. For example, you might enter UTF-16. Note, when no value is supplied, the default UTF-8 will be used.

How to Test the Web Service Using Integrated WebLogic Server

You can run the web service in JDeveloper using Integrated WebLogic Server. You can also deploy the web service to Oracle WebLogic Server to test the service.

To run and test using Integrated WebLogic Server:

1. In the Applications window, expand the application module, expand the **serviceinterface** node, and then select the service implementation class (`AppModuleServiceImpl.java`) file.

   In the web service generated from the `ServiceAppModule` application module in the `oracle.summit.model.amservice` package in the `SummitADF_Examples` workspace, the service implementation class is `ServiceAppModuleServiceImpl.java`.

2. Right-click the service implementation class file, and choose **Run** or **Debug**.

   The Create Default Domain dialog appears the first time you run the application and start a new domain in Integrated WebLogic Server. Use the dialog to define an administrator password for the new domain. Passwords you enter can be eight characters or more and must have a numeric character.

   JDeveloper initializes the server instance, and then deploys the application and starts the web service. During this time, the output from these processes is displayed in the **Running** tab of the Log window. After the web service has started, the target URL is also displayed in the Log window.

   > **Tip:**
   >
   > In the Log window, you can click the target URL link to launch the HTTP Analyzer. This is a convenient shortcut for testing with JDeveloper Integrated WebLogic Server. For more information about the HTTP Analyzer, see the "Auditing and Monitoring Java Projects" chapter in *Developing Applications with Oracle JDeveloper*.

3. Copy the target URL (beginning with `http://`) from the Log window.

   For example, if the Log window displays:

   ```
   http://<ipaddress>/ADFServiceDemo-ADFModel-context-root
   ```

   Integrated WebLogic Server will display the service endpoint URL. So there is no need to append the service name.

4. Launch a web browser, paste the target URL you copied from the Log window into the browser address field, and submit the HTTP request.

5. In the test page, choose the operation you want to invoke from the **Operations** dropdown list and enter sample data in its parameter fields.

6. When you are ready, press **Invoke** to submit the operation and view the results for the operation in the Test Results page.

   The Test Results page displays the XML Soap format of the information returned by the operation.

How to Prevent Custom Service Methods from Timing Out

When you test the web service you may find that some of your custom methods exceed the established timeout limitation established by the Java Transaction API (JTA). The JTA timeout setting establishes an execution boundary for service methods that by default may not exceed 30 seconds. You can use the Administration Console for Oracle WebLogic Server to increase the JTA timeout setting. If you still receive a timeout exception or you anticipate that the custom methods of the service interface may be long running, you can specify an EJB transaction attribute for the stateless session bean to prevent the EJB from executing those methods in a JTA transaction.

For example, calls to the `afterCommit()` method to execute JDBC SQL statements from view objects or entity objects can result in the following `SQLException`: `java.sql.SQLException: The transaction is no longer active - status: 'Committed'. No further JDBC access is allowed within this transaction`

To perform custom operations in the case of the afterCommit() method, the service-enabled method annotation `TransactionAttribute` must be configured as `NOT_SUPPORTED`. To make a custom method exempt from timing out, you set `TransactionAttributeType.NOT_SUPPORTED` in the Properties window specifically for that method. As the following example shows, JDeveloper updates the method by adding the annotation `@TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED)`.

```
@TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED)
public void updateCustomerInterests(List<String> pCategoryIds) throws
                                                       ServiceException {
        invokeCustom((Method) _map.get("updateCustomerInterests"),
             new Object[] { pCategoryIds }, new String[] { null }, false);
}
```

Because a method with this transaction attribute setting will not be executed in the JTA transaction, it is your responsibility to enforce control over the transaction using the ADF Business Components methods of the `oracle.jbo.ApplicationModule` and `oracle.jbo.Transaction` interfaces. For instance, as the following example shows, the methods of the implementation class of the application module that you service-enabled, will need to call `am.getDBTransaction().commit()` or `rollback()` in order to complete the transaction.

```
public void updateCustomerInterests(List pCategoryIds) {
     try
     {
        if (pCategoryIds != null && pCategoryIds.size() > 0) {
            List<Integer> copyOfCategoryIds = (List<Integer>)
                                       this.cloneList(pCategoryIds);
            ViewObject selectedCategories =
                                       this.getSelectedCategoriesShuttleList();
            RowSetIterator rsi = selectedCategories.createRowSetIterator(null);
            // remove rows for the current user not in the list of product keys
            while (rsi.hasNext()) {
                Row r = rsi.next();
                Number interestId = (Number)r.getAttribute("CategoryId");
                // existing row is in the list, we're ok, so remove from list.
                if (copyOfCategoryIds.contains(interestId)) {
                    copyOfCategoryIds.remove(interestId);
                }
                // if the existing row is in not list, remove it.
                else {
                    r.remove();
                }
            }
            rsi.closeRowSetIterator();
            // at this point, add new rows for the keys that are left
                    for (int i =0 ;i < copyOfCategoryIds.size(); i++ )  {
                        Row newRow = selectedCategories.createRow();
                        selectedCategories.insertRow(newRow);
                        newRow.setAttribute("CategoryId", (String)
                                       copyOfCategoryIds.get(i).toString());
                    }
        this.getTransaction().commit();
        }
     }
     catch (JboException e)
     {
        this.getTransaction().rollback();
        throw e;
     }
}
```

You should not change the default transaction attribute setting for the standard service methods generated for the service interface (see Table 15-1). The standard methods will execute within the default execution boundary set for the JTA transaction.

Before you begin:

It may be helpful to have an understanding of how the SDO framework supports service-enabled ADF view objects and enables web service clients to access rows of data and perform service operations. For more information, see Publishing Service-Enabled Application Modules.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Service-Enabled Application Modules.

To prevent custom methods from executing in a JTA transaction:

1. In the Applications window, expand the application module, expand the **serviceinterface** node, then double-click the service implementation class (`AppModuleServiceImpl.java`) file.

   In the web service generated from the `ServiceAppModule` application module in the `oracle.summit.model.amservice` package in the `SummitADF_Examples` workspace, the service implementation class is `ServiceAppModuleServiceImpl.java`.

2. In the source editor, locate the custom method that you want to prevent from timing out and place your cursor on the method.

3. In the Properties window, expand **Stateless Session Bean**, and then select **TransactionAttributeType.NOT_SUPPORTED** from the **TransactionAttribute** dropdown list.

   As shown in `updateCustomerInterests()` in the NOT_SUPPORTED annotation for the service method example above, JDeveloper updates the custom service method with the annotation.

4. Save the service implementation class.

5. In the Applications window, double-click the application module implementation class (`AppModuleImpl.java`) file.

   The implementation class defines the custom methods that you exposed through the service interface. In the `SummitADF_Examples` workspace, the application module implementation class is `oracle.summit.model.amservice.ServiceAppModuleImpl.java`.

6. In the source editor, in the custom method that implements the service method that you previously set the **TransactionAttribute** property on, add the custom code that will commit and roll back the transaction.

As the above implementation example for the service method shows, if you configured the **TransactionAttribute** property on the service method named `updateCustomerInterests()`, then you would open the implementation class for the application module, locate the custom method `updateCustomerInterests()`, and add `am.getDBTransaction().commit()` and `rollback()` as part of the method's `try` and `catch` statements.

7. Save the application module implementation class.

How to Deploy Web Services to Oracle WebLogic Server

You can deploy the web service to Oracle WebLogic Server, for example to perform a second stage of testing the service.

Before you begin:

It may be helpful to have an understanding of how the SDO framework supports service-enabled ADF view objects and enables web service clients to access rows of data and perform service operations. For more information, see Publishing Service-Enabled Application Modules.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Service-Enabled Application Modules.

You will need to complete these tasks:

1. If your project contains a standard web service (a Java class or interface with the `@WebService` annotation), you must remove the `@WebService` annotation from the Java class before you can deploy the Business Components web service. If you attempt to deploy the Business Components web service from the same project as the standard web service, deployment will fail with an Oracle WebLogic Server exception error. It is therefore necessary to create standard web services in a separate project from your ADF Business Components service.
2. If you created an asynchronous web service, before you can deploy the service you must configure the queues used to store the request and response. For information about configuring the request and response queues, see the "Creating the Request and Response Queues" section of the "Developing Asynchronous Web Services" chapter in *Developing Oracle Infrastructure Web Services*.
3. If you configured authorization for the web service, as described in How to Secure the ADF Web Service for Access By SOAP Clients, edit the `weblogic-application.xml` file to define application ID parameters. This file appears in the Applications window in the Application Resources panel, under the **Descriptors** and **META-INF** nodes.

   Add the following `<application-param>` definition as the first element:

```
<weblogic-application xmlns="http://www.bea.com/ns/weblogic/weblogic-
application"
                  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                  xsi:schemaLocation=
                        "http://www.bea.com/ns/weblogic/weblogic-
application.xsd">
   <application-param>
      <param-name>jps.policystore.applicationid</param-name>
      <param-value>ApplicationName</param-value>
   </application-param>
   ...
</weblogic-application>
```

Note that the *ApplicationName* that you enter must match the name identified in the `jazn-data.xml` policy store definition:

```
<jazn-data>
   <policy-store>
        <applications>
            <application>
                <name>ApplicationName</name>
                <app-roles>
                    ...
                </app-roles>
                <jazn-policy>
                    ...
                </jazn-policy>
            </application>
        </applications>
    </policy-store>
</jazn-data>
```

4. Edit the `ejb-jar.xml` file to add the following `JpsInterceptor` definition required by the EJB for application roles evaluation. This file appears in the Applications window under the **META-INF** node of the web service project.

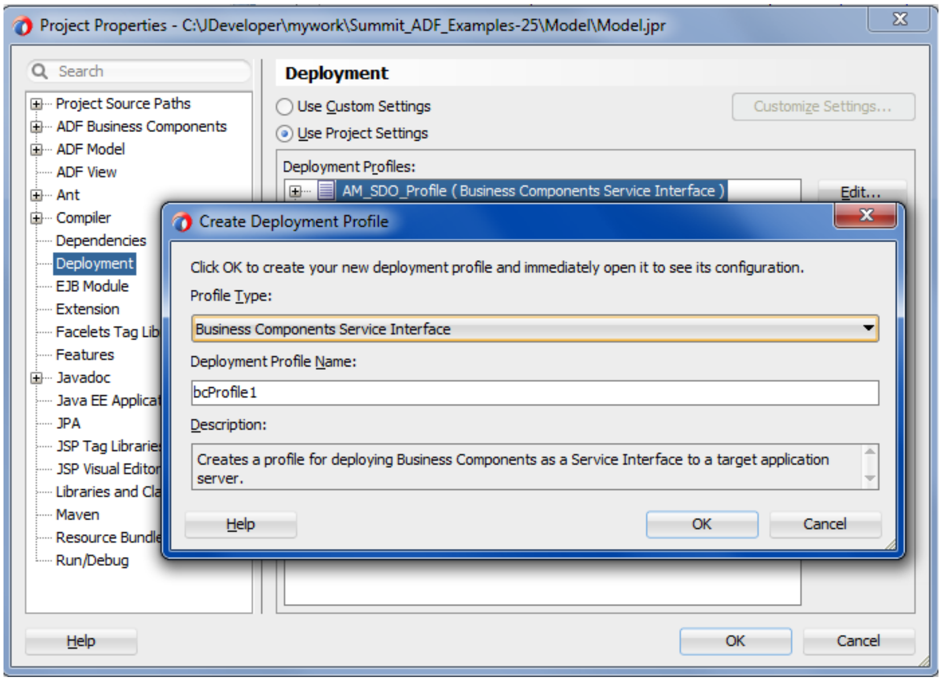
```
<?xml version = '1.0' encoding = 'windows-1252'?>
<ejb-jar xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
         http://java.sun.com/xml/ns/j2ee/ejb-jar_3_0.xsd" version="3.0"
         xmlns="http://java.sun.com/xml/ns/javaee">
  <enterprise-beans>
    ...
  </enterprise-beans>
  <interceptors>
     <interceptor>
        <interceptor-class>
            oracle.security.jps.ee.ejb.JpsInterceptor
        </interceptor-class>
           <env-entry>
              <env-entry-name>application.name</env-entry-name>
              <env-entry-type>java.lang.String</env-entry-type>
              <env-entry-value>ApplicationName</env-entry-value>
              <injection-target>
                  <injection-target-class>
                      oracle.security.jps.ee.ejb.JpsInterceptor
                  </injection-target-class>
                  <injection-target-name>
                      application_name
                  </injection-target-name>
              </injection-target>
           </env-entry>
     </interceptor>
     ...
  <interceptors>
  <assembly-descriptor>
     <interceptor-binding>
        <ejb-name>*</ejb-name>
        <interceptor-class>
             oracle.security.jps.ee.ejb.JpsInterceptor
        </interceptor-class>
     </interceptor-binding>
  </assembly-descriptor>
</ejb-jar>
```

Note that *ApplicationName* must also match the application name identified in the jazn-data.xml policy store definition.

To deploy to Oracle WebLogic Server:

1. Create an application server connection to Oracle WebLogic Server:
   a. In the main menu, choose **Window** and then **Application Servers**.
   b. In the Application Servers window, right-click the **Application Servers** node and choose **New Application Server**, and complete the Create Application Server Connection wizard.
2. Create a service deployment profile:
   a. In the Applications window, right-click the project that contains the web service and choose **Project Properties**.
   b. In the Project Properties dialog, open the Deployment page and click **New**.
   c. In the Create Deployment Profile dialog, choose **Business Components Service Interface** as the archive type, as shown in Figure 15-14.

Figure 15-14 Dialog for Creating a Business Components Service Deployment Profile



Project Properties - C:\JDeveloper\mywork\Summit_ADF_Examples-25\Model\Model.jpr

Deployment

○ Use Custom Settings
● Use Project Settings

Deployment Profiles: AM_SDO_Profile (Business Components Service Interface)

Create Deployment Profile

Click OK to create your new deployment profile and immediately open it to see its configuration.

Profile Type: Business Components Service Interface

Deployment Profile Name: bcProfile1

Description: Creates a profile for deploying Business Components as a Service Interface to a target application server.

3. In the main menu, choose **Application > Deploy > *deployment profile*** for the deployment profile you created.
4. In the Deploy wizard, on the Deployment Action page, select **Deploy to Application Server** and click **Next**.
5. On the Select Server page, select the application server connection.
6. Click **OK**.

Accessing Remote Data Over the Service-Enabled Application Module

ADF Application Modules publish Service Data Objects (SDOs). You can create service-backed entity objects to utilize SDOs to expose business services in SOAP services.

ADF Business Components application modules offer built-in support for web services and for publishing rows of view object data as service data objects (SDOs). Entity objects that you create in your local data model project can utilize the SDO services that the service-enabled application module exposes on its service interface. By creating service-backed entity objects in your local project, you avoid having to work directly with the service proxy and SDOs programmatically for all common web service data access tasks.

The Create Entity Object wizard makes it easy for you to choose between a local database and the ADF Business Components web service when you create the entity object, as described in How to Use Service-Enabled Entity Objects and View Objects. In this way, service-enabled application modules provide an alternative way to access data that is not available locally in the database.

Once you create the service-backed entity object, you will be able to create view objects, view links, and view criteria to filter the data at runtime. You will also be able to utilize these view objects in your data model as though you were working with locally available data accessed from database tables.

The following sections describe how to augment your data model project using a service-enabled ADF application module.

How to Use Service-Enabled Entity Objects and View Objects

You will want to use the service-backed components as part of your application design strategy when one of the following conditions is true:

- The client data model project needs to work with data from a service-enabled application that is part of a separate business process.
- The client data model project needs to work with data from a pluggable, external service.

In the first case, you provide both sides of the service. In the second case, you may not know what the external service looks like and you may need to perform the following:

1. Even though you might not need a Fusion implementation of the service, since the service-enabled application module's service interface is the supported unit of pluggability and the supported way of creating service-backed entity objects and view objects, you create an application module with a service interface that describes the shape you want your "canonical" pluggable service to have.
2. After generating the service-enabled application module, as described in How to Enable the Application Module Service Interface, you then build service-back entity objects and view objects.
3. Finally, you can create an EJB session bean (or an SCA composite) that supports the same service interface as the "canonical" service-enabled application module

that you created in Step 2 and configure the `connections.xml` file of the client project containing the service-enabled business components based on this service interface to use this "plugged" version instead.

For details about how to expose an application module as a web service, see Publishing Service-Enabled Application Modules.

Creating Entity Objects Backed by SDO Services

You create the service-backed entity object using the Create Entity Object wizard by supplying the URL for the WSDL document that describes the deployed service already running on an application server. You may locate the WSDL either from a UDDI registry or, when the URL endpoint is not accessible, from the WSDL document file that you have downloaded. The wizard uses the WSDL service description to display the list of available service view instances. In the wizard, you select among the displayed view instances to specify the entity object's data source. At the time you run the wizard, the service endpoint must be accessible in order to locate the WSDL document.

Only unsecured (HTTP) service endpoints may be used to create the service-backed entity object; the Create Entity Object wizard displays an error message when you attempt to access a secured (HTTPS) service endpoint.

Before you begin:

It may be helpful to have an understanding of how to access ADF Business Components services in the client Fusion web application. For more information, see Accessing Remote Data Over the Service-Enabled Application Module.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Service-Enabled Application Modules.

You will need to complete these tasks:

1. Obtain the URL for the WSDL document that describes the deployed service. If the unsecured HTTP URL is not accessible, download the WSDL file for the service and use the downloaded WSDL to obtain the service from the local file system.
2. Expose an application module as a web service, as described in How to Enable the Application Module Service Interface.
3. Service-enable specific view object as needed, as described in How to Service-Enable Individual View Objects.
4. Define a complex type (one that includes currency codes or a unit of measure) for attributes of the service-enabled view object as needed, as described in Associating Related SDO Properties Using Complex Data Types.

To create the entity object that uses a service view instance as its data source:

1. In the Applications window, right-click the project in which you want to create the entity object and choose **New** and then **Entity Object**.
2. On the Name page of the Create Entity Object wizard, do the following to create the entity object:
   a. Enter the package name in which the entity object will be created and enter the entity object name.

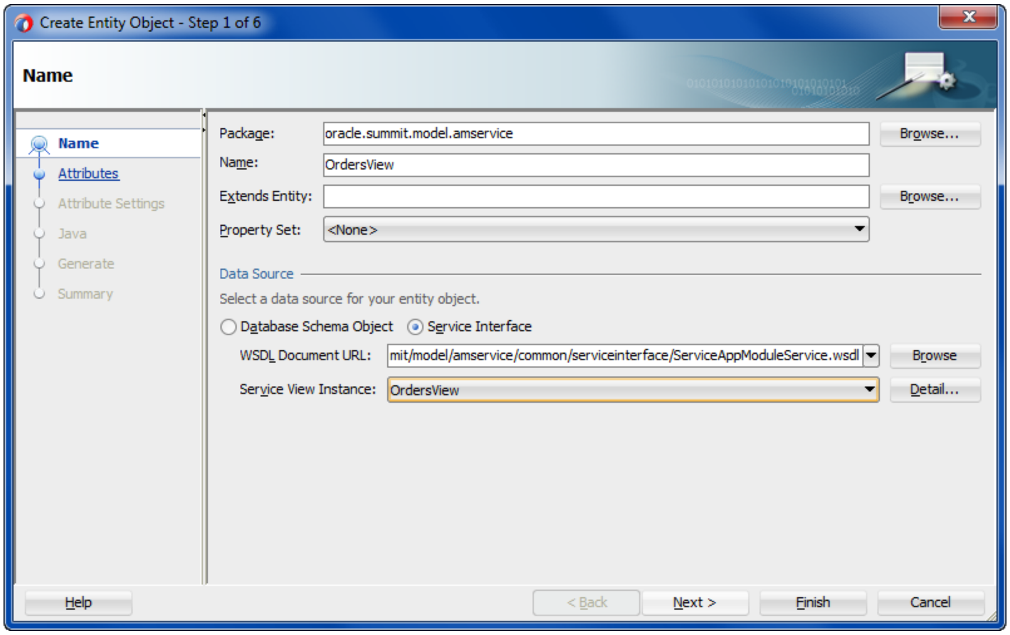
b. Select **Service Interface** as the data source for which you want to create the entity object.

c. Enter the WSDL document URL for the published web service, or click **Browse** and choose **UDDI Registry** to use the Find Web Services wizard to locate the web service from a UDDI registry. If the URL endpoint is not accessible, copy the WSDL file to the local file system and then click **Browse** and choose **File System** to navigate to the file.

   JDeveloper will attempt to connect with the service endpoint and populate the list of service view instances from the WSDL service description. If the endpoint is unavailable, the list will remain empty.

d. From the **Service View Instance** dropdown, choose the appropriate service view instance as the data source, as shown in Figure 15-15.

Figure 15-15 Service Interface as Data Source in the Create Entity Object Wizard



Create Entity Object - Step 1 of 6

Name

Package: oracle.summit.model.amservice

Name: OrdersView

Extends Entity:

Property Set: <None>

Data Source

Select a data source for your entity object.

○ Database Schema Object ● Service Interface

WSDL Document URL: mit/model/amservice/common/serviceinterface/ServiceAppModuleService.wsdl

Service View Instance: OrdersView

3. Click **Next** and modify the attributes settings of the entity object before you complete the wizard.

   For example, on the Attributes Settings page, you can enable the **Refresh After Insert** and **Refresh After Update** options for attributes that you anticipate will be modified whenever the entity is modified. Typical candidates include a version number column or an updated date column in the row.

4. Click **Finish**.

Using Complex Data Types with Service-Backed Entity Object Attributes

As a result of creating a service-backed entity object, JDeveloper automatically exposes attributes that were defined by the SDO properties of the base service-enabled view object. When your entity object contains attributes with complex types, you will need to select the complex type's related attribute from the entity object. For example, suppose that your service-backed entity object defines the `OrderTotal` attribute and a `CurrencyCode` attribute to specify the currency code of allowed countries. You will need to map the related attribute CurrencyCode to the SDO property type specified by the service-enabled view object. Complex types support these service types:

- AmountType service type, for use with any property that defines a currency code
- MeasureType service type, for use with any property that defines a unit of measure

You use the Attributes page of the overview editor to select the entity object attribute defined as a complex type. You use the Details tab that you display from the Attributes page of the overview editor to map the complex type of the selected attribute to a related attribute of the appropriate type.

Before you begin:

It may be helpful to have an understanding of how to access ADF Business Components services in the client Fusion web application. For more information, see Accessing Remote Data Over the Service-Enabled Application Module.

It may be helpful to have an understanding of complex types and SDO properties. For more information, see Associating Related SDO Properties Using Complex Data Types.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Service-Enabled Application Modules.

You will need to complete these tasks:

1. Expose an application module as a web service, as described in How to Enable the Application Module Service Interface.
2. Service-enable the desired view object, as described in How to Service-Enable Individual View Objects.
3. Specify a complex type for individual attributes of the service-enabled view object, as described in Associating Related SDO Properties Using Complex Data Types.
4. Create the service-backed entity object from the service-enabled view object, as described in Creating Entity Objects Backed by SDO Services.

To associate attributes using a complex type in the service-backed entity object:

1. In the Applications window, double-click the service-backed entity object.
2. In the overview editor, click the **Attributes** navigation tab.
3. In the Attributes page, select the attribute backed by an attribute defined as a complex type in the service-enabled view object.

   The attribute you select will be defined as a numeric type by the SDO property of the service-enabled view object.
4. With the attribute selected, click the **Details** tab and then in the **Service** section, in the **currencyCode** or **unitCode** dropdown list, select the entity object attribute that you want to associate with the complex type.

   The dropdown list displays all String attributes that the entity object defines. Select the attribute that is appropriate to map as the related attribute in the complex type definition. For example, to associate a currency code with the OrderTotal attribute that displays the amount paid by a customer, you might select the CurrencyCode attribute in the Orders service-backed entity object.

The SDO framework supports related service attribute values **currencyCode** and **unitCode**. When the editor displays **currencyCode**, the attribute you associate must specify currency information. When the editor displays **unitCode**, the attribute you associate must specify a unit of measure.

Creating View Objects Backed by SDO Services

After you add the service-backed entity object to your project, you can create service-backed view objects to query and optionally filter the data from the web service for use in the user interface. A service-backed view object is a view object whose single entity usage references an entity object that is backed an SDO service. You cannot make existing view objects service-backed. Instead, when you create the view object, the new view object will automatically be service-backed if its entity usage is a service-backed entity object.

Before you begin:

It may be helpful to have an understanding of how to access ADF Business Components services in the client Fusion web application. For more information, see Accessing Remote Data Over the Service-Enabled Application Module.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Service-Enabled Application Modules.

You will need to complete this task:

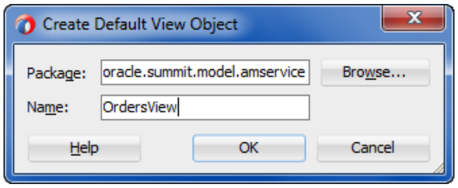
Create the service-backed entity object, as described in Creating Entity Objects Backed by SDO Services.

To create a view object from the service-backed entity object:

1. In the Applications window, right-click the service-backed entity object and choose **New Default View Object**.
2. In the Create Default View Object dialog, enter the package name in which the view object will be created and enter the view object name, as shown in Figure 15-16.

   The generated view object will contain the same attributes as the entity object. You can optionally edit the view object in the overview editor to customize the query. You can also define view criteria for the view object when you want to filter the data from the web service. For details about filtering query results, see Working with Named View Criteria.

Figure 15-16 View Object Can Be Created from Service-Backed Entity Object



Package: oracle.summit.model.amservice
Name: OrdersView

What Happens When You Create Service-Backed Business Components

The service-backed entity object is an entity object that encapsulates the details of accessing and, if necessary, modifying a row of data from an ADF Business Components web service. After you use the Create Entity Object wizard to create the service-backed entity object, JDeveloper saves additional service-related metadata in the `<Datasource>` element of the entity component definition. The entity component definition includes all the attributes that you selected from the service-enabled view object instance, including attributes with a complex type definition, as shown in the following example.

The service-backed view object references the single, service-backed entity object in its metadata just as any entity-based view object does. You can use the service-backed view object just as you would use any other view object. For details about working with view objects, see Defining SQL Queries Using View Objects. The ADF runtime handles the interaction with the ADF Business Components web service.

```
<Entity
   xmlns="http://xmlns.example.com/bc4j"
   Name="Customer_ServiceBasedEO"
   InheritPersnalization="true"
   AliasName="CustomerSEO"
   BindingStyle="OracleName"
   UseGlueCode="false">
   <DataSource
      DataSourceClass="oracle.jbo.datasource.svc.SIEODataSourceImpl"
      Type="ServiceInterface">
      <ServiceInterface
         ServiceName="{http://www.globalcompany.com/oesvc/}OrderEntryService"
         SDOName="{http://www.globalcompany.com/oesvc/}CustomersSVO"
         SVIName="{http://www.globalcompany.com/oesvc/}CustomersSVO"
         CreateOpName="createCustomer"
         UpdateOpName="updateCustomer"
         DeleteOpName="deleteCustomer"
         GetOpName="getCustomer"
         FindOpName="findCustomers"
         ProcessOpName="processCustomers"/>
   </DataSource>
   <Attribute
      Name="CustomerId"
      ColumnName="CustomerId"
      SQLType="NUMERIC"
      Type="oracle.jbo.domain.Number"
      ColumnType="NUMBER"
      PrimaryKey="true"/>
   <!-- ... Attribute that is associated with complex type attribute ... -->
   <Attribute
      Name="CurrencyCode"
      Precision="255"
      ColumnName="CurrencyCode"
      SQLType="VARCHAR"
      Type="java.lang.String"
      ColumnType="VARCHAR2"/>
   <!-- ... Attribute with complex type mapping ... -->
   <Attribute
      Name="OrderTotal"
```

```
        ColumnName="OrdTotal"
        SQLType="NUMERIC"
        Type="java.math.BigDecimal"
        ColumnType="NUMBER"
        <Properties>
           <SchemaBasedProperties>
             <DomainAttrMappings>
               <DomainAttrMapping
                 MappedAttrName="CurrencyCode"
                 Name="currencyCode"/>
             </DomainAttrMappings>
          </SchemaBasedProperties>
        </Properties>
     </Attribute>
     <!-- ... Other Attribute elements here ... -->
  </Entity>
```

How to Update the Data Model for Service-Backed Business Components

Because the service interface exposes individual view instances, you are responsible for defining hierarchical relationships between service-backed entity objects (through associations) and service-backed view objects (through view links) in your consuming project. View links and associations are not automatically created when you create the service-backed business component. For example, if the application module of the published ADF Business Components service defines a master-detail relationship that you want to utilize, then you must define a view link for the corresponding view objects in your own project to preserve this hierarchy.

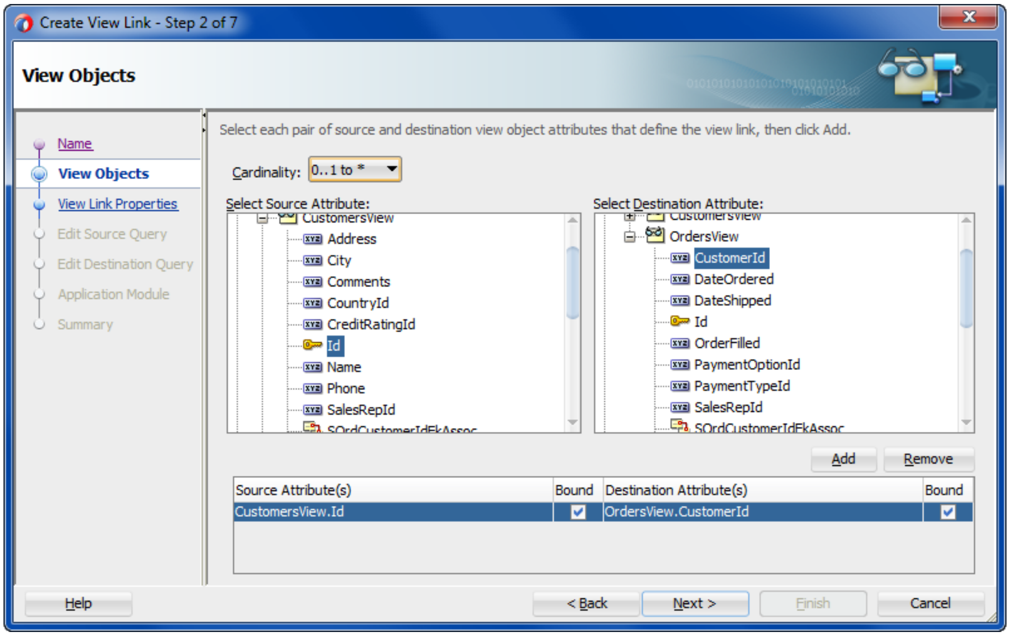
Furthermore, while you can create view links between view objects that query their data locally and service-backed view objects (and the other way around), once you define the view link, you will not be able to create entity-based view objects with the following entity object usages:

- The view object will not be able to reference a secondary entity usage that is a service-backed entity object.
- The view object will not be able to reference a primary entity usage that is a service-backed entity object with secondary entity usages.

The same restrictions apply to associations in the client project between regular entity objects and service-backed entity objects: while you can create the associations, you will not be able to create view objects.

You use the Create View Link wizard to specify relationships between the view objects that your project defines, as shown in Figure 15-17. For details about creating view links, see How to Create a Master-Detail Hierarchy Based on Entity Associations.

Figure 15-17   One to Many Relationship Defined in Create View Link Wizard



Select each pair of source and destination view object attributes that define the view link, then click Add.

Cardinality: 0..1 to *

Source Attribute(s)	Bound	Destination Attribute(s)	Bound
CustomersView.Id	☑	OrdersView.CustomerId	☑

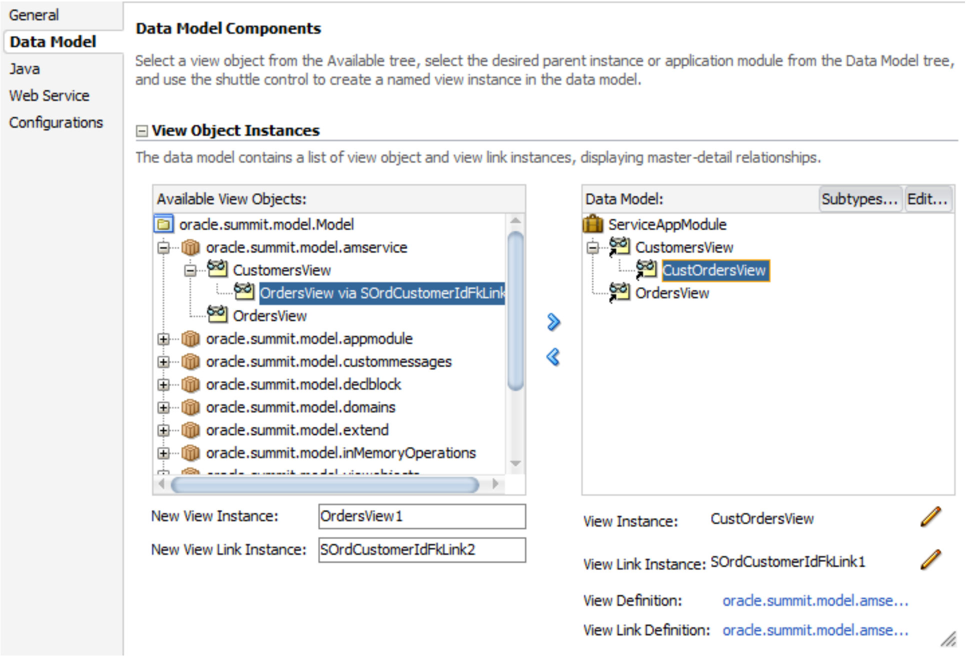
View links you create may define relationships between service-backed view objects and view objects that query locally accessed database tables. For example, you might choose to drive a database-derived detail view object with a service-backed master view object. You can create view links with the combinations shown in Table 15-3.

Table 15-3   Supported View Link Combinations Involving Service-Backed View Objects

Use Case	Master View Object Type	View Linked Detail View Object Type	View Link Cardinality
Local master rows with remote details	Query-based	Service-backed	One-to-many
Remote master rows with local details	Service-backed	Query-based	One-to-many
Local master rows with remote reference information	Query-based	Service-backed	Many-to-one
Remote master rows with local reference information	Service-backed	Query-based	Many-to-one

Once you have defined the desired view hierarchy, using the Create View Link wizard, you use the overview editor for your project's application module to define new view instances on the data model, as shown in Figure 15-18. The updated data model allows you to expose the view objects as ADF data controls that enable databinding with the user interface components of the Fusion web application. For details about updating the data model, see Adding Master-Detail View Object Instances to an Application Module.

Figure 15-18 Data Model Contains Service View Instances



General
Data Model
Java
Web Service
Configurations

Data Model Components

Select a view object from the Available tree, select the desired parent instance or application module from the Data Model tree, and use the shuttle control to create a named view instance in the data model.

View Object Instances

The data model contains a list of view object and view link instances, displaying master-detail relationships.

Available View Objects:
oracle.summit.model.Model
oracle.summit.model.amservice
CustomersView
OrdersView via SOrdCustomerIdFkLink
OrdersView
oracle.summit.model.appmodule
oracle.summit.model.custommessages
oracle.summit.model.declblock
oracle.summit.model.domains
oracle.summit.model.extend
oracle.summit.model.inMemoryOperations
Data Model:
Subtypes...
Edit...
ServiceAppModule
CustomersView
CustOrdersView
OrdersView

New View Instance: OrdersView1

New View Link Instance: SOrdCustomerIdFkLink2

View Instance: CustOrdersView

View Link Instance: SOrdCustomerIdFkLink1

View Definition: oracle.summit.model.amse...

View Link Definition: oracle.summit.model.amse...

How to Configure the Service-Backed Business Components Runtime

Before you can run your application and interact with the published service-enabled ADF application module to invoke service operations, you need to describe the published service, including the service's endpoint provider type and other configuration information. The ADF Business Components `ServiceFactory` class (`oracle.jbo.client.svc.ServiceFactory`) returns a proxy for the service, then uses the service proxy to invoke the service operations. The service factory can return proxies for three different service endpoint providers, to support these transport protocols:

- When the service endpoint provider is ADF Business Component, the transport protocol is EJB RMI.
- When the service endpoint provider is SOA Fabric, the transport protocol is SOA Fabric SDO binding.
- When the service endpoint provider is SOAP (for JAX-WS clients), the transport protocol is SOAP.

To configure the consuming application to invoke published service operations:

1. Add the *`bcProfileName`*`_Common.jar` file for the SDO's generated classes to the client project's classpath.
2. Update the `connections.xml` file in the client project's `.adf/META-INF` folder to describe the published ADF Business Components service.

   The updates you make to the file will depend on the transport protocol your application uses: EJB RMI protocol, SOA Fabric SDO binding, or SOAP protocol (for JAX-WS clients).

Adding the SDO Client Library to the Classpath

Before your application can access the published service, the service consuming project must have access to the generated SDO classes and their schema definitions. These files are packaged in the `*bcProfileName*_Common.jar` file generated by the development team responsible for publishing the service.

To make the SDO classes and their schema definitions available to your application, obtain the `*bcProfileName*_Common.jar` file from the service-provider team and place this JAR file in a folder of your local project. For example, you may copy the JAR file into your project's `deploy` folder. You can then use JDeveloper to add the JAR file to your project's classpath with a SDO client library you create. For steps to generate the SDO classes JAR file, see How to Deploy Web Services to Oracle WebLogic Server.

Before you begin:

It may be helpful to have an understanding of how to access ADF Business Components services in the client Fusion web application. For more information, see Accessing Remote Data Over the Service-Enabled Application Module.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Service-Enabled Application Modules.

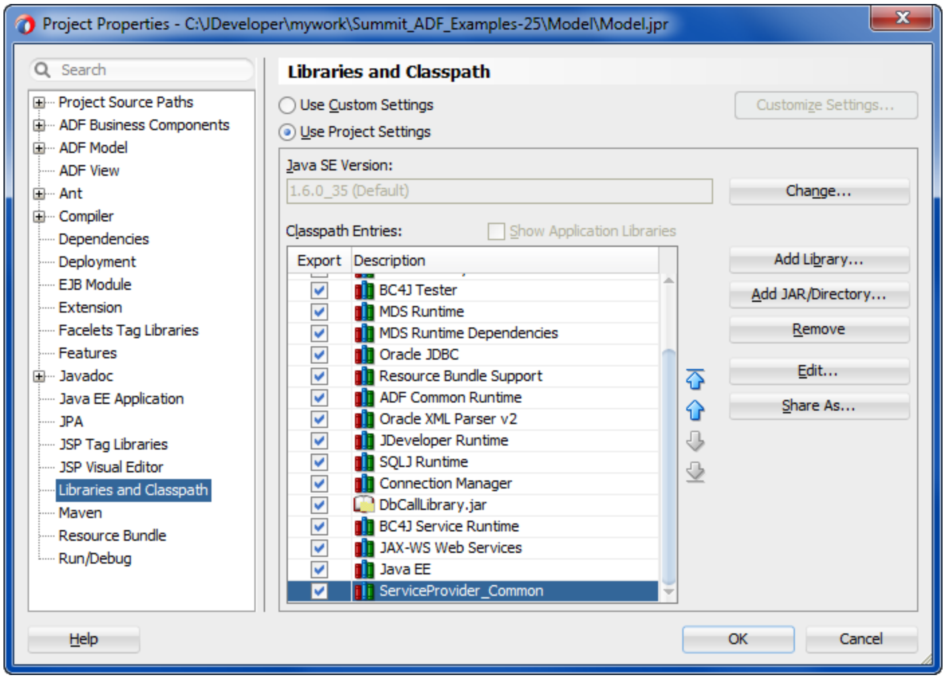
To add the SDO client library to the classpath:

1. In the Applications window, right-click the project that contains the SDO classes and choose **Project Properties**.
2. In the Project Properties dialog, select **Libraries and Classpath** and click **Add Libraries**.
3. In the Add Library dialog, click **New** to create the SDO client library.
4. In the Create Library dialog, click **Add Entry** to add a classpath entry.
5. In the Select Path Entry dialog, browse to the folder that contains the `*bcProfileName*_Common.jar` file and select the file to view it in the Create Library dialog.

   The Select Path Entry dialog lets you browse the file system or local area network to locate the JAR file. If you cannot browse the `deploy` folder of the service-provider's application workspace to obtain the JAR file, you must obtain the file and copy it into your own project's folder. For example, you may have copied the JAR file into your project's `deploy` folder.
6. Click **OK** in the dialogs to display the Project Properties dialog with the SDO client library selected. Click **OK** to add the library to the classpath.

   Figure 15-19 shows the SDO client library with the name `ServiceProvider_Common.jar` selected. In this case, the library name is the same as the JAR file name. Optionally, you can edit the library name in the Create Library dialog.

Figure 15-19   SDO Client Library Classpath Entry



Libraries and Classpath

○ Use Custom Settings
● Use Project Settings

Java SE Version: 1.6.0_35 (Default)

Classpath Entries: □ Show Application Libraries

Export	Description
☑	BC4J Tester
☑	MDS Runtime
☑	MDS Runtime Dependencies
☑	Oracle JDBC
☑	Resource Bundle Support
☑	ADF Common Runtime
☑	Oracle XML Parser v2
☑	JDeveloper Runtime
☑	SQLJ Runtime
☑	Connection Manager
☑	DbCallLibrary.jar
☑	BC4J Service Runtime
☑	JAX-WS Web Services
☑	Java EE
☑	ServiceProvider_Common

Registering the ADF Business Components Service in the Consuming Application's connections.xml for the EJB RMI Protocol

When the service endpoint provider is ADF Business Components, the service factory will return an EJB object proxy bound to a stateless session bean running in the EJB container. You must provide the JNDI context information to allow the consuming application to look up the published service.

Lookup information that you provide to register the published ADF Business Components service appears in the consuming Fusion web application's `connections.xml` file, located in the `.adf/META-INF` folder relative to the application. The ADF connection architecture uses this file to encapsulate the details of the service endpoint provider.

The JNDI lookup information you provide will depend on whether the published service runs locally (in the same JVM) with the consuming application or runs remotely on a separate server from the consuming application. Typically, the ADF Business Components service is in a different application from the consuming application and is therefore run remotely.

To register the published service with your client application, update the `connections.xml` file, as shown in the following example. When the ADF Business Components service runs local to the consuming application (as occurs when you run within JDeveloper), the service factory needs only the JNDI name to look up the service.

> **Note:**
>
> When you deploy the calling application to standalone Oracle WebLogic Server, you will use Oracle Enterprise Manager to configure the JNDI context properties instead of editing the `connections.xml` file. For instructions, refer to the online documentation in Oracle Enterprise Manager.

```
<References xmlns="http://xmlns.oracle.com/adf/jndi">
  <Reference name="{www.globalcompany.com}ServiceAppModuleService"
             className="oracle.jbo.client.svc.Service" xmlns="">
     <Factory className="oracle.jbo.client.svc.ServiceFactory"/>
     <RefAddresses>
       <StringRefAddr addrType="serviceInterfaceName">
        <Contents>oracle.summit.model.amservice.common.
                                            serviceinterface.ServiceAppModuleService</Contents>
       </StringRefAddr>
       <StringRefAddr addrType="serviceEndpointProvider">
          <Contents>ADFBC</Contents>
       </StringRefAddr>
       <StringRefAddr addrType="jndiName">
          <Contents>ServiceAppModuleServiceBean#oracle.summit.model.amservice.common.
                                            serviceinterface.ServiceAppModuleService</Contents>
       </StringRefAddr>
       <StringRefAddr addrType="serviceSchemaName">
          <Contents>ServiceAppModuleService.xsd</Contents>
       </StringRefAddr>
       <StringRefAddr addrType="serviceSchemaLocation">
          <Contents>oracle/summit/model/amservice/common/serviceinterface/</Contents>
       </StringRefAddr>
     </RefAddresses>
  </Reference>
  ...
</References>
```

When the ADF Business Components service runs remotely to the calling client, then remote JNDI context information needs to be added to the `connections.xml` file. You can edit these JNDI context properties in the `connections.xml` file, as shown the example:

- `jndiFactoryInitial` should be set to `weblogic.jndi.WLInitialContextFactory`.
- `jndiProviderURL` is the JNDI provider URL that indicates the location of the JNDI server. The URL should be composed as `t3://`*`<hostname>`*`:`*`<server port>`*.

  For example, specify a URL like: `t3://localhost:8888`, where `t3` is the Oracle WebLogic protocol, `localhost` is the host name that the remote Oracle WebLogic Server instance runs in, `8888` is the port number.
- `jndiSecurityPrincipal` specifies the principal (user name) with permission to access the remote JNDI.

  When you deploy the service to standalone Oracle WebLogic Server, the user name can be read from the file.
- `jndiSecurityCredentials` specifies the credentials (password) to be used for the security principal.

When you deploy the service to standalone Oracle WebLogic Server in a test environment, you can specify credentials in plain text for the JNDI provider. For example, you can specify `weblogic/weblogic1`, which are the default administrator user name/password credentials with sufficient privileges to access JNDI provider for Oracle WebLogic Server.

When you deploy the service to a production environment, you must remove the plain text password to avoid creating a security vulnerability. As the following example shows, the `connections.xml` file must contain `<SecureRefAddr addrType="jndiSecurityCredentials"/>` with no password. To configure the service password for standalone Oracle WebLogic Server, you must use Oracle Enterprise Manager, which will store the encrypted password in Oracle's credential store.

```
<References xmlns="http://xmlns.oracle.com/adf/jndi">
  <Reference name="{www.globalcompany.com}ServiceAppModuleService"
             className="oracle.jbo.client.svc.Service" xmlns="">
     <Factory className="oracle.jbo.client.svc.ServiceFactory"/>
     <RefAddresses>
       <StringRefAddr addrType="serviceInterfaceName">
        <Contents>oracle.summit.model.amservice.common.
                                                serviceinterface.ServiceAppModuleService</Cont
       </StringRefAddr>
       <StringRefAddr addrType="serviceEndpointProvider">
          <Contents>ADFBC</Contents>
       </StringRefAddr>
       <StringRefAddr addrType="jndiName">
          <Contents>ServiceAppModuleServiceBean#oracle.summit.model.amservice.common.
                                                serviceinterface.ServiceAppModuleService</Cont
       </StringRefAddr>
       <StringRefAddr addrType="jndiFactoryInitial">
          <Contents>weblogic.jndi.WLInitialContextFactory</Contents>
       </StringRefAddr>
       <StringRefAddr addrType="jndiProviderURL">
          <Contents>t3://localhost:8888</Contents>
       </StringRefAddr>
       <StringRefAddr addrType="jndiSecurityPrincipal">
          <Contents>a_username</Contents>
       </StringRefAddr>
       <SecureRefAddr addrType="jndiSecurityCredentials"/>
       <StringRefAddr addrType="serviceSchemaName">
          <Contents>ServiceAppModuleService.xsd</Contents>
       </StringRefAddr>
       <StringRefAddr addrType="serviceSchemaLocation">
          <Contents>oracle/summit/model/amservice/common/serviceinterface/</Contents>
       </StringRefAddr>
     </RefAddresses>
  </Reference>
  ...
</References>
```

Registering the ADF Business Components Service in the Consuming Application's connections.xml for the SOAP Protocol

When the service endpoint provider is SOAP, the service factory will create a dynamic JAX-WS client proxy. You must provide the WSDL URL and port name to allow the consuming application to look up the published service. Additionally, for the SOAP client, OWSM client security policy can be attached as part of the SOAP header.

Lookup information that you provide to register the published ADF Business Components service appears in the consuming Fusion web application's `connections.xml` file, located in the `.adf/META-INF` folder relative to the application. The ADF connection architecture uses this file to encapsulate the details of the service endpoint provider.

> **Note:**
>
> When you deploy the calling application to standalone Oracle WebLogic Server, you will use Oracle Enterprise Manager to configure the JNDI context properties instead of editing the `connections.xml` file. For instructions, refer to the online documentation in Oracle Enterprise Manager.

To register the published service with your client application for the SOAP protocol, depending on whether your application uses identity propagation or identity switching, update the `connections.xml` file following the examples below. Identity propagation and switching are similar in that each process involves propagating an identity. In Fusion web applications, identity propagation involves propagating the identity that is currently executing code. Identity switching, on the other hand, involves propagating an application identity that is different from that currently executing code.

To register the published service with your client application so the user identity will be switched based on the credential key, specify the clientside policy `oracle/wss11_username_token_with_message_protection_client_policy` in the `connections.xml` file as the following example illustrates.

> **Note:**
>
> The `connections.xml` file supports OWSM security policy client overrides. When the security policy is `oracle/wss11_username_token_with_message_protection_client_policy`, the `csf-key` property can be overridden to specify the consuming application's credentials.

```
<Reference name="{http://xmlns.oracle.com/apps/sample/hrService/}HrService"
className="oracle.jbo.client.svc.Service" xmlns="">
   <Factory className="oracle.jbo.client.svc.ServiceFactory"/>
   <RefAddresses>
      <StringRefAddr addrType="serviceInterfaceName">
            <Contents>oracle.apps.sample.hrService.HrService</Contents>
      </StringRefAddr>
      <StringRefAddr addrType="serviceEndpointProvider">
         <Contents>SOAP</Contents>
      </StringRefAddr>
      <StringRefAddr addrType="webServiceConnectionName">
         <Contents>HrServiceConnection</Contents>
      </StringRefAddr>
      <StringRefAddr addrType="serviceSchemaName">
         <Contents>HrService.xsd</Contents>
      </StringRefAddr>
      <StringRefAddr addrType="serviceSchemaLocation">
         <Contents>oracle/apps/sample/hrService/</Contents>
      </StringRefAddr>
   </RefAddresses>
</Reference>
<Reference name="HrServiceConnection"

className="oracle.adf.model.connection.webservice.impl.WebServiceConnectionImpl"
xmlns="">
   <Factory

className="oracle.adf.model.connection.webservice.api.WebServiceConnectionFactory
"/>
   <RefAddresses>
      <XmlRefAddr addrType="WebServiceConnection">
         <Contents>
            <wsconnection
               description="http://rws65094fwks:7202/MySampleSvc/HrService?WSDL"
               service="{http://xmlns.oracle.com/apps/sample/
hrService/}HrService">
               <model
                  name="{http://xmlns.oracle.com/apps/sample/
hrService/}HrService"
                  xmlns="http://oracle.com/ws/model">
                  <service name="{http://xmlns.oracle.com/apps/sample/
hrService/}HrService">
                     <port name="HrServiceSoapHttpPort"
                        binding="{http://xmlns.oracle.com/apps/sample/
hrService/}HrServiceSoapHttp"
                        portType="http://xmlns.oracle.com/apps/sample/
hrService/}HrService">
                        <call-properties xmlns="http://oracle.com/adf">
                           <call-property id="csf-key" xmlns="">
                               <name>csf-key</name>
                               <value>meuser.credentials</value>
                            </call-property>
                        </call-properties>
                        <policy-references xmlns="http://oracle.com/adf">
                           <policy-reference category="security"
                            uri="oracle/
wss11_username_token_with_message_protection_client_policy"
                            enabled="true"
                            id="oracle/
wss11_username_token_with_message_protection_client_policy"
                            xmlns=""/>
                        </policy-references>
                        <soapaddressUrl="http://rws65094fwks:7202/MySampleSvc/
HrService"
                            xmlns="http://schemas.xmlsoap.org/wsdl/soap/"/>
                     </port>
                  </service>
               </model>
            </wsconnection>
         </Contents>
      </XmlRefAddr>
   </RefAddresses>
   </Reference>
```

To register the published service with your client application so the user identity will be propagated to the caller, specify the clientside policy `oracle/wss11_saml_token_with_message_protection_client_policy` in the `connections.xml` file, as the following example illustrates.

```
<Reference name="{http://xmlns.oracle.com/apps/sample/hrService/}HrService"
className="oracle.jbo.client.svc.Service" xmlns="">
   <Factory className="oracle.jbo.client.svc.ServiceFactory"/>
   <RefAddresses>
      <StringRefAddr addrType="serviceInterfaceName">
            <Contents>oracle.apps.sample.hrService.HrService</Contents>
      </StringRefAddr>
      <StringRefAddr addrType="serviceEndpointProvider">
         <Contents>SOAP</Contents>
      </StringRefAddr>
      <StringRefAddr addrType="webServiceConnectionName">
         <Contents>HrServiceConnection</Contents>
      </StringRefAddr>
      <StringRefAddr addrType="serviceSchemaName">
         <Contents>HrService.xsd</Contents>
      </StringRefAddr>
      <StringRefAddr addrType="serviceSchemaLocation">
         <Contents>oracle/apps/sample/hrService/</Contents>
      </StringRefAddr>
   </RefAddresses>
</Reference>
<Reference name="HrServiceConnection"

className="oracle.adf.model.connection.webservice.impl.WebServiceConnectionImpl"
xmlns="">
   <Factory

className="oracle.adf.model.connection.webservice.api.WebServiceConnectionFactory
"/>
   <RefAddresses>
      <XmlRefAddr addrType="WebServiceConnection">
         <Contents>
            <wsconnection
               description="http://rws65094fwks:7202/MySampleSvc/HrService?WSDL"
               service="{http://xmlns.oracle.com/apps/sample/
hrService/}HrService">
               <model
                  name="{http://xmlns.oracle.com/apps/sample/
hrService/}HrService"
                  xmlns="http://oracle.com/ws/model">
                  <service
                     name="{http://xmlns.oracle.com/apps/sample/
hrService/}HrService">
                     <port name="HrServiceSoapHttpPort"
                        binding="{http://xmlns.oracle.com/apps/sample/
hrService/}HrServiceSoapHttp"
                        portType="http://xmlns.oracle.com/apps/sample/
hrService/}HrService">
                        <policy-references xmlns="http://oracle.com/adf">
                           <policy-reference category="security"
                                uri="oracle/
wss11_saml_token_with_message_protection_client_policy"
                                enabled="true"
                                id="oracle/
wss11_saml_token_with_message_protection_client_policy"
                                xmlns=""/>
                        </policy-references>
                        <soap addressUrl="http://rws65094fwks:7202/MySampleSvc/
HrService"
                              xmlns="http://schemas.xmlsoap.org/wsdl/soap/"/>
                     </port>
                  </service>
               </model>
            </wsconnection>
         </Contents>
      </XmlRefAddr>
   </RefAddresses>
</Reference>
```

Registering the ADF Business Components Service in the Consuming Application's connections.xml for Fabric SDO Binding

When the service endpoint provider is Fabric, the service factory will return a SOA Fabric composite proxy and call the service running inside a Fabric composite through Fabric's SDO binding. You must provide the name of the Fabric composite to allow the consuming application to look up the published service.

Lookup information that you provide to register the published ADF Business Components service appears in the consuming Fusion web application's `connections.xml` file, located in the `.adf/META-INF` folder relative to the application. The ADF connection architecture uses this file to encapsulate the details of the service endpoint provider.

> **Note:**
>
> When you deploy the calling application to standalone Oracle WebLogic Server, you will use Oracle Enterprise Manager to configure the JNDI context properties instead of editing the `connections.xml` file. For instructions, refer to the online documentation in Oracle Enterprise Manager.

To register the published service with your client application for the Fabric protocol, update the `connections.xml` file, as the following example shows, where `fabricAddress` is the name of the Fabric composite for the published service.

```
<References xmlns="http://xmlns.oracle.com/adf/jndi">
  <Reference name="{www.globalcompany.com}ServiceAppModuleService"
             className="oracle.jbo.client.svc.Service" xmlns="">
     <Factory className="oracle.jbo.client.svc.ServiceFactory"/>
     <RefAddresses>
       <StringRefAddr addrType="serviceInterfaceName">
        <Contents>oracle.summit.model.amservice.common.
                                             serviceinterface.ServiceAppModuleService</Content
       </StringRefAddr>
       <StringRefAddr addrType="serviceEndpointProvider">
          <Contents>Fabric</Contents>
       </StringRefAddr>
       <StringRefAddr addrType="fabricAddress">
         <Contents>fabric_ServiceAppModuleService</Contents>
       <StringRefAddr addrType="serviceSchemaName">
          <Contents>ServiceAppModuleService.xsd</Contents>
       </StringRefAddr>
       <StringRefAddr addrType="serviceSchemaLocation">
          <Contents>oracle/summit/model/amservice/common/serviceinterface/</Contents>
       </StringRefAddr>
     </RefAddresses>
  </Reference>
  ...
</References>
```

How to Test the Service-Backed Components in the Oracle ADF Model Tester

Before you can launch the Oracle ADF Model Tester, your project must meet the runtime requirements, as described in How to Configure the Service-Backed Business Components Runtime. The Oracle ADF Model Tester will display the view objects you create from the remote web service and allow you to interact with the service to perform standard CRUD operations.

Because the application module that you run can access locally queried data and remotely queried data together, service-backed view objects and database-derived view objects will display in the same tester. If the endpoint is unavailable at the time you select the service-backed view object in the Oracle ADF Model Tester, you will get a runtime exception.

For details about running the Oracle ADF Model Tester, see How to Run the Oracle ADF Model Tester Using Configurations.

How to Invoke Operations of the Service-Backed Components in the Consuming Application

The ADF Business Components service interface requires that you return a service proxy to ensure that operations you invoke use the transport protocol specified by the published service.

Before you begin:

It may be helpful to have an understanding of how to access ADF Business Components services in the client Fusion web application. For more information, see Accessing Remote Data Over the Service-Enabled Application Module.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Service-Enabled Application Modules.

You will need to complete this task:

- Ensure that the consuming application has the correct libraries on the classpath. In the Applications window, double-click the project and in the Project Properties dialog, select **Libraries and Classpath** and confirm the following libraries appear:
  - **Java EE 1.5**
  - **Oracle XML Parser v2**
  - **BC4J Service Client**
  - **JAX-WS Client**
  - The service's common JAR file

As the following example shows, when you invoke the operation, you perform the following tasks:

1. Import the `oracle.jbo.client.svc.ServiceFactory` class and published service class.
2. Call `getServiceProxy()` on the service factory object and pass in the service name in the form `<`*`serviceName`*`>.NAME`. The ADF service factory embeds a `SDOHelperContext` ID in the service proxy object returned by this method to ensure delivery of the latest ADF Business Component service schema metadata to the SDO.

   The schema (`.xsd` files) for the service object may be stored in MDS and may have been extended for example to add more business component attributes, extend existing types, or define new types. The local helper context allows customization of individual service's schema definitions without affecting other service's SDO metadata or requiring restarting the application.
3. Call `create()` on a data factory object, where the proxy object is obtained from the `getServiceProxy()` call.
4. Invoke the operation on the proxy object and return a data object.
5. Save the data object return as XML.

```
import commonj.sdo.DataObject;
import commonj.sdo.helper.DataFactory;
import commonj.sdo.helper.XMLHelper;
import hr.common.Dept;
import hr.common.serviceinterface.HRAppService;
import oracle.jbo.client.svc.ServiceFactory;
   ...
   {
      HRAppService proxy = (HRAppService)
ServiceFactory.getServiceProxy(HRAppService.NAME);

      Dept dept = (Dept)
               ServiceFactory.getDataFactory(proxy).create("http://example.com/hr/common/"
"Dept");
      dept.setDname("ENGINEERING");
      ...
      dept = proxy.createDept(dept);
      String xml = ServiceFactory.getXMLHelper(proxy).save((DataObject) dept,
                                   "http://example.com/hr/common/", "dept");
      out.print(xml);
   }
```

What You May Need to Know About Creating Service Data Objects in the Consuming Application

The ADF service interface framework defines a set of common data objects, such as `oracle.jbo.common.service.types.FindCriteria`, `oracle.jbo.common.service.types.FindControl`, `oracle.jbo.common.service.types.ProcessControl`, and so on. Those data objects are used in standard `findXxx()` method and `processXxx()` method calls. You can create these data objects in the consuming application and pass them to the standard service methods. As the following example shows, you can need to call the data factory class' `create()` method to construct those data objects before passing them into the `findXxx()` or `processXxx()` methods.

```
FindCriteria fc = (FindCriteria) ServiceFactory.getDataFactory(proxy).create
                       ("http://xmlns.oracle.com/adf/svc/types/",
"FindCriteria");
```

What You May Need to Know About Invoking Built-In Service Methods in the Consuming Application

The ADF service interface framework defines several built-in SOAP service methods that may be optionally enabled for the service view instances, including `getDftlObjAttrHints()`, `getServiceLastUpdateTime()`, and `getEntityList()`, as described in Table 15-2. These methods primarily support programmatic interaction with the service endpoint by the consuming application and are not supported at design time by data controls.

What Happens at Runtime: How the Application Accesses the Published Application Module

The ADF runtime obtains the data source information from the service-backed entity object XML definition to automate interactions with the service interface methods as needed. By using the service-backed entity object, you avoid having to work directly with the service proxy and service data objects programmatically for all common web service data access tasks. The ADF service factory looks up the service and then uses the service interface you specified in the `connections.xml` to invoke the service methods.

When your application accesses an ADF Business Components web service, each remote call is stateless, and the remote service will not participate in the same transaction as the business component that uses a service-enabled application module's service interface.

In the majority of the cases, calls to remote services will be informational in nature and will not make changes to remote objects. However, if you must use a remote service to make changes, then keep these points in mind:

- An exception thrown by the remote service will cause the local transaction to fail.
- If you successfully call a remote service that results in modifying data, and then subsequently your local transaction fails for any reason, then it is the responsibility of your error handling code to perform a compensating transaction against the remote service to "undo" the previous change made.

What You May Need to Know About Service-Backed Entity Objects and View Objects

You will use some web services to access reference information. However, other services you call may modify data. This data modification might be in your own company's database if the service was written by a member of your own or another team in your company. If the web service is outside your firewall, of course the database being modified will be managed by another company. In either of these situations, it is important to understand that any data modifications performed by a web service you invoke will occur in its own distinct transaction unrelated to the service-enabled application module's current unit of work. For example, if you have invoked a web service that modifies data and then you later call `rollback()` to cancel the pending changes in the application module's current unit of work, rolling back the changes has no effect on the changes performed by the web service you called in the process. You may need to invoke a corresponding web service method to perform a compensating change to account for your rollback of the application module's transaction.

At runtime, ADF handles the interaction with the ADF Business Components web service. However, you should be aware that service-backed business components have the following design time restrictions that may restrict your application's runtime behavior.

- View objects that you create cannot reference a service-backed entity object as a secondary entity object usage.
- View objects that you create cannot produce a flattened join from two or more related entity objects when at least one of those entity objects is a service-backed entity object.
- Service-backed view objects that you create from service-backed entity objects will not reference secondary entity usages.

For more details about how these restrictions apply at design time, see How to Update the Data Model for Service-Backed Business Components.

Accessing Polymorphic Collections in the Consuming Application

ADF Business Components allows Service Data Objects (SDOs) as an alternative approach to invoke polymorphic sub-type view object instances in the application module.

The ADF Model project may contain base view objects that define one or more subtype view objects derived from a common base view object. When the business components developer wants to expose the polymorphic usage of the base view object instance, they add the subtype to the ADF data model. However, when the developer creates the SDO service interface from application module view instances, the subtype SDO for the polymorphic subtype view objects are not directly referenced in the SDO service interface.

After generating the SDO interface from a ADF Model project that defines base view objects and subtype view objects, the generated XML schema (`.xsd`) will be strongly-typed for the base view object, but not for the polymorphic subtype. Therefore, web service clients that need to access polymorphic subtype view objects use an alternative approach to invoke exposed operations of the SDO interface.

> **Note:**
>
> The example in this section refers to the custom project `PolymorphicVO_WSClient` in the `SummitADF_Examples` application workspace. The client project access the SDO service interface created in the `oracle.summit.model.polymorphicvo` package in the Model project of the same workspace.

How to Generate Web Service Client Proxy Classes From the Service-Enabled Application Module

In JDeveloper, you can generate Java classes that act as a proxy to the remote service-enabled application module web service that you add to your web service client project. To accomplish this, you use the Create Web Service Client and Proxy wizard and supply a WSDL URL to the deployed service. The wizard creates a client file which uses the proxy classes and allows you to make Java method calls to access the web service.

Before you begin:

It may be helpful to have an understanding of how the SDO framework supports service-enabled ADF view objects and enables web service clients to access rows of data and perform service operations. For more information, see Publishing Service-Enabled Application Modules.

It may be helpful to understand how JDeveloper supports accessing web services in a Java applications by creating JAX-WS web service client and proxy classes. For more information, see the "Creating JAX-WS Web Services and Clients" section in *Developing Applications with Oracle JDeveloper*.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Service-Enabled Application Modules.

You will need to complete these tasks:

1. Create the desired subtype view objects and configure the polymorphic usages on the application module view instance using the subtypes, as described in Defining Polymorphic View Objects.
2. Enable the service interface for the application module, as described in How to Enable the Application Module Service Interface.
3. Create the client project that will contain the web service proxy classes that you will use to invoke service methods.

   Tip: If you have not already created a Java client project in JDeveloper, you create a custom project to contain the web service proxy classes, as described the "Creating Applications and Projects" section in *Developing Applications with Oracle JDeveloper*.
4. Run the web service in JDeveloper to obtain the service endpoint URL that you will use to specify the location of the WSDL document.

   Tip: You can run the service in Integrated WebLogic Server and obtain the service endpoint URL from the Log window, as described in How to Test the Web Service Using Integrated WebLogic Server.

To generate the service client and proxy files:

1. In the Applications window, select the client project that you want to contain the client proxy classes and choose **File > New Gallery** and from the **Categories** list, expand **Business Tier** and **Web Services**, and then select **Web Service Client and Proxy**.
2. In the Create Web Service Client and Proxy wizard, on the Select Web Service Description page, paste the service endpoint URL that you copy from Log window

into the **WSDL Document URL** field and append ?WSDL to the end of the URL, accept all defaults and click **Finish**.

For example, if the name of the ADF web service you deploy is ADFModuleService, the WSDL document URL with ?WSDL appended would like this:

```
http://<ipaddress>/ADFServiceDemo-ADFModel-context-root/
AppModuleService?WSDL
```

How to Invoke Operations of Polymorphic View Object Using Generated Proxy Classes

In JDeveloper, you can create a Java client that accesses web services through JAXB (Java Architecture for XML Binding), using generated classes from the web service schemas. To generate JAXB packages, proxy classes, and interfaces, the client developer runs a JAXB binding compiler against the ADF Business Components service schema (WSDL). The developer requires no knowledge of the service XML to work with the proxy classes.

Using the generated proxy classes, developers can access and manipulate polymorphic collections in method calls that you make in the Java class files of your client project. A sample of such a file is the generated SOAP service client Java class file. The generated client file is named for the deployed ADF web service that you supplied with the WSDL document URL. For example, if the name of the ADF web service you deploy is ADFModuleService, the SOAP service client Java class is named AppModuleServiceSoapHttpPortClient.java.

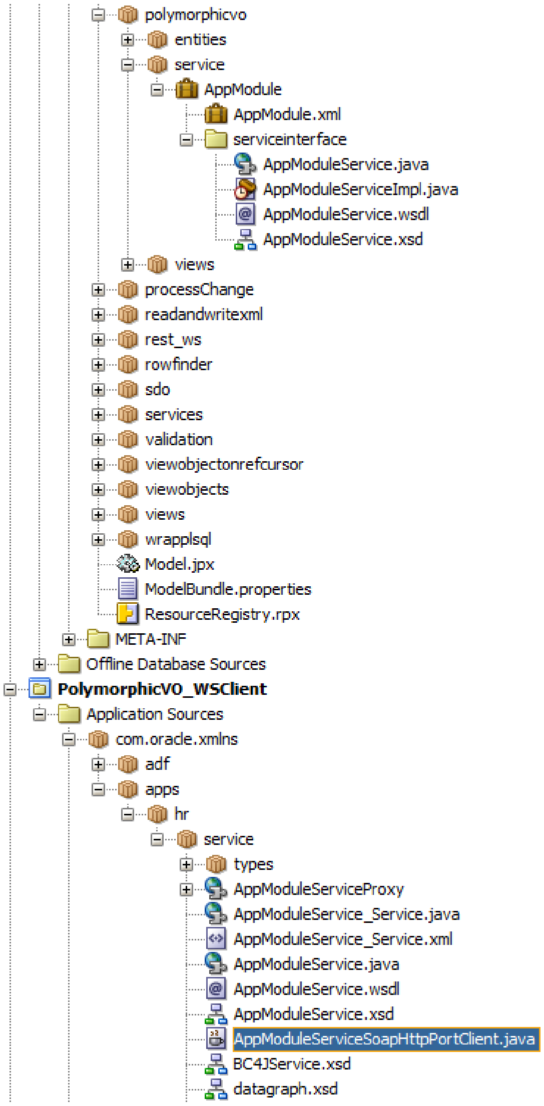
Before you begin:

It may be helpful to have an understanding of why polymorphic collections cannot be accessed as a remote service. For more information, see Accessing Polymorphic Collections in the Consuming Application.

You will need to complete these tasks:

- Create the Java client project that contains Java proxy classes for the service-enabled application module web service. For details, see How to Generate Web Service Client Proxy Classes From the Service-Enabled Application Module.
- In the Applications window, expand the client project and locate the SOAP service client file. As Figure 15-17 shows, the AppModuleServiceSoapHttpPortClient.java file is located in the com.oracle.xmlns package of the client project.

Figure 15-20   JAXB Generated SOAP Service Client File



You may proceed to edit the SOAP service client file to invoke the exposed operations of the base and subtype view objects. As the following example shows, factory objects are created for the subtype view object:

1. Create the collection using the factory method of the subtype view object.

   In this example, calling `createSalespersonViewExSDO()` on the service factory object creates an employee of subtype `Salesperson`.

2. Call the setters of the service factory object to set the required and optional attributes that the subtype's base view object defines.

   Note that optional attribute values must be cast as a special type `factory.create`*`ViewAttrName`* type, as described in What You May Need to Know About Invoking Service Methods in the Client Application.

3. Call the setters of the service factory object to set the required and optional attributes that the subtype view object defines.

In this example, the `Salesperson` subtype defines a single required attribute `CommissionPct` for the sales commission of sales department employees.

4. Invoke other operations on the collection.

   In this example, the `deleteSEmpView1()` deletes the created Salesperson row on the Employee collection and the `findSEmpView1ByDept()` finds the list of employees by their department.

```
package com.oracle.xmlns.apps.hr.service;

...

import oracle.summit.model.polymorphicvo.views.common.ObjectFactory;
import oracle.summit.model.polymorphicvo.views.common.SEmpViewSDO;

import oracle.summit.model.polymorphicvo.views.common.SalespersonViewExSDO;

...

public class AppModuleServiceSoapHttpPortClient {
    public static void main(String[] args) {
        AppModuleService_Service appModuleService_Service =
                                     new AppModuleService_Service();
        AppModuleService appModuleService =

appModuleService_Service.getAppModuleServiceSoapHttpPort();

        ...

        // Add your code to call the desired methods.
        try {
            SEmpViewSDO emp = testCreate(appModuleService);
            testFind(appModuleService);
            testDelete(appModuleService, emp);
            testFind(appModuleService);
        } catch (ServiceException e) {
        }

    }

...
    // 1. Create an employee of subtype Salesperson
    private static SEmpViewSDO testCreate(AppModuleService deptService) throws
                                ServiceException
    {
       oracle.summit.model.polymorphicvo.views.common.ObjectFactory factory = new
                oracle.summit.model.polymorphicvo.views.common.ObjectFactory();

       System.out.println("\n*** Testing createEmp ***\n");
       SalespersonViewExSDO salesperson = factory.createSalespersonViewExSDO();
       // 2. Set the attrs of the base SEmpView view object
       salesperson.setId(8001);
       salesperson.setFirstName(factory.createSEmpViewSDOFirstName("Lynn"));
       salesperson.setLastName("Munsinger");
       salesperson.setEmail(factory.createSEmpViewSDOEmail
                              ("lmunsing@summit.com"));
       salesperson.setTitleId(factory.createSEmpViewSDOTitleId
                              (new BigDecimal(2)));
       salesperson.setSalary(factory.createSEmpViewSDOSalary
                              (new BigDecimal(2000)));
       // 3. Set the attrs of the subtype SalespersonView view object
       salesperson.setCommissionPct
                            (factory.createSalespersonViewExSDOCommissionPct
                               (new BigDecimal(20)));
       salesperson.setDeptId(factory.createSEmpViewSDODeptId(31));

       return printEmp(deptService.createSEmpView1(salesperson));
    }

    // 4. Delete the employee of type Salesperson
    private static void testDelete(AppModuleService deptService, SEmpViewSDO emp)
                         throws ServiceException
    {
       System.out.println("\n*** Testing deleteEmp ***\n");
       deptService.deleteSEmpView1(emp);
    }

    // 5. Print
    private static SEmpViewSDO printEmp(SEmpViewSDO emp)
    {
       System.out.print(emp.getId() + "\t" + emp.getEmail().getValue() + "\t" +
             emp.getTitle().getValue() + "\t" +
                        "\t" + emp.getSalary().getValue());
       if (emp instanceof SalespersonViewExSDO)
       {
          System.out.print("\t" + ((SalespersonViewExSDO)
                             emp).getCommissionPct().getValue());
       }
       System.out.println();
       return emp;
    }

    // 6. Find employees by department
    private static void testFind(AppModuleService deptService) throws
                                                             ServiceException
    {
       com.oracle.xmlns.adf.svc.types.ObjectFactory factory = new
                        com.oracle.xmlns.adf.svc.types.ObjectFactory();

       System.out.println("\n*** Testing findEmpsByDept ***\n");
       FindCriteria fc = factory.createFindCriteria();
       fc.setFetchStart(0);
       fc.setFetchSize(-1);

       List<SEmpViewSDO> empList =
                          deptService.findSEmpView1ByDept(fc, "SALES", null);
       for (SEmpViewSDO emp : empList)
       {
          printEmp(emp);
       }
    }
}
```

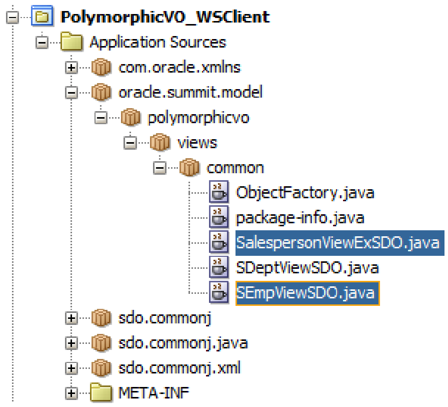
Running `AppModuleServiceSoapHttpPortClient.java` client in JDeveloper produces the following output to the Log window. The `createEmp` method creates an employee in the sales department as a `Salesperson` subtype of the `EmpView` collection. The attribute value `20` is the `Commission` attribute of the new polymorphic view row. The `findEmpsByDept` method displays a mix of `EmpView` base view rows and `SalespersonView` subtype rows. The remaining methods delete the created salesperson from the `EmpView` collection and print the result.

```
*** Testing createEmp ***

8001 lmunsing@summit.com    Sales Representative    2000   20

*** Testing findEmpsByDept ***

3     mnagayam@summit.com   VP, Operations          1400
11    cmagee@summit.com     Sales Representative    1400   10
13    ysedeghi@summit.com   Sales Representative    1515   10
14    mnguyen@summit.com    Sales Representative    1525   15
15    adumas@summit.com     Sales Representative    1450   17.5
23    rpatel@summit.com     Stock Clerk             795
8001  lmunsing@summit.com   Sales Representative    2000   20

*** Testing deleteEmp ***

*** Testing findEmpsByDept ***

3   mnagayam@summit.com   VP, Operations            400
11  cmagee@summit.com     Sales Representative      1400   10
13  ysedeghi@summit.com   Sales Representative      1515   10
14  mnguyen@summit.com    Sales Representative      1525   15
15  adumas@summit.com     Sales Representative      1450   17.5
23  rpatel@summit.com     Stock Clerk               795
```

What Happens When You Generate Java Proxy Classes With the SDO Schema

The JAXB binding compiler generates Java proxy classes for the application module service and object factory classes for the various SDO view instances exposed by the service interface WSDL. Because the polymorphic view object is defined in the type hierarchy of the base view object, it also has a strongly-typed proxy class. As Figure 15-21 shows, the `SalespersonViewSDO.java` proxy class appears along with the proxy class `SEmpViewSDO.java` for the base view object `SEmpView`.

Figure 15-21 JAXB Generated SOAP Service Client Proxy Classes



- PolymorphicVO_WSClient
  - Application Sources
    - com.oracle.xmlns
    - oracle.summit.model
      - polymorphicvo
        - views
          - common
            - ObjectFactory.java
            - package-info.java
            - SalespersonViewExSDO.java
            - SDeptViewSDO.java
            - SEmpViewSDO.java
    - sdo.commonj
    - sdo.commonj.java
    - sdo.commonj.xml
    - META-INF

As the following example shows, when the `EmployeeView` view object has a subtype `SalespersonView` view object defined, the generated web service client contains the proxy class `SalespersonViewSDO.java` which extends the base proxy class (`SEmpViewSDO`) and defines the accessor methods for the `commissionPct` attribute.

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "SalespersonViewExSDO", propOrder = { "commissionPct" })
public class SalespersonViewExSDO extends SEmpViewSDO {

    @XmlElementRef(name = "CommissionPct", namespace =
                   "/oracle/summit/model/polymorphicvo/views/common/",
                   type = JAXBElement.class, required = false)
    protected JAXBElement<BigDecimal> commissionPct;
    /**
     * Gets the value of the commissionPct property.
     *
     * @return
     *     possible object is
     *     {@link JAXBElement }{@code <}{@link BigDecimal }{@code >}
     *
     */
    public JAXBElement<BigDecimal> getCommissionPct() {
        return commissionPct;
    }

    /**
     * Sets the value of the commissionPct property.
     *
     * @param value
     *     allowed object is
     *     {@link JAXBElement }{@code <}{@link BigDecimal }{@code >}
     *
     */
    public void setCommissionPct(JAXBElement<BigDecimal> value)
{        this.commissionPct = value;
    }
}
```

What You May Need to Know About Invoking Service Methods in the Client Application

The generated service factory classes of the ADF Business Components service interface define the builtin methods that may be optionally enabled to create the service view instances. For example, for the polymorphic view object, the `createSalespersonView()` is enabled to create an instance of the polymorphic view object. When you use the create factory method to define the attributes of the view instance, the method signature for required attributes of the view object differs from attributes that may be optionally defined.

For example, when the `SalepersonViewSDO.java` defines setters for the required attributes of the view object, your client code can use a Java primitive value for the required attribute. However, setters for optional attributes must use a special type that indicates the XML element is missing and is not just an empty value. The factory object method supports the special type for optional view object attributes. As the following example shows, the method signature for `setFirstName` invokes `factory.createSEmpViewSDOFirstName`, whereas, the method signature for required attributes, like `setId` and `setLastName`, take an integer and String value.

```
private static SEmpViewSDO testCreate(AppModuleService deptService) throws
                                                              ServiceException
  {
    oracle.summit.model.polymorphicvo.views.common.ObjectFactory factory = new
            oracle.summit.model.polymorphicvo.views.common.ObjectFactory();
    System.out.println("\n*** Testing createEmp ***\n");
    SalespersonViewExSDO salesperson = factory.createSalespersonViewExSDO();
    salesperson.setId(8001);
    salesperson.setFirstName(factory.createSEmpViewSDOFirstName("Lynn"));
    salesperson.setLastName("Munsinger");
    salesperson.setEmail
            (factory.createSEmpViewSDOEmail("lmunsing@summit.com"));
    salesperson.setTitleId
            (factory.createSEmpViewSDOTitleId(new BigDecimal(2)));
    salesperson.setSalary
            (factory.createSEmpViewSDOSalary(new BigDecimal(2000)));
    salesperson.setCommissionPct
            (factory.createSalespersonViewExSDOCommissionPct(new
BigDecimal(20)));
    salesperson.setDeptId(factory.createSEmpViewSDODeptId(31));

    return printEmp(deptService.createSEmpView1(salesperson));
}
```

Working with the Find Method Filter Model in the Consuming Application

ADF Business Components defines a hierarchical filter model for the `FindCriteria` object that allows SOAP clients to control the behavior of find methods invoked as a SOAP request or in client code, programmatically.

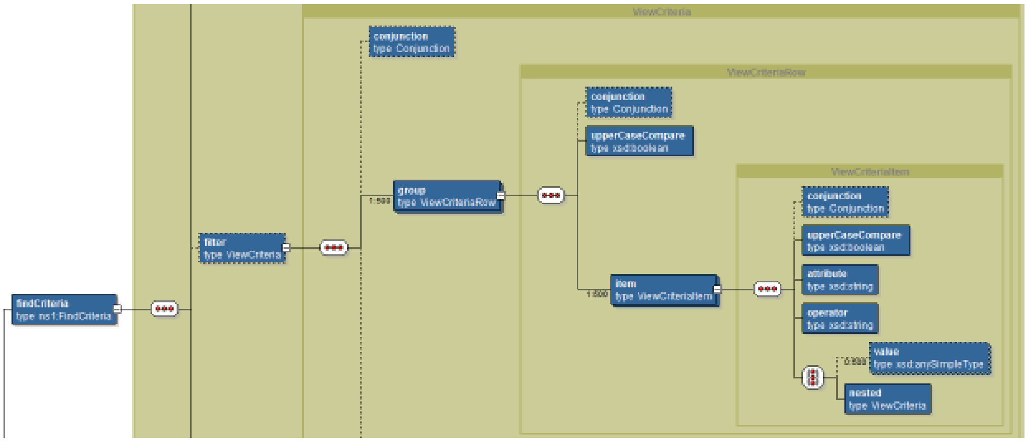
filter —>group—>item

The filter contains all runtime conditions and corresponds to a view criteria in ADF. View criteria restrict returned data to those rows matching the defined filters. At runtime, the filters will be converted to SQL WHERE clauses for the purpose of providing a data filter to a view object. Within the filter, the group contains a set of runtime filter conditions for one or more attributes of the SDO.

> **Note:**
>
> Attributes that participate in find method queries must be defined as queriable by the backing view object. The item defines one filter condition for a specific attribute.

The XSD diagram for the FindCriteria object shows in detail how the arguments for the find method are constructed.

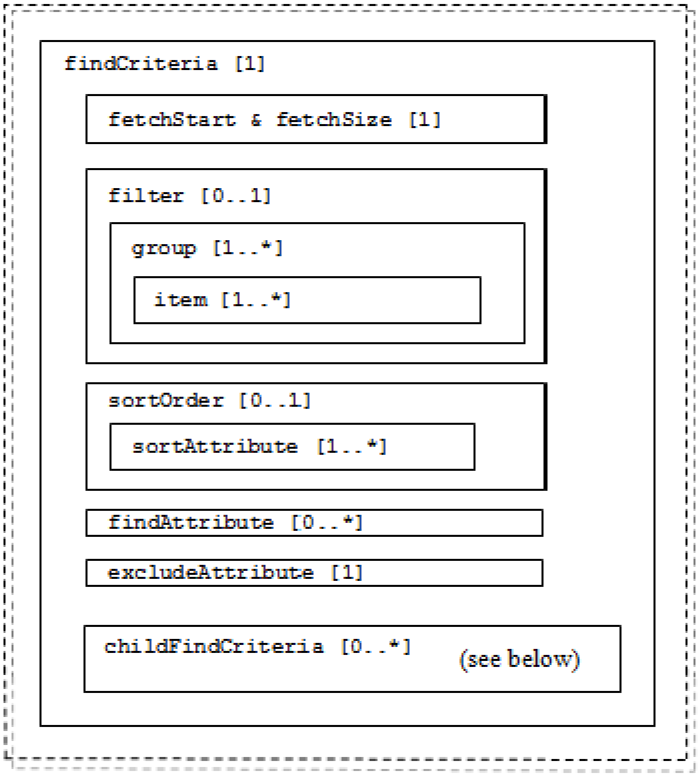
Figure 15-22   FindCriteria XSD Diagram



Conjunctions (And, Or, Not, AndNot, OrNot) may be specified to define multiple items in a group and define filter conditions are evaluated in relation to each other. For example, an And conjunction specifies that the query results meet both joined conditions. An Or conjunction specifies that the query results meet either or both joined conditions. When a conjunction is omitted, And behavior is assumed.

The following block diagram is another way to represent the containment model of the FindCriteria object.

Figure 15-23   FindCriteria Containment Hierarchy



To specify the filter `item` for the FindCriteria, the client must define the following:

- `Attribute` is a case-sensitive attribute name to be filtered by the filter condition.
- `Operator` is applied to the filter condition. Operators are appropriate to the attribute type and are defined by `oracle.jbo.common.JboCompOper`. See note below for exceptions.
- `Value` specifies the attribute filter criteria, which can include non-leading `%` wildcard.

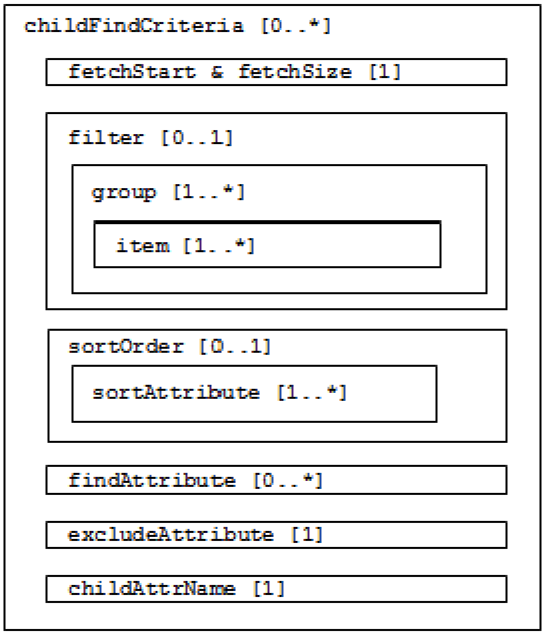
> **Note:**
>
> SOAP services clients may specify conditions using all operators defined by `oracle.jbo.common.JboCompOper`, with the exception of LIKE (use CONTAINS instead), IN, and NOTIN. The operators IN and NOTIN have runtime support and may therefore be applied programmatically. The operators CONTAINSALL and CONTAINSDELIMITEDID are supported for multi-select attributes.

Additionally, the client may specify the following FindCriteria elements to format the result set response:

- `FetchStart` (required) specifies the zero-based row index to start with. The default is 0 (first row).
- `FetchSize` (required) specifies the maximum number of rows to retrieve.
- `SortOrder` (optional) contains directive for sorting the result set in ascending or descending order.
- `FindAttribute` (optional) specifies the subset of attributes to retrieve when the filter criteria is satisfied. When none is satisfied, all attributes are fetched.
- `ExcludeAttribute` (optional) a boolean flag that when set to `True`, attributes specified by `FindAttribute` are not returned.

When attributes of a child object need to be conditionally evaluated and/or returned in the payload, a `childFindCriteria` structure may be added within the FindCriteria structure. The elements of the `childFindCriteria` are the same as used in the FindCriteria structure, with one addition: `childAttrName` specifies which child object to apply the filter criteria to, as the following block diagram illustrates for a single child object.

Figure 15-24 FindCriteria Containment Hierarchy with Child Find Criteria



How to Control Find Method Behavior Using a SOAP Request

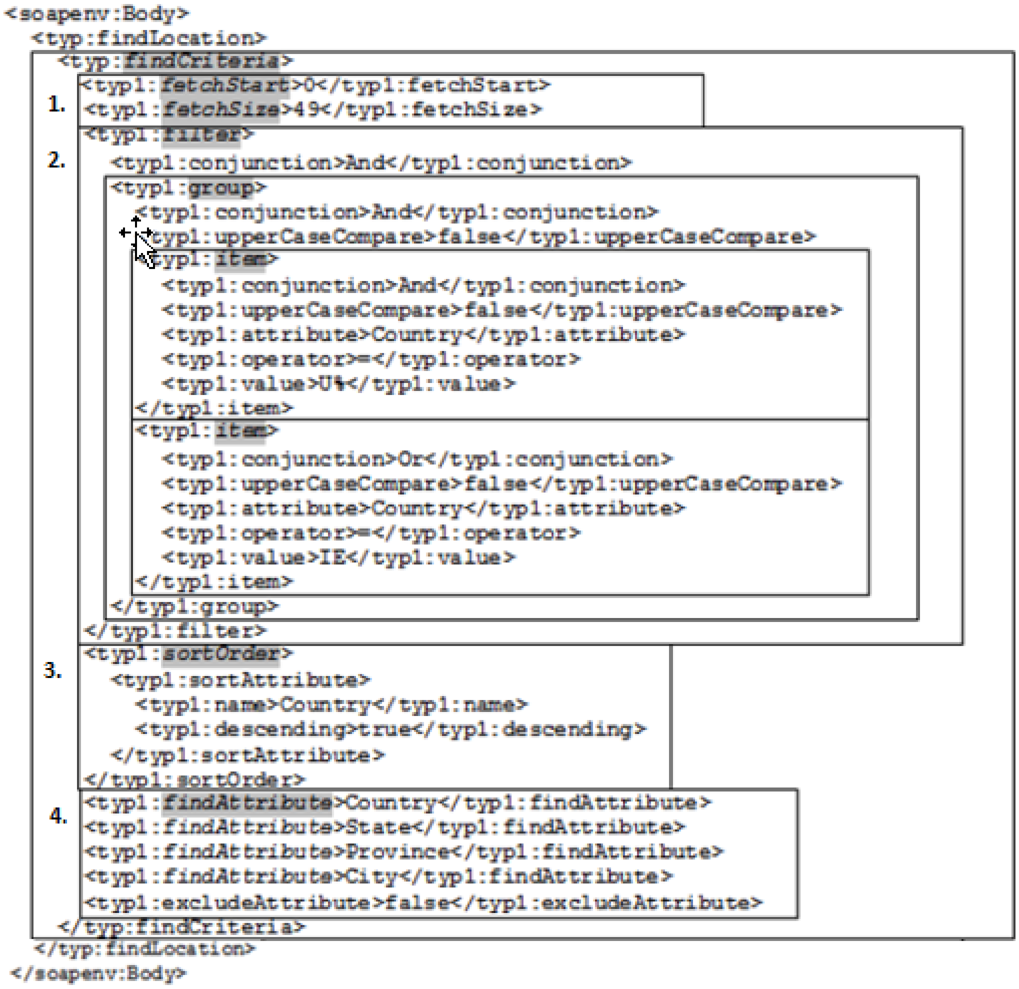
To construct a find request payload for a SDO, it is necessary to understand the containment hierarchy of the find method filter model. Payload building blocks are derived from the elements defined by the find method XSD. For details about the containment hierarchy, see Working with the Find Method Filter Model in the Consuming Application.

To define a SOAP request for a FindCriteria, follow this process:

1. Specify the number of rows in the result set, and on which row to start (required).
2. Define the filter condition on the parent object, specifying one or more view criteria items (required).
3. Specify the sort order for the result, if desired (optional). Default is descending order.
4. Specify one or more attributes to retrieve when the filter criteria is satisfied (optional).
5. Specify filter conditions on child objects, if any (optional). See Figure 15-25.

The following request is designed to return a result set matching this description on the parent object:
Find up to 50 Locations where the Country attribute value is US or IE. Return only the Country, State, Province, and City attributes. Sort the result set in descending Country order. Do not return any translated values.

Figure 15-25   Find Request on Parent Object

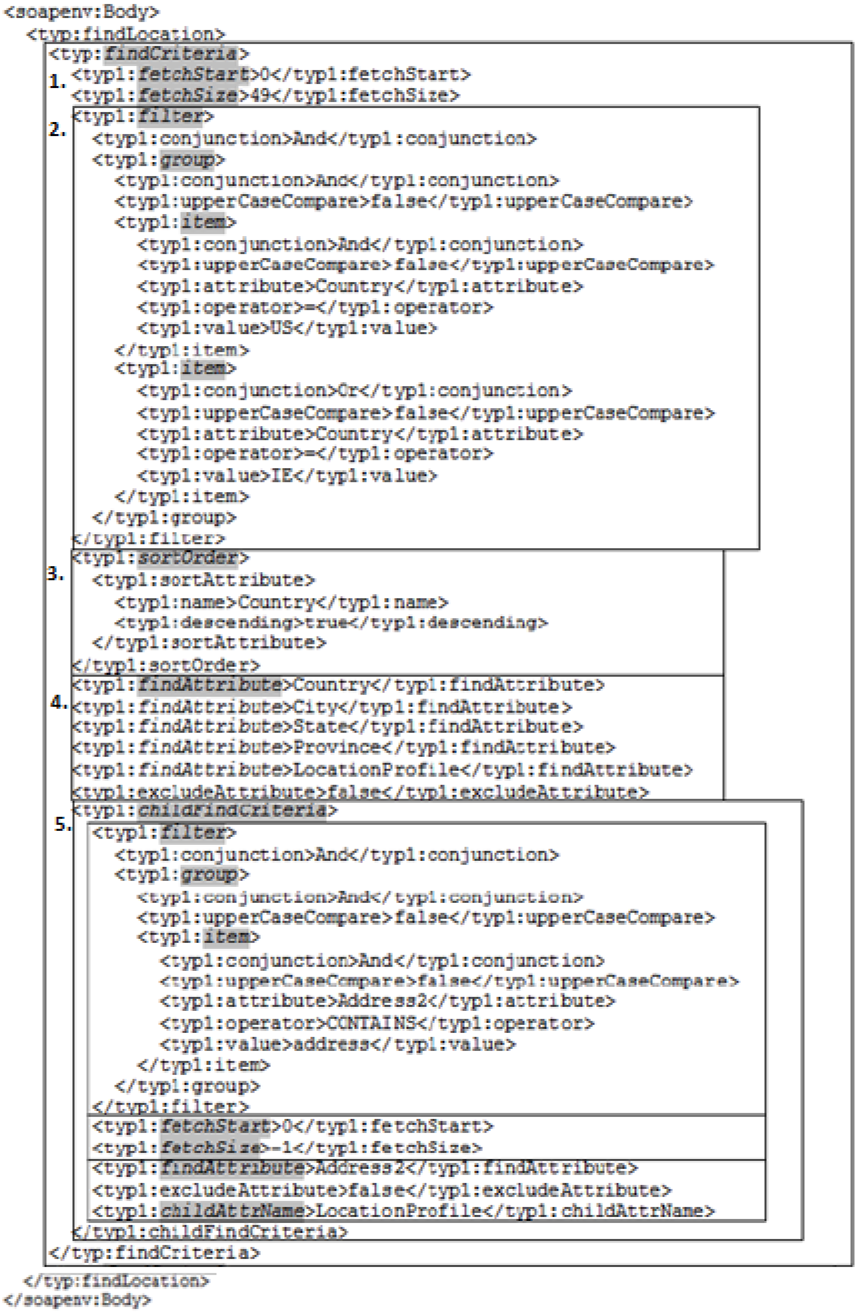


```
<soapenv:Body>
  <typ:findLocation>
    <typ:findCriteria>
      <typ1:fetchStart>0</typ1:fetchStart>
      <typ1:fetchSize>49</typ1:fetchSize>
      <typ1:filter>
        <typ1:conjunction>And</typ1:conjunction>
        <typ1:group>
          <typ1:conjunction>And</typ1:conjunction>
          <typ1:upperCaseCompare>false</typ1:upperCaseCompare>
          <typ1:item>
            <typ1:conjunction>And</typ1:conjunction>
            <typ1:upperCaseCompare>false</typ1:upperCaseCompare>
            <typ1:attribute>Country</typ1:attribute>
            <typ1:operator>=</typ1:operator>
            <typ1:value>U%</typ1:value>
          </typ1:item>
          <typ1:item>
            <typ1:conjunction>Or</typ1:conjunction>
            <typ1:upperCaseCompare>false</typ1:upperCaseCompare>
            <typ1:attribute>Country</typ1:attribute>
            <typ1:operator>=</typ1:operator>
            <typ1:value>IE</typ1:value>
          </typ1:item>
        </typ1:group>
      </typ1:filter>
      <typ1:sortOrder>
        <typ1:sortAttribute>
          <typ1:name>Country</typ1:name>
          <typ1:descending>true</typ1:descending>
        </typ1:sortAttribute>
      </typ1:sortOrder>
      <typ1:findAttribute>Country</typ1:findAttribute>
      <typ1:findAttribute>State</typ1:findAttribute>
      <typ1:findAttribute>Province</typ1:findAttribute>
      <typ1:findAttribute>City</typ1:findAttribute>
      <typ1:excludeAttribute>false</typ1:excludeAttribute>
    </typ:findCriteria>
  </typ:findLocation>
</soapenv:Body>
```

The following request is designed to return a result set for a filtered child object matching this description:

For each row in the result set (each qualifying object instance), include the LocationProfile: including the FloorNumber, Building, City, State, Province, Country and EffectiveStartDate attributes, if the EffectiveStartDate is on or after the date specified.

Figure 15-26   Find Request on Parent and Child Objects



```
<soapenv:Body>
  <typ:findLocation>
   <typ:findCriteria>
     <typ1:fetchStart>0</typ1:fetchStart>
     <typ1:fetchSize>49</typ1:fetchSize>
     <typ1:filter>
       <typ1:conjunction>And</typ1:conjunction>
       <typ1:group>
         <typ1:conjunction>And</typ1:conjunction>
         <typ1:upperCaseCompare>false</typ1:upperCaseCompare>
         <typ1:item>
           <typ1:conjunction>And</typ1:conjunction>
           <typ1:upperCaseCompare>false</typ1:upperCaseCompare>
           <typ1:attribute>Country</typ1:attribute>
           <typ1:operator>=</typ1:operator>
           <typ1:value>US</typ1:value>
         </typ1:item>
         <typ1:item>
           <typ1:conjunction>Or</typ1:conjunction>
           <typ1:upperCaseCompare>false</typ1:upperCaseCompare>
           <typ1:attribute>Country</typ1:attribute>
           <typ1:operator>=</typ1:operator>
           <typ1:value>IE</typ1:value>
         </typ1:item>
       </typ1:group>
     </typ1:filter>
     <typ1:sortOrder>
       <typ1:sortAttribute>
         <typ1:name>Country</typ1:name>
         <typ1:descending>true</typ1:descending>
       </typ1:sortAttribute>
     </typ1:sortOrder>
     <typ1:findAttribute>Country</typ1:findAttribute>
     <typ1:findAttribute>City</typ1:findAttribute>
     <typ1:findAttribute>State</typ1:findAttribute>
     <typ1:findAttribute>Province</typ1:findAttribute>
     <typ1:findAttribute>LocationProfile</typ1:findAttribute>
     <typ1:excludeAttribute>false</typ1:excludeAttribute>
     <typ1:childFindCriteria>
       <typ1:filter>
         <typ1:conjunction>And</typ1:conjunction>
         <typ1:group>
           <typ1:conjunction>And</typ1:conjunction>
           <typ1:upperCaseCompare>false</typ1:upperCaseCompare>
           <typ1:item>
             <typ1:conjunction>And</typ1:conjunction>
             <typ1:upperCaseCompare>false</typ1:upperCaseCompare>
             <typ1:attribute>Address2</typ1:attribute>
             <typ1:operator>CONTAINS</typ1:operator>
             <typ1:value>address</typ1:value>
           </typ1:item>
         </typ1:group>
       </typ1:filter>
       <typ1:fetchStart>0</typ1:fetchStart>
       <typ1:fetchSize>-1</typ1:fetchSize>
       <typ1:findAttribute>Address2</typ1:findAttribute>
       <typ1:excludeAttribute>false</typ1:excludeAttribute>
       <typ1:childAttrName>LocationProfile</typ1:childAttrName>
     </typ1:childFindCriteria>
   </typ:findCriteria>
  </typ:findLocation>
</soapenv:Body>
```

How to Control Find Method Behavior Using a Java Client

The standard find method API provides control over the query behavior. To work with the API, it is necessary to understand the object containment hierarchy of the find method filter model. For details about the containment hierarchy, see Working with the Find Method Filter Model in the Consuming Application,

The following sample illustrates the process for programmatically creating a `FindCriteria` object:

1. Create a new `FindCriteria` (the Filter).
2. Create a new `ViewCriteriaItem` (the Item).
3. Create a new `ViewCriteriaRow` (the Group).
4. Add the created items and groups to the `FindCriteria`.

```
//1. Create FindCriteria
     final FindCriteria findCriteria =
(FindCriteria)DataFactory.INSTANCE.create(FindCriteria.class);
     findCriteria.setFetchStart(0);
     findCriteria.setFetchSize(-1);
     findCriteria.setExcludeAttribute(false);

//2. Create ViewCriteriaItem
     final ViewCriteriaItem viewCriteriaItem =
(ViewCriteriaItem)DataFactory.INSTANCE.create(ViewCriteriaItem.class);
     viewCriteriaItem.setConjunction("AND"); //Can also be OR
     viewCriteriaItem.setUpperCaseCompare(true);
     viewCriteriaItem.setAttribute("EmpName");
     viewCriteriaItem.setOperator("LIKE"); //LIKE used for WILDCARD comparison
      final List<Object> item_values = new ArrayList<Object>(5);
     item_values.add(viewCriteriaItem);
      viewCriteriaItem.setValue(item_values);

//3. Create ViewCriteriaRow.
//4. Further add ViewCriteriaItem into ViewCriteriaRow and ViewCriteriaRow into
FindCriteria.
     final ViewCriteriaRow viewCriteriaRow =
(ViewCriteriaRow)DataFactory.INSTANCE.create(ViewCriteriaRow.class);
     viewCriteriaRow.setConjunction("AND");
     viewCriteriaRow.setUpperCaseCompare(true);
     final List<Object> item = new ArrayList<Object>(5);
     item.add(viewCriteriaItem);
     viewCriteriaRow.setItem(item);
     findCriteria.getFilter().getGroup().add(viewCriteriaRow);
```

When you want to query with a list of partial attributes list

By default, the find operation returns all the attributes including all details. When you only need some attributes, you should set the partial attributes on the `FindCriteria` parameter of the find method. Do this in the following situations:

- SDO contains LOB, which can be very expensive to retrieve and transfer.
- SDO contains details that are not needed, such as translations. Querying detail is also expensive.

Note: The standard `getXXX` function does not take a `FindCriteria`, so this function always returns everything. You should use `findXXX` to trim your return attributes.

The following example shows how to set the partial attributes to include only `Dname`, `Loc` from `Dept`, and exclude `Empno` from `Emp`.

```
FindCriteria fc = (FindCriteria)datafactory.create(FindCriteria.class);
List l = new ArrayList();
l.add("Dname");
l.add("Loc");
l.add("Emp");
fc.setFindAttribute(l);
List cfcl = new ArrayList();
ChildFindCriteria cfc =
(ChildFindCriteria)datafactory.create(ChildFindCriteria.class);
cfc.setChildAttrName("Emp");
List cl = new ArrayList();
cl.add("Empno");
cfc.setFindAttribute(cl);
cfc.setExcludeAttribute(true);
cfcl.add(cfc);
fc.setChildFindCriteria(cfcl);
DeptResult res = svc.findDept(fc, null);
```

The following example shows how to set the partial attributes to exclude `PurchaseOrderLine` from `PurchaseOrder`.

```
FindCriteria fc = (FindCriteria)datafactory.create(FindCriteria.class);
List l = new ArrayList();
fc.setExcludeAttribute(true);
l.add("PurchaseOrderLine");
fc.setFindAttribute(l);
PchaseOrderResult res = svc.findPurchaseOrder(fc, null);
```

When you want to filter the result using a view criteria

The find API allows you to specify the WHERE clause of your query. The WHERE clause can be set on any level of the SDO.

The following example shows how to retrieve only the departments with a department number greater than 10 and child employees whose names start with "A".

```
FindCriteria fc = (FindCriteria)datafactory.create(FindCriteria.class);
//create the view criteria item
List value = new ArrayList();
value.add(new Integer(10));
ViewCriteriaItem vci =
(ViewCriteriaItem)datafactory.create(ViewCriteriaItem.class);
vci.setValue(value);
vci.setAttribute("Deptno");
List<ViewCriteriaItem> items = new ArrayList(1);
items.add(vci);
//create view criteria row
ViewCriteriaRow vcr = (ViewCriteriaRow)
datafactory.create(ViewCriteriaRow.class);
vcr.setItem(items);
//create the view criteria
List group = new ArrayList();
group.add(vcr);
ViewCriteria vc = (ViewCriteria)datafactory.create(ViewCriteria.class);
vc.setGroup(group);
//set filter
fc.setFilter(vc);

List cfcl = new ArrayList();
//create the child find criteria
ChildFindCriteria cfc =
(ChildFindCriteria)datafactory.create(ChildFindCriteria.class);
cfc.setChildAttrName("Emp");
//create the child view criteira
ViewCriteria cvc = (ViewCriteria)datafactory.create(ViewCriteria.class);
cfc.setFilter(cvc);
//create the view criteria item
List cvalue = new ArrayList();
cvalue.add("A%");
```

```
ViewCriteriaItem cvci =
(ViewCriteriaItem)datafactory.create(ViewCriteriaItem.class);
cvci.setValue(value);
cvci.setAttribute("Dname");
cvci.setOperator("LIKE");
List<ViewCriteriaItem> citems = new ArrayList(1);
citems.add(cvci);
//create child view criteria row
ViewCriteriaRow cvcr = (ViewCriteriaRow)
datafactory.create(ViewCriteriaRow.class);
cvcr.setItem(citems);
List cgroup = new ArrayList();
cgroup.add(cvcr);
cvc.setGroup(cgroup);

DeptResult dres = svc.findDept(fc, null);
```

You can also query the parents with the children that satisfy certain criteria. For example, the following sample retrieves the departments with employees whose salary is greater than $10,000.

```
//create the view criteria item on the employees
List nvalue = new ArrayList();
nvalue.add(new BigDecimal(10000));
ViewCriteriaItem nvci =
(ViewCriteriaItem)datafactory.create(ViewCriteriaItem.class);
nvci.setValue(nvalue);
nvci.setAttribute("Salary");
nvci.setOperation(">");
List<ViewCriteriaItem> nitems = new ArrayList(1);
nitems.add(nvci);
//create view criteria row
ViewCriteriaRow nvcr = (ViewCriteriaRow)
datafactory.create(ViewCriteriaRow.class);
nvcr.setItem(nitems);
//create the nested view criteria
List ngroup = new ArrayList();
ngroup.add(nvcr);
ViewCriteria nvc = (ViewCriteria)datafactory.create(ViewCriteria.class);
nvc.setGroup(ngroup);

//create the view criteria item on the department
ViewCriteriaItem vci =
(ViewCriteriaItem)datafactory.create(ViewCriteriaItem.class);
vci.setAttribute("Emp");
vci.setNested(nvc);
List<ViewCriteriaItem> items = new ArrayList(1);
items.add(vci);
//create view criteria row
ViewCriteriaRow vcr = (ViewCriteriaRow)
datafactory.create(ViewCriteriaRow.class);
vcr.setItem(items);
//create view criteria on department
ViewCriteria vc = (ViewCriteria)datafactory.create(ViewCriteria.class);
List group = new ArrayList();
group.add(vcr);
vc.setGroup(group);
//set filter
FindCriteria fc = (FindCriteria)datafactory.create(FindCriteria.class);
fc.setFilter(vc);
```

```
DeptResult dres = svc.findDept(fc, null);
```

When you want to page the result

If you know that your query might return a large amount of data, you should make multiple service invocations, and use `setFetchStart` and `setFetchSize` to control the amount of the data that you want to retrieve.

The following sample shows how to retrieve only the second employee and the employee's department.

```
FindCriteria fc = (FindCriteria)datafactory.create(FindCriteria.class);
List cfcl = new ArrayList();
ChildFindCriteria cfc =
(ChildFindCriteria)datafactory.create(ChildFindCriteria.class);
cfc.setChildAttrName("Emp");
cfc.setFetchStart(1);
cfc.setFetchSize(1);
cfcl.add(cfc);
fc.setChildFindCriteria(cfcl);
DeptResult dres = svc.findDept(fc, null);
```

16

Creating ADF RESTful Web Services with Application Modules

This chapter describes how to create ADF REST resources based on view object instances of the ADF application module to make these resources available as ADF RESTful web services in a Fusion web application.
This chapter includes the following sections:

About RESTful Web Services and ADF Business Components

ADF Business Components view object instances can be exposed as web services using REST. Client developers interact with these resources through the REST API enabled by the ADF REST runtime.

REpresentational State Transfer (REST) supports a stateless client-server architecture in which the web services are viewed as resources identified by their URL. HTTP Request methods such as GET and POST are the verbs that the developer can use in a web service client request message to describe the necessary create, read, update, and delete (CRUD) actions to be performed on the resource represented by the HTTP URL. REST prescribes the use of standards such as HTTP, URL, URI, XML, HTML, as well as Resource Types and MIME Types. Therefore, REST is not a standard but a proposed architectural style for web services.

Web services conforming to the REST design constraints are referred to as being RESTful. This type of web service allows the HTTP client and the HTTP server to exchange information about resources identified by using a URL. The request and response contain a representation of the resource, in a specific format, which defines the state of the actual resource.

In the Fusion web application, ADF REST resources acted on by RESTful web services are backed by view object instances exposed in the data model of the ADF Business Components Model project. ADF Business Components developers working in the ADF Model project can decide on the set of attributes to expose from backing view objects, the actions to make available CRUD operations, and the view link relationships to preserve for the resulting resource.

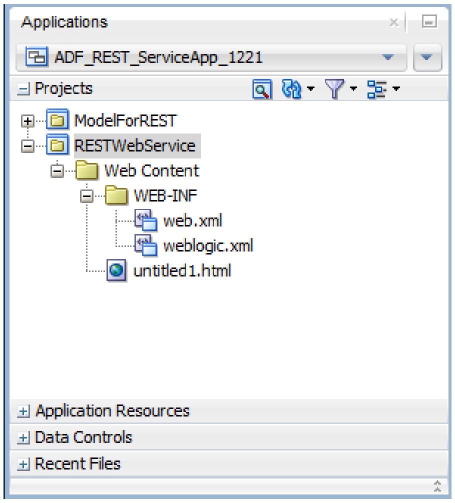
Note:

When you create view objects that you intend to expose as ADF REST resources, use the **Declarative** option on the Query page of the Create View Object wizard to create view objects with declarative SQL mode option enabled. View objects created in declarative SQL mode at design time support an ADF REST framework runtime optimization that allows resource collections to be efficiently filtered by client requests made with the URL query parameter `fields`.

The design-time choices made by ADF Business Components developers are captured in REST resource definition XML files. Web service client developers may interact with these resource definitions through the REST API enabled by the ADF REST runtime. As a result, the consuming service client may invoke CRUD operations to interact with the REST resources and ADF business objects. The data is shaped by the resource's backing view object instance, with the parent-child relationships intact.

When you create the first REST resource in the ADF Business Components project, JDeveloper also generates a separate RESTful web service project to register the ADF REST servlet. Figure 16-1 shows the `web.xml` file that JDeveloper generates. You use this project to run and deploy the RESTful web service.

Figure 16-1 Applications Window Displays RESTful Web Service Project



Security for ADF REST resources is enforced in RESTful web services similar to the way you secure the Fusion web application. In JDeveloper, you run the Configure ADF Security wizard to enforce security on the ADF Model project. After you run the wizard, the REST resources of the deployed RESTful web service will be protected by default and will require that you grant access to the operations of the service.

You can test security in JDeveloper by creating test users and granting them access privileges to the resources. The security policies that confer these access rights, may then be migrated from the application level to a domain-level security policy store, where enterprise users are provisioned with access rights.

RESTful Web Services Use Cases and Examples

Any development team can publish an ADF REST resource to contribute to the Fusion web application. ADF REST application developers can manage the functionality that they expose to the service client using these version features:

- Registering REST resources with an application-specific resource version name (or number) to support versioning of REST resources.
- Declaring a default ADF REST framework version for REST resources allows service clients to utilize new features and enhancements at runtime, such as the advanced query capabilities offered in version 2 of the ADF REST runtime framework. By declaring a default framework version to process requests, REST application developers may opt into the features introduced by a specific ADF REST framework when they are ready. In JDeveloper starting with release 12.2.1.2.0, versions 1, 2, and 3 exist. In the release 12.2.1.4.0 and later, versions 1, 2, 3, 4, 5, 6, and 7 exist.

Specific runtime features of the ADF REST framework that support the exchange of resource information by client and server include:

- Interacting with the REST resource is supported by separate JSON-based media type structures for the resource, action execution, and the results of action execution.
- Passing a custom header to specify the version of the ADF REST runtime framework that will be used to process requests. The version header allows service clients to override the default version declaration defined by the REST application. In JDeveloper starting with release 12.2.1.2.0, versions 1, 2, and 3 exist. In the release 12.2.1.4.0 and later, versions 1, 2, 3, 4, 5, 6, and 7 exist.
- Interacting with the REST resource using standard HTTP request methods, including GET, POST, PATCH, and DELETE.
- Executing an HTTP method using an `X-HTTP-Method-Override` header on a POST request method when client restrictions prevent the use of standard HTTP methods to interact with the REST resource.
- Merging payload content with the REST resource using a POST method in combination with a header `Upsert-Mode` set to true. This action implements create if the record does not exist or update if the record exists.
- Locating a REST resource item (such as a specific employee) or REST resource collection (such as an collection of all employees) is supported by REST-compliant URI path names based on resource names defined in the ADF Business Components Model project.
- Obtaining a description of the REST resource, including the resource attributes and available actions, is supported by a specific media type and describe action.
- Obtaining a error responses in the form of a JSON payload with exception detail when ADF REST framework version 4 or later is enabled. Alternatively, with framework version 3 or earlier, error responses are in the form of a simple message string.

- Linking to a canonical REST resource is supported when a resource with a super set of updatable entities is defined. The canonical link supports alternate links for the backing view object.
- Filtering of the REST resource is supported by query string parameters on the URI specified by the client to access the specific resource.
- Encoding formats are supported on the REST resource to enable compression and decompression.
- Content streaming of BLOB and CLOB attributes exposed by the REST resource is supported by the ADF REST framework.
- Performing data consistency checks while invoking the RESTful web service is supported by the ADF REST framework. This capability uses version history in the database to enable clients to manage HTTP payloads according to updates in the resource itself.
- Authorizing users to access the REST resource is enabled by ADF Security permission grants.

Additional Functionality for RESTful Web Services

You may find it helpful to understand other Oracle ADF features before you start working with RESTful web services. Following are links to supporting functionality that may be of interest.

- For details about creating view objects with the recommended declarative SQL mode enabled to support efficient resource collection filtering at runtime, see Working with View Objects in Declarative SQL Mode,
- For details about enabling change indicator attributes to entity objects when you want to support data consistency checking on the service client, see How to Protect Against Losing Simultaneously Updated Data.
- For details about creating row finders on view object when you want to expose finders in ADF REST resource, see Working with Row Finders.
- For details about creating LOV-enabled view object attributes when you want to expose LOV links in the ADF REST resource, see Working with List of Values (LOV) in View Object Attributes.
- For information about creating polymorphic view objects to expose subtype view rows in the ADF REST resource, see Working with Polymorphic View Rows.
- For information about the ADF REST framework that enables runtime access to ADF REST resources using a REST API, see Consuming ADF RESTful Web Services.
- For details about using the ADF REST data control to design the user interface, see Creating ADF REST Data Controls from ADF RESTful Web Services.
- For information about creating ADF REST resources using Create Business Components from Tables wizard, see How to Create a Data Model Project for ADF REST Web Applications.

Creating ADF REST Resources Using the Application Module

You can use view object instances that you expose on the ADF application module to create the ADF REST resource.

The application module is an ADF Business Components framework component that encapsulates business logic as a set of related business functions exposed as view object instances. You may map one or more view object instances that appear in the application module data model to ADF REST resources. You use the overview editor for your application module to create the ADF REST resource. The REST resource that you add to the ADF Model project may be published as a web service that follows the REST architectural style.

> **Note:**
>
> When you create view objects that you intend to expose as ADF REST resources, use the **Declarative** option on the Query page of the Create View Object wizard to create view objects with declarative SQL mode option enabled. View objects created in declarative SQL mode at design time support an ADF REST framework runtime optimization that allows resource collections to be efficiently filtered by client requests made with the URL query parameter `fields`.

How to Create ADF REST Resources Using the Create Business Components from Tables Wizard

In the New Gallery, in the **ADF Business Components** category, JDeveloper offers a wizard to create each kind of business component. Each wizard allows you to specify the component name for the new component and to select the package into which you'd like to organize the component. If the package does not yet exist, the new component becomes the first component in that new package.

The Create Business Components from Tables wizard is particularly useful in JDeveloper because it is the only wizard that combines generating the various business components along with creating REST resources in a single end to end process. This wizard lets you create REST resources quickly and easily. You can create entity objects based on an online or offline database, then you can create either entity-based view objects or query-based view objects, and an application module to contain the view instances of the data model, and then create the ADF REST resources.

Before you begin:

It may be helpful to have an understanding of the various ADF business components. For more information, see About ADF Business Components.

You will need to complete this task:

Create the data model project that will contain the business components, as described in How to Create a Data Model Project for ADF REST Web Applications.

To create REST resources from **Business Components from Tables** wizard in one pass:

1. In the Applications window, right-click the project in which you want to create the entity objects and choose **New**.
2. In the New Gallery, expand **Business Tier**, select **ADF Business Components** and then **Business Components from Tables**, and click **OK**.

   If this is the first component you are creating in the project, the Initialize Business Components Project dialog appears to allow you to select a database connection. Complete the dialog, as described in How to Initialize the Data Model Project With a Database Connection.
3. In the Create Business Components from Tables wizard, on the Entity Objects page, filter the database schema to display available database objects and select those database objects for which you want to create entity objects. Click **Next**.

   Note that if you change a default package name, you must not use reserved words like `rest` in the package name. For details about package naming, see What You May Need to Know About Package Naming Conventions.
4. On the Entity-based View Objects page, select among the available entity objects to create updatable view object definitions for your entity object definitions. Click **Next**.

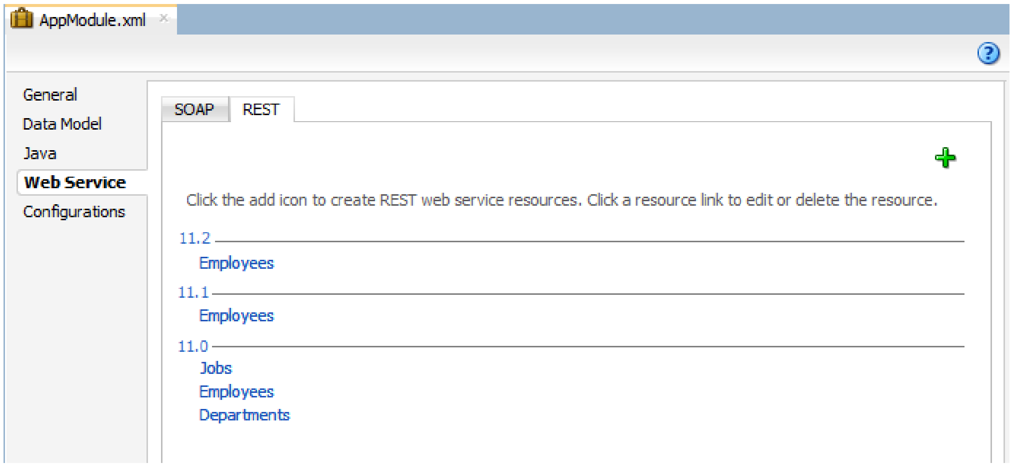
   Note that the view object definitions you create in this wizard will contain a single usage of an entity object, and will expose all the attributes in that object.
5. On the Query-based View Objects page, select among the available database objects to create read-only view object definitions. Click **Next**.
6. On the Application Module page, select the **Add to Application Module** checkbox and leave the other default selections unchanged. Click **Next**.
7. On the REST Resources page, choose an existing Release Version from the dropdown list or click the **Create New Release Version** icon. Select among the available view objects to add as view instances and create REST resources. Click **Next**.
8. You can optionally specify attributes to hide in the resource payload and also shape it in the Attribute Settings page. Click **Next**.
9. On the Summary page, review the list of business components that the wizard will create with REST resources, and click **Finish**.
10. You may create additional ADF REST resources using the overview editor for application modules. For details, see How to Create ADF REST Resources from View Object Instances.

How to Create ADF REST Resources from View Object Instances

View object instances are the business objects on which the Fusion web application operates. The ADF application module exposes these business objects in the data model it defines to specify the data that the service client can display and manipulate. After you have created the Model project, optionally using the application template for ADF REST, you may create additional ADF REST resources using the overview editor for the application module. In this case, any view object instance that the application module defines may have an ADF REST resource defined.

Adding an ADF REST resource is an option that appears in the Web Service page of the application module overview editor. Figure 16-2 shows how the editor displays the REST resources that have been created for view object instances on the AppModule application module. Notice that the editor organizes REST resources on the application module by their assigned release-specific version names (in this example, named 11.0, 11.1, and 11.2).

Figure 16-2 ADF REST Resources Defined on the Application Module



Click the add icon to create REST web service resources. Click a resource link to edit or delete the resource.

11.2
- Employees

11.1
- Employees

11.0
- Jobs
- Employees
- Departments

Note:

When you create an ADF REST resource JDeveloper prompts you to specify a version release name for the resource. The version is an alphanumeric identifier that helps you to manage the current and subsequent versions of resources that you add to the ADF Model project across multiple releases.

After you select the view object instance, name the REST resource, and select a version, JDeveloper opens the resource in the overview editor for the REST resource. You use the overview editor to edit the REST resource representation by exposing view object accessors and available operations as payload objects and actions of RESTful web services. JDeveloper saves the REST resource definition files as subpackages of the ADF application module organized by their assigned release version names.

For details about customizing the REST resource, see How to Edit the ADF REST Resource Definition.

Before you begin:

It may be helpful to have an understanding of how the ADF REST framework supports application modules and enables web service clients to access rows of data and perform service operations. For more information, see About RESTful Web Services and ADF Business Components.

You may also find it helpful to understand other Oracle ADF features that depend on ADF REST resources. For more information, see Additional Functionality for RESTful Web Services.

You will need to complete these tasks:

- Create the application module data model that contains the view object instances that you want the REST resource to expose, as described in How to Create an Application Module.
- Create a release version identifier in the `adf-config.xml` file to associate ADF REST resources you create with a specific version. Each REST resource must be associated with a release version to help manage the current and subsequent versions of ADF REST resources that you add to the ADF Model project, as described in Versioning the ADF REST Resource.

To create the ADF REST resource:

1. In the Applications window, double-click the application module that contains the view instance for which you want to create the ADF REST resource.
2. In the overview editor, click the **Web Service** navigation tab, then click the **REST** tab, and then click the **Enable Support for REST Service** add icon.

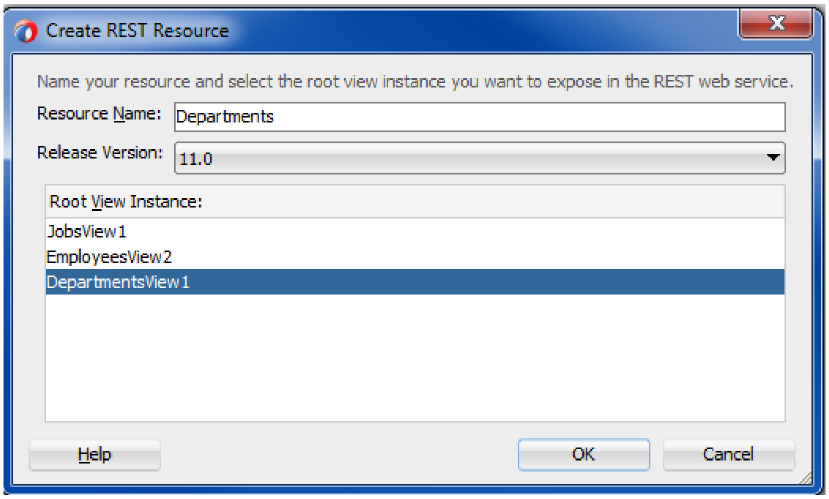
   When you create the first REST resource, JDeveloper also generates a separate RESTful web service project to register the ADF REST servlet.
3. In Create REST Resource dialog, select the root view instance to expose in a RESTful web service.

   The dialog displays view instances defined in the application module data model. Only the root view instance of a master-detail hierarchy may be selected.
4. Enter the **Resource Name** of the REST resource collection and select a release version from the dropdown list, and then click **OK**.

   The resource name is the identifier that service clients use to interact with view instance. For example, as the following figure shows, to name a collection based on a `DeptView1` view instance, you might name enter `Departments` to indicate the full collection name. For more information about naming, see What You May Need to Know About Naming ADF REST Resources.

   The release version is an identifier that you must create in the `adf-config.xml` file to manage the creation of resources in JDeveloper. The version will be part of the URL that service clients use to invoke operations on a specific REST resource, as described in Versioning the ADF REST Resource.

Figure 16-3 Assigning the ADF REST Resource to a Release Version



Create REST Resource

Name your resource and select the root view instance you want to expose in the REST web service.

Resource Name: Departments

Release Version: 11.0

Root View Instance:

JobsView1
EmployeesView2
DepartmentsView1

5. In the overview editor for the REST resource, you can proceed to edit the resource and its structure by performing any of the following tasks:

   a. Add and remove child resources defined by view link accessors. For details, see How to Edit the ADF REST Resource Definition.

   b. Add and remove resources defined by the same backing view instance. For details, see How to Add and Remove Resources Based on the Same View Instance.

   c. Add and remove attributes. For details, see How to Hide and Expose Attributes in the ADF REST Resource.

   d. Modify access to standard operations defined by HTTP methods, canonical links, row finder filters, and subtype view usages. For details, see Customizing the ADF REST Resource Representation and Methods.

6. Click **File** and then **Save All** to save the selections on this resource.

   JDeveloper saves the REST resource selections that you make in the overview editor in the resource definition file for the resource, as described in What Happens When You Create REST Resources in the ADF Business Components Project.

What Happens When You Create REST Resources in the ADF Business Components Project

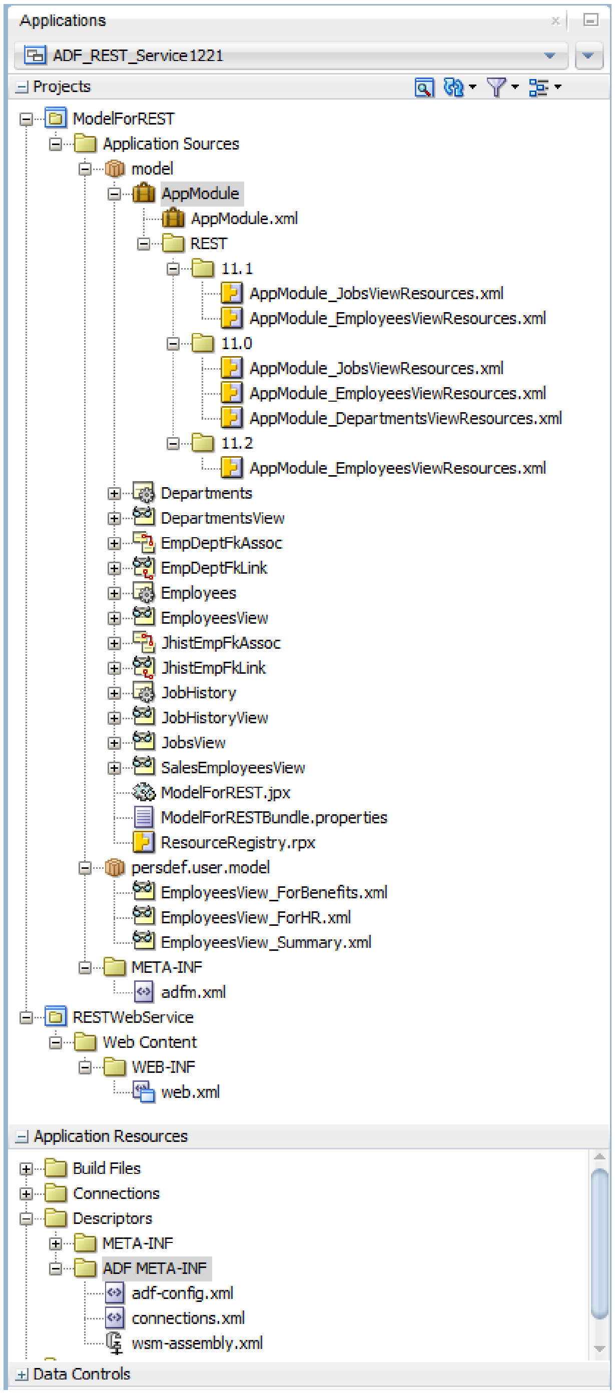
JDeveloper generates the REST resources and displays them in the Applications window organized in the model project by their version release names, as shown in Figure 16-4.

JDeveloper also generates a separate RESTful web service project (**RESTWebService**) to register the ADF REST servlet. Figure 16-4 shows the `web.xml` file that JDeveloper generates. You use this project to run and deploy the RESTful web service.

The `ResourceRegistry.rpx` file that appears in the model project uses metadata annotations to specifies the list of REST resources that will be exposed as a RESTful web service.

The `adf-config.xml` file is updated when you create the version release names. This file appears in the Applications window in the Application Resources panel, under the **Descriptors** and **ADF META-INF** nodes. The `connections.xml` file in the same folder is created when you first create the ADF Business Components.

Figure 16-4   REST Resources Appear Below Application Module Node



```
Applications
ADF_REST_Service1221
Projects
ModelForREST
  Application Sources
    model
      AppModule
        AppModule.xml
        REST
          11.1
            AppModule_JobsViewResources.xml
            AppModule_EmployeesViewResources.xml
          11.0
            AppModule_JobsViewResources.xml
            AppModule_EmployeesViewResources.xml
            AppModule_DepartmentsViewResources.xml
          11.2
            AppModule_EmployeesViewResources.xml
      Departments
      DepartmentsView
      EmpDeptFkAssoc
      EmpDeptFkLink
      Employees
      EmployeesView
      JhistEmpFkAssoc
      JhistEmpFkLink
      JobHistory
      JobHistoryView
      JobsView
      SalesEmployeesView
      ModelForREST.jpx
      ModelForRESTBundle.properties
      ResourceRegistry.rpx
    persdef.user.model
      EmployeesView_ForBenefits.xml
      EmployeesView_ForHR.xml
      EmployeesView_Summary.xml
    META-INF
      adfm.xml
RESTWebService
  Web Content
    WEB-INF
      web.xml
Application Resources
Build Files
Connections
Descriptors
  META-INF
  ADF META-INF
    adf-config.xml
    connections.xml
    wsm-assembly.xml
Data Controls
```

How to Edit the ADF REST Resource Definition

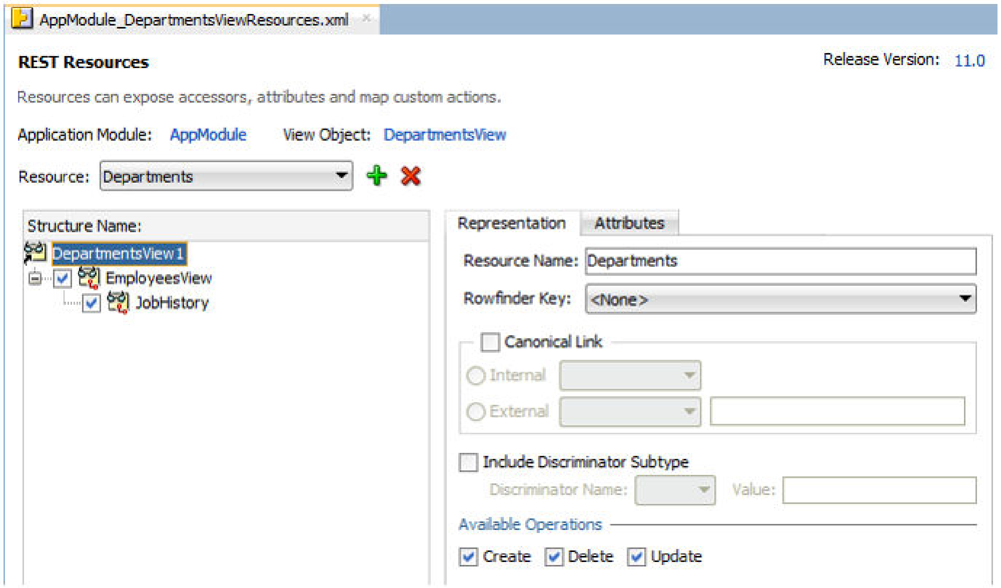
After you add an ADF REST resource definition to the ADF Model project, the REST section of the application module overview editor displays the names of the resources you have created. You can open the REST resource definition in its own editor by clicking the resource name in the list (as shown in Figure 16-2).

The overview editor for REST resources displays the options for editing the structure of the resource. From the standpoint of the RESTful web service, the REST resource structure determines the available data that may be invoked by REST calls at runtime.

At design time, you can edit the REST resource definition to select only the resources that you want to expose from the view object hierarchy backing the resource. When you first create a REST resource, you are required to select a view object exposed by the data model of the application module to back the resource. This original view object selection defines the root resource of the REST resource. You may optionally expose additional child view objects if present.

For example, the left side of the REST resource editor shown in Figure 16-5 displays the default structure of the resource, `Departments`. The resource structure appears as a tree with the root node representing the root resource (in this case `DepartmentsView1`). The leaf nodes of the tree represent child resources. In this example, the `Departments` resource is comprised of a root resource `DepartmentsView1` and various child resources, including `Employees` and `JobHistory`. Because the `DepartmentsView1`, `Employees`, and `JobHistory` resources are related as view instances in the data model, they appear in the hierarchy. You use the checkboxes beside leaf nodes to add and remove child resources from the resource structure.

Figure 16-5 Overview Editor Displays Resource Structure



AppModule_DepartmentsViewResources.xml

REST Resources

Release Version: 11.0

Resources can expose accessors, attributes and map custom actions.

Application Module: AppModule View Object: DepartmentsView

Resource: Departments

Structure Name:
DepartmentsView1
☑ EmployeesView
☑ JobHistory

Representation | Attributes

Resource Name: Departments
Rowfinder Key: <None>

□ Canonical Link
○ Internal
○ External

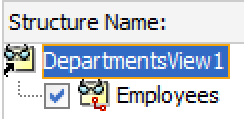
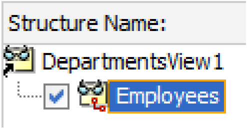
□ Include Discriminator Subtype
Discriminator Name: Value:

Available Operations
☑ Create ☑ Delete ☑ Update

> **Note:**
>
> The right side of the REST resource editor displays the available options corresponding to the node selections that you make in the **Structure** tree.

Table 16-1 summarizes the nodes of the resource structure tree:

Table 16-1 Nodes of the Resource Structure Tree

Node	Description	Editing Options
Root resource node:  Structure Name: DepartmentsView1 ☑ Employees	Represents the view object instance that backs the root resource selection.	The backing view object instance of the root resource cannot be altered or removed. You may select this node to select options on the root resource. For details about these options, see Customizing the ADF REST Resource Representation and Methods.
Child resource node:  Structure Name: DepartmentsView1 ☑ Employees	Represents a view link accessor used in the data model to define master-details relationships defined for the view object instance that backs the root resource selection. Note: Child nodes are not displayed when the view object instance of the root resource is not included in a master-detail relationship.	Child resources are disabled by default. Select the checkbox to expose a child resource in the resource representation. To remove a child resource from the resource representation, deselect the checkbox. You may select a child resource node to select options on the child resource. For details about these options, see Customizing the ADF REST Resource Representation and Methods.

Before you begin:

It may be helpful to have an understanding of how ADF REST supports application modules and enables web service clients to access rows of data and perform service operations. See About RESTful Web Services and ADF Business Components.

You may also find it helpful to understand other Oracle ADF features that depend on ADF REST resources. See Additional Functionality for RESTful Web Services.

You will need to complete this task:

- Create the desired REST resource, as described in Creating ADF REST Resources Using the Application Module.

To edit the ADF REST resource structure:

1. In the Applications window, double-click the resource definition file that defines the ADF REST resource you want to edit.
2. In the overview editor, make the desired selections.

   Press **F1** to view the JDeveloper Help Center topic for the overview editor.
3. Choose **File** and then **Save All** to save the changes on this resource.

JDeveloper saves your changes in the resource definition file for the resource, as described in What Happens When You Create REST Resources in the ADF Business Components Project.

How to Expose Child Resources in the ADF REST Resource

Child resources may be exposed in the ADF REST resource when the application module data model specifies one or more hierarchical, master-detail relationships. For each master-detail relationship in the data model, the ADF Business Components Model project must define a view link accessor that names the related parent and child view objects, thereby nesting the child or detail view object instance within the parent or master view object instance.

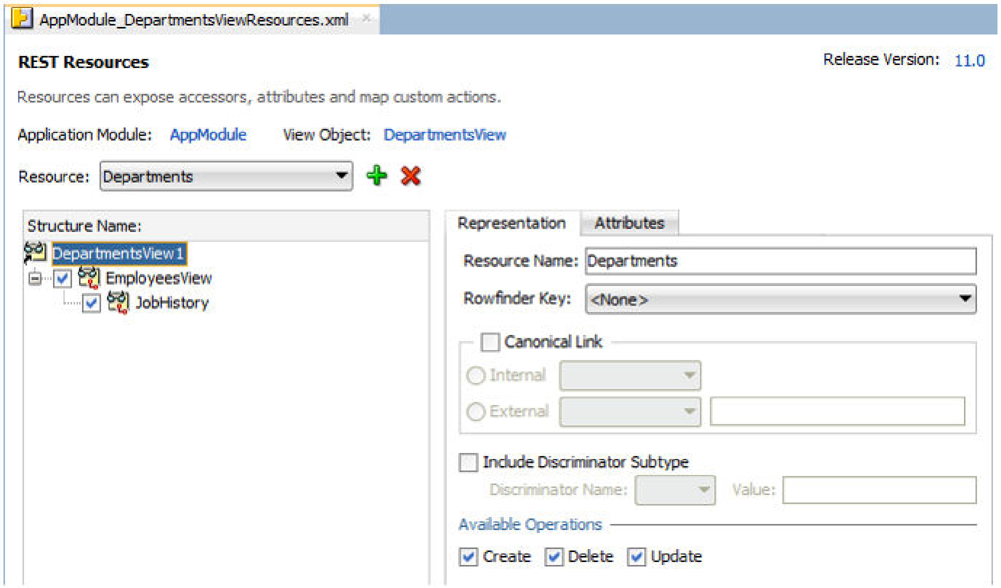
At runtime, a REST resource that exposes nested view instances supports returning collections for the master and detail in a single roundtrip. Thus, exposing child resources at design-time is useful for service clients with the requirement to manipulate master-detail data.

In JDeveloper, the overview editor for REST resources displays master-detail relationships as a resource structure tree with child resources displayed as leaf nodes of the root resource. For example, Figure 16-6 shows the `Departments` resource in the overview editor with the following nested resource structure:

- `DepartmentsView1` is the master view instance and backs the root resource.
- `Employees` is a detail view instance, nested below `DepartmentsView1` and the checkbox for the tree node appears selected, indicating the `Departments` resource exposes the child resource.
- `JobHistory` is another detail view instance, nested below `EmployeesView` and the checkbox for the tree node appears selected, indicating the `Departments` resource exposes the child resource.

By default, JDeveloper creates the REST resource based on the selected backing view instance, but does not include child resources. In the resource structure tree, child resources appear unselected by default to indicate that these resources are not exposed. You may expose or hide child resources, by selecting or unselecting its checkbox.

Figure 16-6 Child Resources Enabled in the Overview Editor



AppModule_DepartmentsViewResources.xml

REST Resources

Release Version: 11.0

Resources can expose accessors, attributes and map custom actions.

Application Module: AppModule View Object: DepartmentsView

Resource: Departments

Structure Name:

- DepartmentsView1
  - ☑ EmployeesView
    - ☑ JobHistory

Representation | Attributes

Resource Name: Departments

Rowfinder Key: <None>

□ Canonical Link

○ Internal

○ External

□ Include Discriminator Subtype

Discriminator Name: Value:

Available Operations

☑ Create ☑ Delete ☑ Update

> **Note:**
>
> By default, child resources are not exposed and must be selected in the overview editor to expose them in the resource.

Before you begin:

It may be helpful to have an understanding of how ADF REST supports application modules and enables service clients to access rows of data and perform service operations. For more information, see About RESTful Web Services and ADF Business Components.

You may also find it helpful to understand other Oracle ADF features that depend on ADF REST resources. For more information, see Additional Functionality for RESTful Web Services.

You will need to complete this task:

- Create the desired REST resource, as described in Creating ADF REST Resources Using the Application Module.

To expose nested resources:

1. In the Applications window, double-click the resource definition file that defines the ADF REST resource you want to edit.
2. In the overview editor for the REST resource, expand the **Structure** tree and select the checkbox of any child resource of the root resource.

   By default, child resource nodes appear unselected in the **Structure** tree, which indicates that nested resources are unexposed in the resource.

If the data model does not define a master-detail hierarchy for the backing view instance of the root resource, then no child resource node will appear in the **Structure** tree. For more information about **Structure** nodes, see Table 16-1.

3. Select the checkbox of another child resource in the **Structure** tree to enable its nested resource on the resource.

What You May Need to Know About Optimizing Child Resource Database Queries

REST requests where M-1 or 1-1 child resources render as nested objects can result in a high number of database queries, based on the number of resource items. For example, child Department resources rendered for an Employee object result in a database roundtrip for each employee. However, if the REST nested object request is made and the ADF REST framework discovers a M-1 or 1-1 child relationship involving the child's primary key under the following additional conditions, then lookup of child resources is optimized by finding a row in the view object's entity cache. If these conditions do not exist, a database query will be performed for the row fetches.

To ensure that the primary key lookup is used to improve the execution time for fetching a one-side view link, nested entity-backed view objects must meet the following conditions:

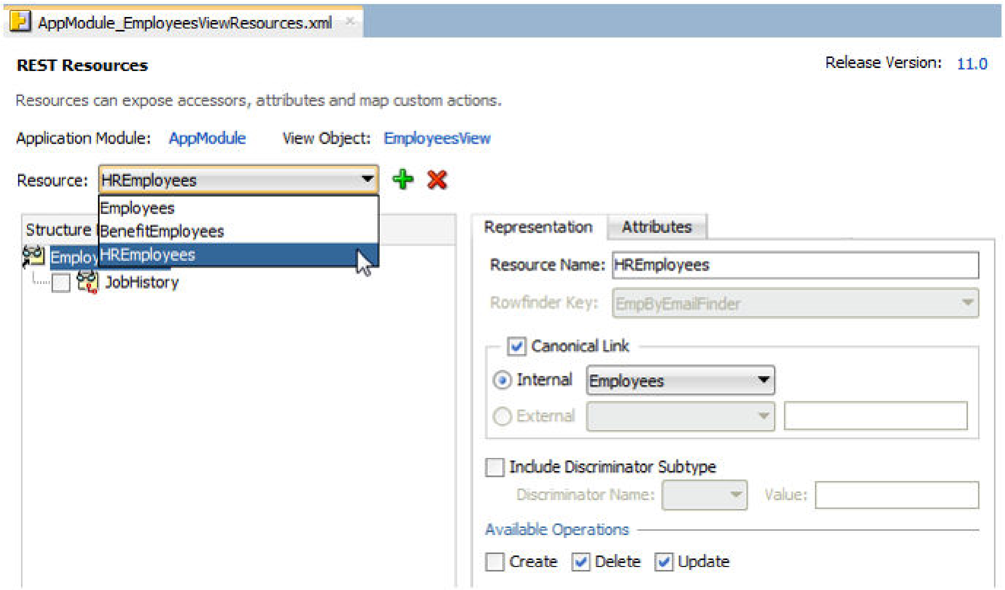
- The view object has an entity reference to a child view object and includes the attributes that the child resource exposes. For example, the employee's department name or manager name.
- The view link must be based on an association.
- The association attributes for the other side are primary keys.
- The association does not have any custom where clauses.
- The view object for the other side does not have any custom where clauses or applied database-only view criteria.

How to Add and Remove Resources Based on the Same View Instance

In JDeveloper, you can use the overview editor for REST resources to define multiple resources for the same view instance. You might add a resource when a single resource definition does not support all of the use cases of your service client. After you add a resource you will be able to modify the operations, list of attributes, and other items of the resource structure to support the desired use cases. Because each resource you define represents a unique resource collection to the service client, resource names should reflect the differences.

The overview editor for REST resources displays the list of resources you add in the **Resource** dropdown. For example, Figure 16-7 shows the `Employees` resource and the `EmployeesAllAttrs` resource in the overview editor for the same backing view object `EmployeesView`.

Figure 16-7 Resource List Supports Multiple Resource in the REST Resource Definition


AppModule_EmployeesViewResources.xml

REST Resources

Release Version: 11.0

Resources can expose accessors, attributes and map custom actions.

Application Module: AppModule View Object: EmployeesView

Resource: HREmployees

Resource Name: HREmployees

Rowfinder Key: EmpByEmailFinder

☑ Canonical Link

● Internal Employees

○ External

□ Include Discriminator Subtype

Discriminator Name: Value:

Available Operations

□ Create ☑ Delete ☑ Update

> **Note:**
>
> JDeveloper lets you remove resource definitions that have been added in the overview editor for REST resources. However, the empty definition file will appear in the Applications window.

Before you begin:

It may be helpful to have an understanding of how the ADF REST framework supports application modules and enables service clients to access rows of data and perform service operations. For more information, see About RESTful Web Services and ADF Business Components.

You may also find it helpful to understand other Oracle ADF features that depend on ADF REST resources. For more information, see Additional Functionality for RESTful Web Services.

You will need to complete this task:

- Create the REST resource for which you want to add or remove a resource, as described in Creating ADF REST Resources Using the Application Module.

To add and remove resources in the overview editor:

1. In the Applications window, double-click the resource definition file that defines the ADF REST resource for which you want to add or remove resources.
2. In the overview editor for the REST resource, choose the resource from the **Resource** dropdown and then click **Delete Resource** to delete a resource that has already been defined for the view object.

If you delete the root resource definition using **Delete Resource**, the resource is deleted, but the empty definition file will appear in the Applications window.

3. In the overview editor for the REST resource, to add a resource that you will create for the view object, click **Add Resource**.
4. In the Add Resource for View Object dialog, enter the **Resource Name** of the REST resource collection that you want to add for the selected root view instance and then click **OK**.
5. In the overview editor for the REST resource, select the new resource from the **Resource** dropdown to edit its resource definition:
   a. Add and remove child resources defined by view link accessors. For details, see How to Edit the ADF REST Resource Definition.
   b. Add and remove attributes. For details, see How to Hide and Expose Attributes in the ADF REST Resource.
   c. Modify access to standard operations defined by HTTP methods, canonical links, row finder filters, and subtype view usages. For details, see Customizing the ADF REST Resource Representation and Methods.

How to Hide a Resource from the Catalog Describe

You can hide a top-level resource from the ADF REST catalog describe by setting the `Visibility` attribute within the `ServiceConfiguration` tag in the resource definition as shown in the following code example.

```
<tree..>
  <ServiceConfiguration Visibility="unlisted"/>
   ...
</tree>
```

You can set the `Visibility` attribute to `unlisted` or `public`. You can use the `include` query parameter to include the description of resources of a certain visibility. For more information, see Retrieving the Catalog Describe Based on Resource Visibility Declaration.

How to Rename the ADF REST Resource

During development of ADF REST resources you may use the overview editor for REST resources to change the name of a resource in the resource definition file. Because service clients use the resource names to interact with the desired REST resource collections, it is important that you finalize the names of ADF REST resources before deploying the service to a production environment.

When you need to change the name of an ADF REST resource *after* deployment, this is best handled by creating an entirely new version of the resource, as described How to Version the ADF REST Resource.

Before you begin:

It may be helpful to have an understanding of how the ADF REST framework supports application modules and enables service clients to access rows of data and perform service operations. For more information, see About RESTful Web Services and ADF Business Components.

You may also find it helpful to understand other Oracle ADF features that depend on ADF REST resources. For more information, see Additional Functionality for RESTful Web Services.

You will need to complete this task:

- Create the REST resource, as described in Creating ADF REST Resources Using the Application Module.

To rename a resource before production deployment:

1. In the Applications window, double-click the resource definition file that defines the ADF REST resource you want to rename.
2. In the overview editor for the REST resource, choose the desired resource from the **Resource** dropdown.

   If you have not created additional resources on the resource definition file, the list displays the original resource created for the backing view object.
3. Click the **Representation** tab of the overview editor and enter the new name in the **Resource Name** field.

What You May Need to Know About Naming ADF REST Resources

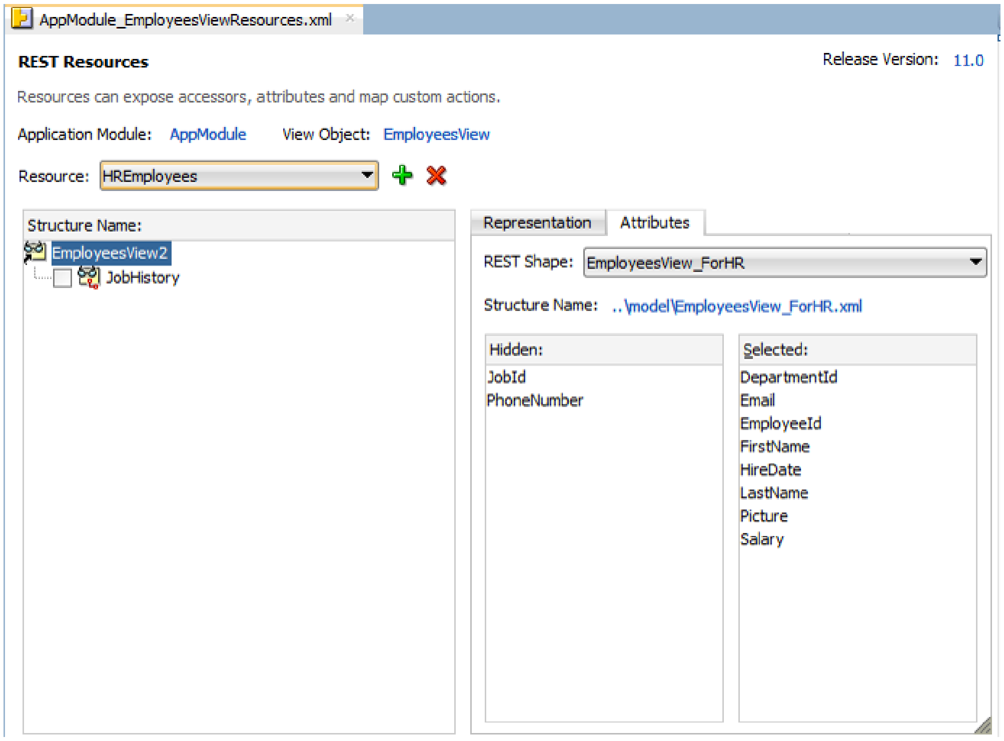
The REST resource name is the way service clients identify and interact with exposed resource collections. REST resource names typically identify the view object the consuming client is to access. As such, REST resource names are nouns, not verbs.

How to Hide and Expose Attributes in the ADF REST Resource

In JDeveloper, you can use the overview editor for REST resources to define multiple resources for the same view instance. You might add a resource when a single resource definition does not support all of the use cases of your service client. After you add a resource, you can modify the list of exposed attributes, and other items of the resource structure, as needed to support the desired use cases.

The overview editor for REST resources displays the list of exposed attributes in the **Attributes** tab. The list is populated by a REST service shape selection that you define on the view object backing the root resource. For example, Figure 16-8 shows the service shape definition `EmployeesView_ForHR` that exposes all but two of the attributes for the `EmployeesView` view object.

Figure 16-8 Attributes Exposed By REST Service Shape Definition Selection



AppModule_EmployeesViewResources.xml

REST Resources

Release Version: 11.0

Resources can expose accessors, attributes and map custom actions.

Application Module: AppModule View Object: EmployeesView

Resource: HREmployees

Structure Name:

- EmployeesView2
  - □ JobHistory

Representation | Attributes

REST Shape: EmployeesView_ForHR

Structure Name: ..\model\EmployeesView_ForHR.xml

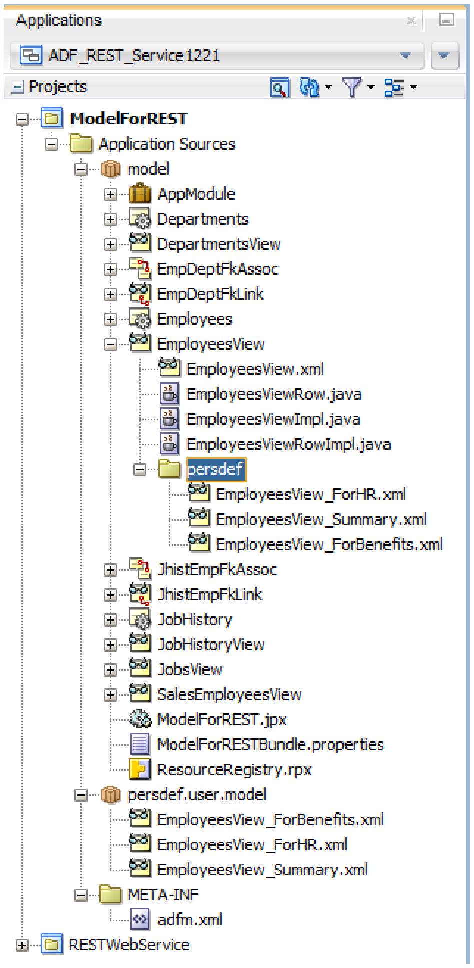
Hidden:	Selected:
JobId	DepartmentId
PhoneNumber	Email
	EmployeeId
	FirstName
	HireDate
	LastName
	Picture
	Salary

Note:

You do not directly modify the list of exposed attributes in the overview editor for REST resources. Rather, you must create service shape definitions on the backing view object, and you then apply the desired **REST Shape** definition to the current **Resource** dropdown selection.

To support the use cases of service clients, you can define as many service shape definitions for each view object as required. The Applications window displays service shape definitions that you create for the view object in the **persdef** folder under the view object node. For example, Figure 16-9 shows two service shape definition files **EmployeesView_forHR.xml** and **EmployeesView_ForBenefits.xml** beneath the **EmployeesView** view object.

Figure 16-9 REST Service Shape Definition Files in the Applications Window



Before you begin:

It may be helpful to have an understanding of how the ADF REST framework supports application modules and enables service clients to access rows of data and perform service operations. For more information, see About RESTful Web Services and ADF Business Components.

You may also find it helpful to understand other Oracle ADF features that depend on ADF REST resources. For more information, see Additional Functionality for RESTful Web Services.

You will need to complete this task:

- Create the REST resource, as described in Creating ADF REST Resources Using the Application Module.

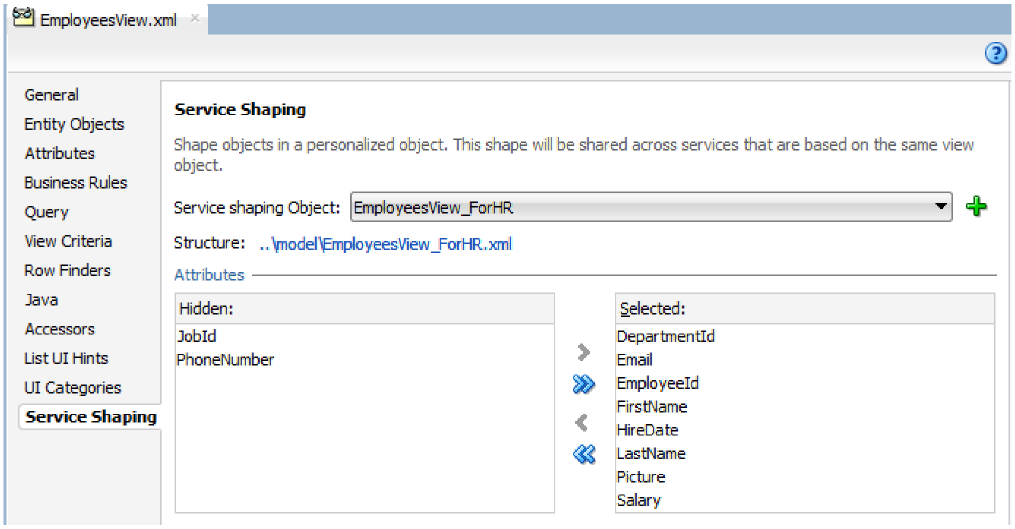
To apply a REST service shape definition to a REST resource:

1. In the Applications window, double-click the resource definition file that defines the ADF REST resource you want to edit.
2. In the overview editor for the REST resource, click the **View Object** link.
3. In the overview editor for the view object, click the **Service Shape** navigation tab.

4. In the Service Shaping page, click the **Create Service Shaping Object** button and then in the Create Service Shape dialog, enter a suffix name to uniquely identify the service shape definition.

   The suffix you enter forms the name of the service shape definition file. For example, the suffix `ForHR` is added to the view object `EmployeesView` to form the service shape definition name `EmployeesView_ForHR`, as shown in Figure 16-10.

Figure 16-10 View Object Service Shape Definition Defines Available Attributes



EmployeesView.xml

Service Shaping

Shape objects in a personalized object. This shape will be shared across services that are based on the same view object.

Service shaping Object: EmployeesView_ForHR

Structure: ..\model\EmployeesView_ForHR.xml

Attributes

Hidden:	Selected:
JobId	DepartmentId
PhoneNumber	Email
	EmployeeId
	FirstName
	HireDate
	LastName
	Picture
	Salary

5. In the Service Shaping page, use the shuttle buttons to alter the list of **Hidden** and **Selected** attributes and save the view object definition.

   When you alter the service shape in the overview editor for REST resources, JDeveloper updates the service shape definition file for the view object. The resource structure of any ADF REST resource that references the service shape will be updated as well.

6. In the overview editor for the REST resource, select the desired resource from the **Resource** dropdown.

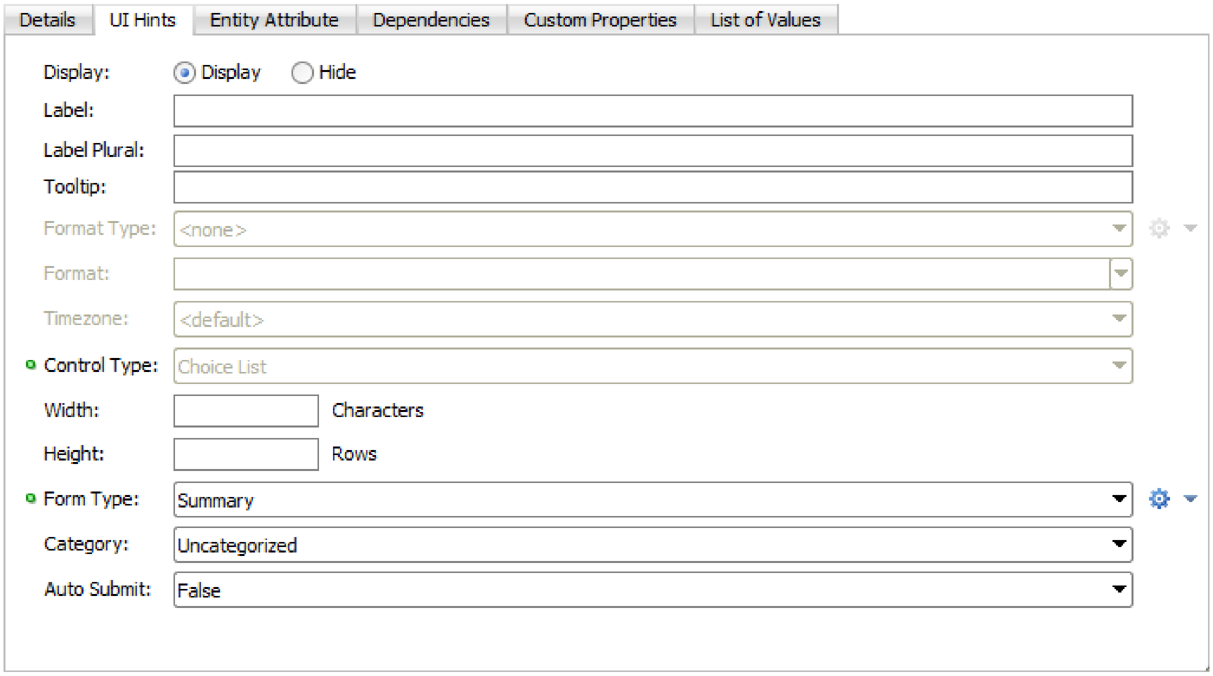
7. Click the **Attributes** tab of the overview editor, and then select the **Service Shaping Object** that defines the list of attributes that you want to expose.

What You May Need to Know About ADF REST Attribute Hints

In JDeveloper, you can use the overview editor to set ADF REST attribute hints.

Setting the attribute hints effects the REST service describe payload. They appear as attribute properties in the `json` payload.

Figure 16-11   UI Hints Editor



Display: ● Display ○ Hide

Label:

Label Plural:

Tooltip:

Format Type: <none>

Format:

Timezone: <default>

Control Type: Choice List

Width: Characters

Height: Rows

Form Type: Summary

Category: Uncategorized

Auto Submit: False

As Figure 16-11 shows, in the overview editor for the view object, you can set UI Hints for the attributes. If you set the `Form Type` property to `Summary`, the `includeInCompactView=true` hint will be surfaced on the attribute. After you have set the `From Type` hint for the attribute, it will suggest as a hint to metadata driven UIs to include the attribute in a compact view.

How to Control the ADF REST Response Payload Fields

The fields returned in a response payload for a GET request are limited by using the `?fields` URL parameter. If you do not specify the `?fields` URL parameter, it might result in a response of larger size. You can control the fields that are returned in the response payload by setting the `Form Type` hint of an attribute either to `Detail` or `Summary`. The default value is `Detail`.

When you set the `Form Type` hint of an attribute to `Summary`, the `includeInCompactView=true` hint will be surfaced on the attribute. See What You May Need to Know About ADF REST Attribute Hints.

You can use the `prefer` HTTP header either to return all the fields in the response payload or only the most common fields by using one of the following `return` options:

- **minimal**: returns only a minimal response to a successful request. This option returns only those attributes in the payload for which `includeInCompactView=true` is set. The minimal response is at the discretion of the server.
- **representation**: returns the current state of the resources in the response to a successful request. This option returns all the fields irrespective of the attributes set. A POST or PATCH request with `return=representation` will return all fields.

```
Content-Type: application/vnd.oracle.adf.resourceitem+json
Prefer: return=representation/minimal
```

What You May Need to Know About Response Payload When a Shape is Defined on a Resource

When you define a compact shape to a REST resource and do not specify `?fields` in the URL, then only those attributes for which the `Form Type` hint is set to `Summary` are returned in the payload.

For example, if you define a REST service shape on a root or child resource, only the fields that support `Form Type=summary` in the REST service shape are returned when you do not specify the `?fields` parameter.

The response payload for a REST resource is controlled at the resource level only if a resource has the `EnableCompactview=view` configuration property set.

```
<tree IterBinding="EmpIter" id="employees" AccessorFolder="Always">
    <ServiceConfiguration EnableCompactView="true"/>
<nodeDefinition .../>
</tree>
```

A REST client can expand or contract the default set of fields by using the + and - operators in the fields parameter.

The following example returns the PhoneNumber field in the response in addition to all fields in the default shape.

```
/employees?fields=+PhoneNumber
```

The following example returns all the fields in the default shape in response, except the email field.

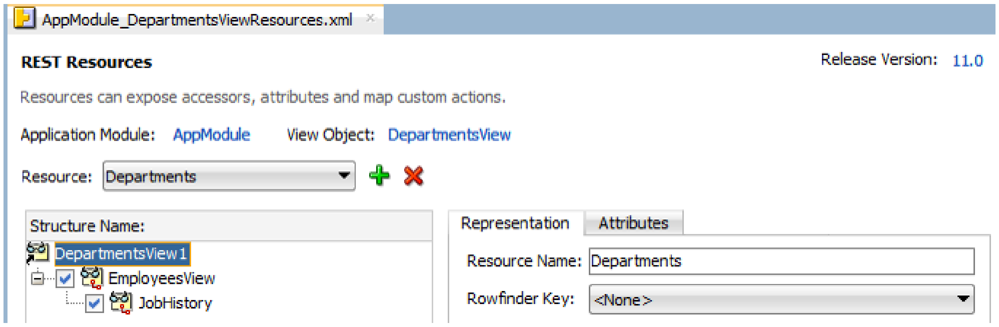
```
/employees?fields=-Email
```

Use the `Prefer` REST HTTP header to toggle between full set of fields and the default set. See How to Control the ADF REST Response Payload Fields.

What You May Need to Know About Modifying the ADF REST Resource Structure

As shown in Figure 16-12, after you create the resource and assign a version identifier, the overview editor for the REST resource displays the version in the top, right corner of the editor to help you identify which version you are editing.

Figure 16-12   REST Resource Editor Displays Version



Release Version: 11.0

Resources can expose accessors, attributes and map custom actions.

Application Module: AppModule   View Object: DepartmentsView

Resource: Departments

Resource Name: Departments

Rowfinder Key: <None>

Changes that you make to ADF REST resources during the development cycle can be made without the need to version the resource. However, once you have deployed the RESTful web service, service clients that access the ADF REST resources rely on the stability of the resource collection to satisfy their use cases.

Over time, as use cases change or the backing view objects of the REST resources change, you may need to modify the resource structure of already deployed resources. At that time, modifications to existing ADF REST resources may become backward-incompatible with the deployed REST service and may necessitate creating a new version identifier and ADF REST resource for that version. For details about when to version the REST resource, see What You May Need to Know About Versioning ADF REST Resources.

JDeveloper lets you manage resource versions in the Release Versions page of the overview editor for the `adf-config.xml` file in the ADF META-INF folder. For details about how to create a new version of ADF REST resources, see Versioning the ADF REST Resource.

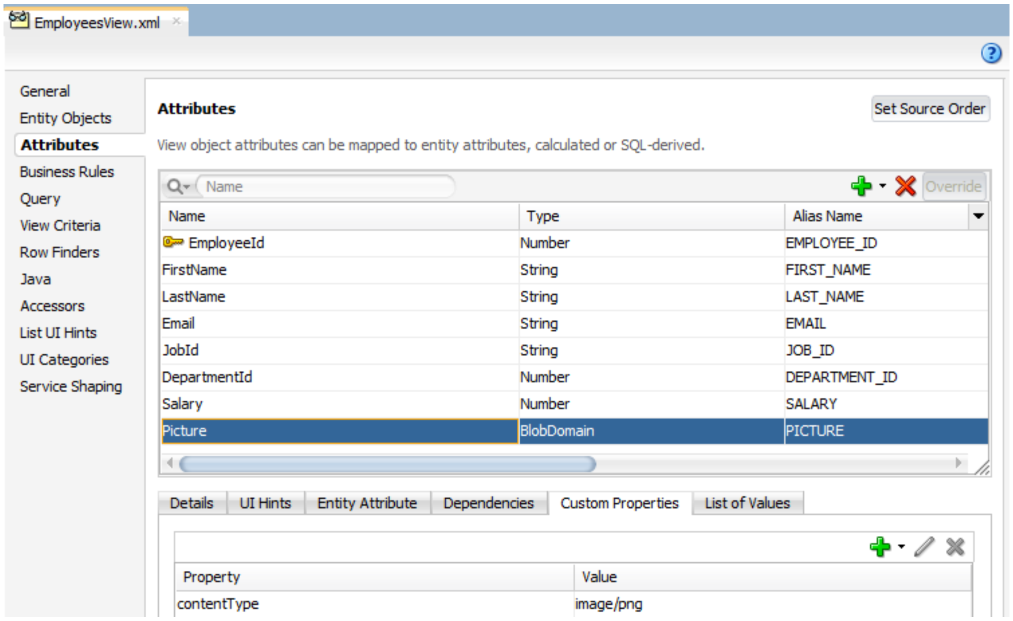
What You May Need to Know About ADF REST Resources and LOB Attributes

A resource that exposes a BLOB or CLOB type attribute will be accessible at runtime as streaming media. By default, the view object attribute will be handled by the ADF REST runtime using the generic content type `application/octet-stream`. When you want to support a specific content type, you can configure the special property `contentType` as a custom property of the LOB-type attribute. For example, when working with PNG image files, the following content type can be configured at design time.

- **Custom Property**: `contentType`
- **Value**: `image/png`

As Figure 16-13 shows, the custom property `contentType` has been added to the view object definition for the `Picture` attribute in the overview editor for the view object.

Figure 16-13 Custom Content Type Configured for LOB Attributes



Attributes

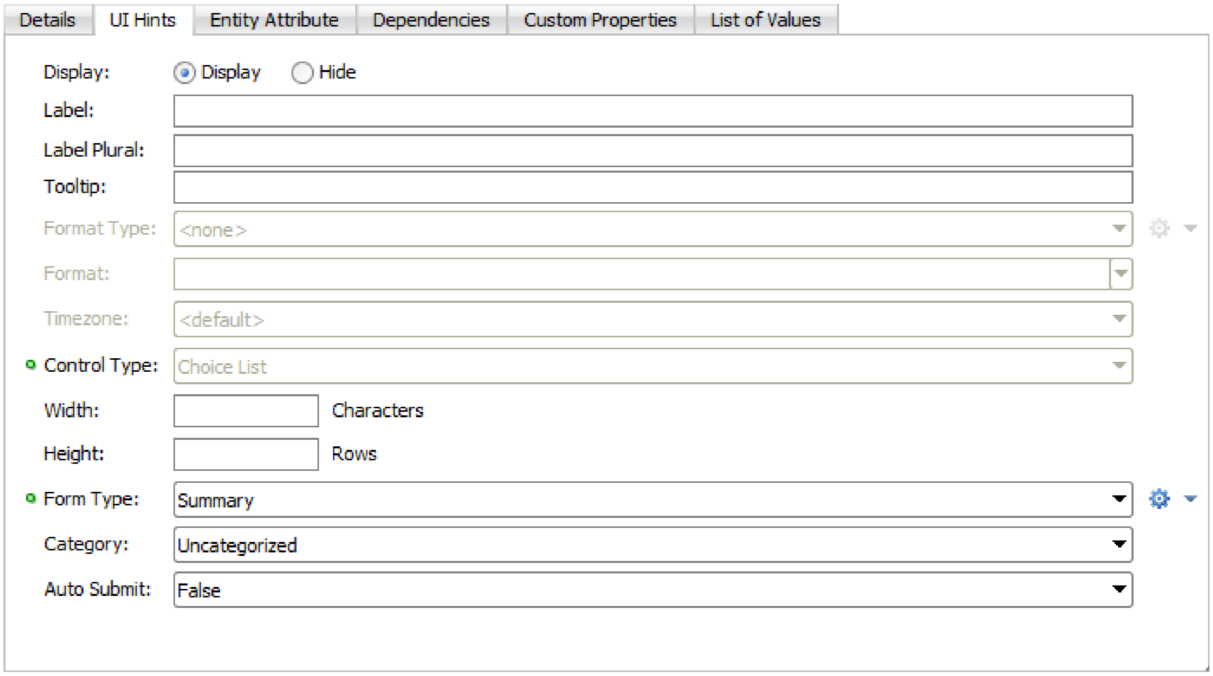
View object attributes can be mapped to entity attributes, calculated or SQL-derived.

Name	Type	Alias Name
EmployeeId	Number	EMPLOYEE_ID
FirstName	String	FIRST_NAME
LastName	String	LAST_NAME
Email	String	EMAIL
JobId	String	JOB_ID
DepartmentId	Number	DEPARTMENT_ID
Salary	Number	SALARY
Picture	BlobDomain	PICTURE

Property	Value
contentType	image/png

As Figure 16-14 shows, in the overview editor for the view object, you can set UI Hints for the attributes. If you set the **Form Type** hint to `Summary`, the `includeInCompactView=true` hint will be surfaced on the attribute. After you have set the **Form Type** hint for the attribute, it will suggest as a hint to metadata driven UIs to include the attribute in a compact view.

Figure 16-14 UI Hints Editor



Display: ● Display ○ Hide

Label:

Label Plural:

Tooltip:

Format Type: <none>

Format:

Timezone: <default>

Control Type: Choice List

Width: Characters

Height: Rows

Form Type: Summary

Category: Uncategorized

Auto Submit: False

What You May Need to Know About LOV Accessors in the ADF REST Resource

A list of value (LOV) attribute is any attribute of a view object that relies on another view object as its data source. Typically, the data source view object specifies a fixed list of possible values for the LOV-enabled attribute. When you explicitly define an LOV-enabled attribute on a view object, ADF Business Components defines an LOV view accessor that the ADF REST runtime will automatically expose in the ADF REST resource as a LOV child resource. Thus, at runtime when an LOV-enable attribute is detected on a resource, service clients may invoke LOV view accessors to obtain the data source of an LOV-enabled attribute and present selection choices to the end user.

Note that the LOV attribute must not rely on an LOV view accessor that references an entity object attribute. Only LOV view accessors based on a view object attribute will be visible to the REST service. At runtime, ADF REST knows nothing about attributes and links defined at the level of entity objects and will return an HTTP error code 500 when the resource contains an LOV attribute defined by an LOV view accessor that references an entity object attribute.

> **Note:**
>
> By default, LOV accessor resources are exposed on resources and are not displayed in the structure tree of the resource overview editor. For details about creating LOV-enabled attributes, see Working with List of Values (LOV) in View Object Attributes.

What You May Need to Know About Mandatory Attributes in the ADF REST Resource

A mandatory attribute is any attribute of a view object that must be supplied in a query and if not supplied, will trigger a validation error at runtime. The ADF REST resource describe identifies whether or not an attribute is mandatory using the property `“mandatory”: true` or `“mandatory”:false`. The mandatory property of the resource describe is based on the view object attribute metadata specified at design time. When the attribute is set as mandatory and the setting is determined at runtime, using a Groovy expression to evaluate to `true` or `false`, then the describe will return `“isMandatoryConditional”:true` in the attribute properties list.

What You May Need to Know About Case Sensitivity in Resource Collection Query Operations

The ADF REST framework supports case insensitive search when using a `q` query parameter to filter search results. The flag named `restV1QueryCaseSensitive` in the `adf-config.xml` file can be set to `false` if you want to perform case insensitive search. The default value of the flag is `true` which implies case sensitive search.

Customizing the ADF REST Resource Representation

JDeveloper allows you to edit the ADF REST resource definition. You can enable HTTP actions, expose a row finder, enable a canonical resource link for nodes of the tree, expose subtype view objects in the resource, and configure the resource cache duration.

You use the overview editor for the ADF REST resource to edit the ADF REST resource definition. For example, selections that you make in the resource structure tree let you enable HTTP actions, expose a row finder, and enable a canonical resource link for each node of the tree.

The customization choices that you make for REST resources in your ADF Model project ultimately affect the ability of service clients to interact with these resources at runtime. Therefore it is important to understand the requirements of the service client use cases to support the desired ADF REST runtime features though the various design time customizations described in this section.

For details about the ADF REST runtime, see Consuming ADF RESTful Web Services.

> **Note:**
>
> Custom properties defined for a resource attribute are included in the describe. However, custom properties whose name begins with `INTERNAL`, `internal`, or __ (double underscore), are hidden from the describe.

How to Expose Canonical Resources in the ADF REST Resource

The ADF REST resource representation limits the attributes returned in a runtime response to the attributes queried by the resource's backing view object. When defining the view object query at design time, it is therefore important to expose only the attributes required to support specific use cases of the service client. However, in order to allow the service client to navigate from the partial REST resource representation to the full representation of the same data, you can configure a link to a canonical resource.

Typically, the canonical resource is considered as the source of truth and therefore specifies the full representation of a given data source. An example of a canonical resource, can be illustrated by the `employees` resource with a link to a resource that describes a `manager` instance. In this scenario, where typical use cases may require only the manager name, the non-canonical `manager` resource includes only the name attribute. And, for cases where the name is not sufficient to identify the manager, the non-canonical resource may link to a canonical resource that exposes all attributes of the `manager` instance, including name, id, department, hire date, and so on. Thus, the canonical resource and the non-canonical resource exist as follows:

- Non-canonical resources and the canonical resource expose different representations of the same data source.
- Non-canonical resources expose only a representation of the data source needed to support service client use cases and are therefore a partial representation.

- Non-canonical resources include a link to its canonical, full representation when defined.

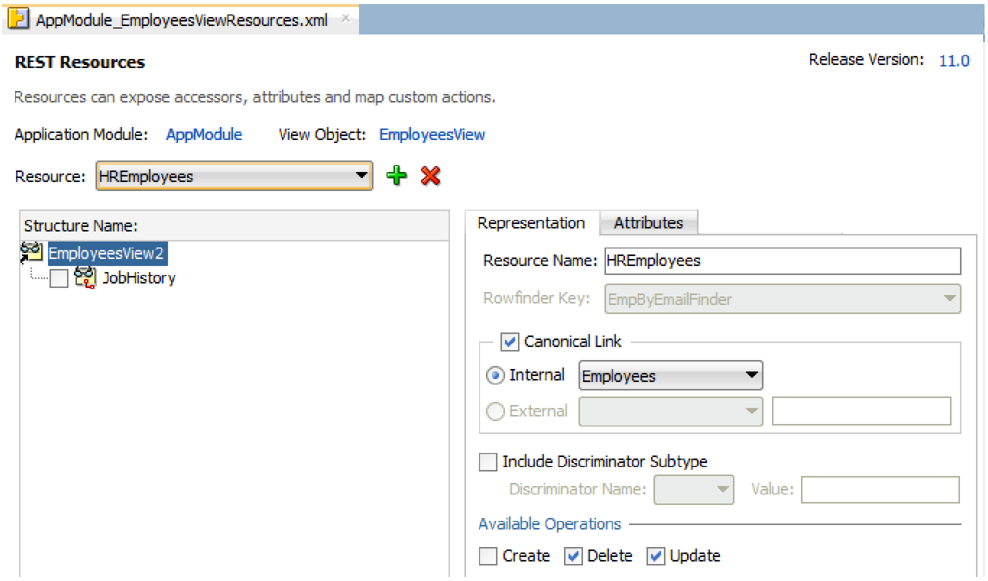
In the overview editor for the resource, for each resource node in the resource structure tree, you may choose a canonical resource from an existing REST resource defined by the ADF Model project or you may choose an external resource defined by an existing URL connection in the `connections.xml` file of the application.

The internal canonical resource that you can select in the overview editor for REST resources, must meet the following criteria:

1. Your ADF Model project must define a view object that accesses the same data source backing the existing, partial representation resource. The view object may or may not be entity-based.
2. The view object must query a superset of available attributes queried by the non-canonical resources.
3. The superset representation view object must be specified as a view instance on the data model of the application module.

Figure 16-15 shows the `EmployeesView` root resource node selection in the overview editor and the Resource dropdown selection `HREmployees`, with the internal canonical resource `Employees` enabled. In this case, the database table specified by each backing view object is the same for both the `HREmployees` and the `Employees` resource. In this example, `Employees` is the canonical resource and contains all attributes from the backing view object whereas the `HREmployees` resource is backed by the same view object but queries only the attributes required to support a specific use case.

Figure 16-15 Canonical Resource Enabled in the Overview Editor



AppModule_EmployeesViewResources.xml

REST Resources

Release Version: 11.0

Resources can expose accessors, attributes and map custom actions.

Application Module: AppModule View Object: EmployeesView

Resource: HREmployees

Structure Name:
EmployeesView2
□ JobHistory

Resource Name: HREmployees
Rowfinder Key: EmpByEmailFinder

☑ Canonical Link
● Internal Employees
○ External

□ Include Discriminator Subtype
Discriminator Name: Value:

Available Operations
□ Create ☑ Delete ☑ Update

> **Note:**
>
> A resource (resource_partial) can only select another resource as canonical (resource_full) as long as the resource_full contains all the attributes that resource_partial exposes.

Before you begin:

It may be helpful to have an understanding of how the ADF REST framework supports application modules and enables service clients to access rows of data and perform service operations. For more information, see About RESTful Web Services and ADF Business Components.

You may also find it helpful to understand other Oracle ADF features that depend on ADF REST resources. For more information, see Additional Functionality for RESTful Web Services.

You will need to complete these tasks:

- Create the desired REST resource, as described in Creating ADF REST Resources Using the Application Module.
- When you want the REST resource's canonical link to expose an internal canonical resource, create a view object definition that queries the superset of available attributes from the data source (must be the same data source as the non-canonical REST resource) and expose that view object in the data model, then create a resource for the internal canonical resource. For details about creating a resource, see How to Create ADF REST Resources from View Object Instances.
- When you want the REST resource's canonical link to expose an external canonical resource, create a URL connection in the application connection file (`connections.xml`), as described in the “Create URL Connection Dialog” topic in the JDeveloper Online Help.

To expose a canonical resource link:

1. In the Applications window, double-click the resource definition file that defines the ADF REST resource you want to edit.
2. In the overview editor for the REST resource, choose the desired resource from the **Resource** dropdown, expand the **Structure** tree and select the node for which you want to define a canonical resource link.

   You can define a canonical link for each resource node in the resource tree. For more information about **Structure** nodes, see Table 16-1.
3. With the resource node selected in the **Structure** tree, click the **Representation** tab of the overview editor, and then select **Canonical Link**.

   If you do not want to expose a canonical resource for the current resource accessor node selection, deselect **Canonical Link**.
4. In the **Canonical Link** section, select whether the canonical resource is **Internal** or **External** and choose the desired resource from the dropdown list to represent the canonical version of the resource.

The dropdown list for internal resources is populated by the resources that you added to the current resource; these are resources that are internal to the application and must be based on the same data source as the current resource.

The dropdown list for external resources is populated by URL connections that you have defined for the application and registered in the application connection file (`connections.xml`).

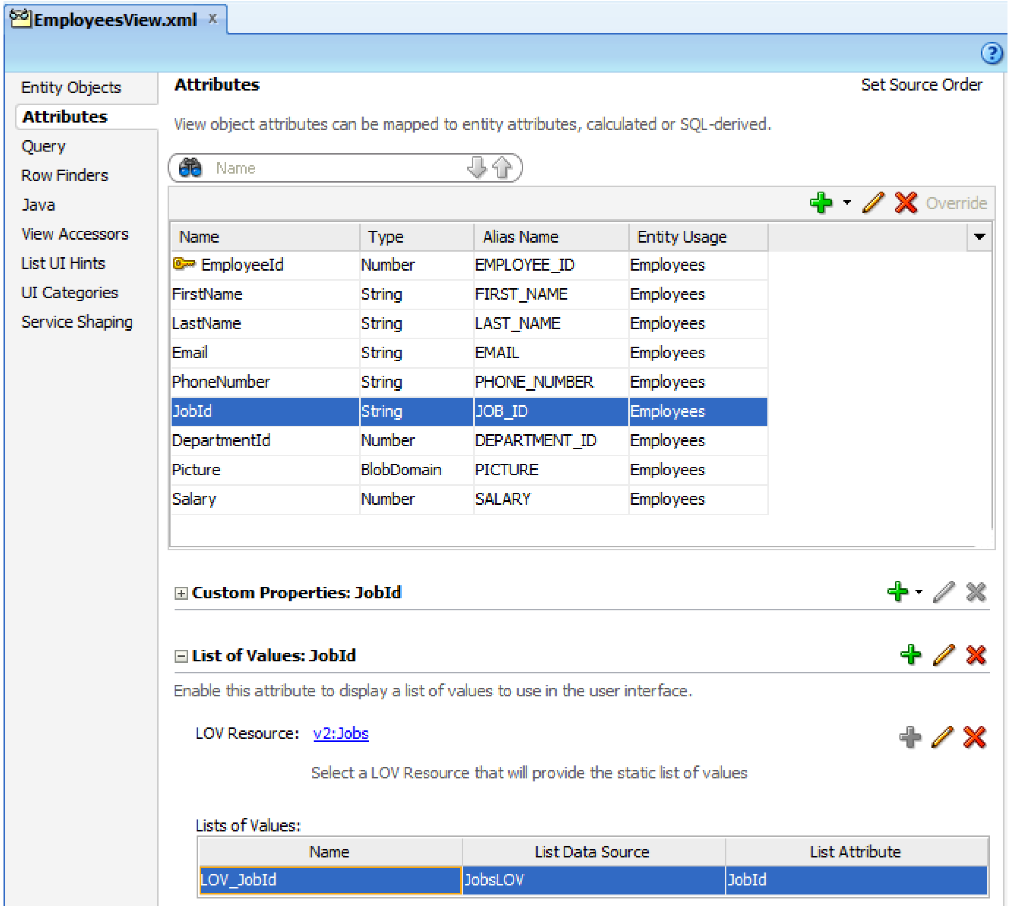
How to Support Create Operations on ADF REST Resources with LOV Attributes

When you want the ADF REST resource to support resource operations that require a list of values (LOV) without a row context to populate the list, you must create a LOV resource. For example, a service client may invoke a resource item create operation on the REST resource, in which case no row context will be available for the LOV list. In this situation, the LOV resource does not exist as a child resource and must instead be accessed at runtime by the service client independent from resource items that they attempt to create.

> **Note:**
>
> No design time changes are required to support resources with LOV-enabled attributes when the service client needs to access *existing* resource items. It is only necessary to create an LOV resource at design time when you want to enable service clients to create new resource items and the resource item is backed by an LOV-enabled attribute. In the case of existing resource items, the resource item provides a row context within the resource collection that allows the ADF REST runtime to automatically generate LOV links as a child resource. For more information about LOV child resources, see What You May Need to Know About LOV Accessors in the ADF REST Resource.

In JDeveloper, you use the overview editor for view objects to create LOV definitions for individual attributes in the Attributes page of the editor. The same page is used to define the static LOV resource. For example, Figure 16-16 shows the `EmployeesView` view object and the LOV-enabled attribute `JobId` in the overview editor for the view object backing the `Employees` resource. The static LOV resource is identified by the specific ADF REST resource that it supports: in this case, `v2` (the internal version identifier of the `Jobs` resource). Below the static LOV resource is the LOV definition used by the LOV-enabled attribute of the view object.

Figure 16-16 Static LOV Resource Supports Create Operations on the REST Resource



Attributes

View object attributes can be mapped to entity attributes, calculated or SQL-derived.

Name	Type	Alias Name	Entity Usage
EmployeeId	Number	EMPLOYEE_ID	Employees
FirstName	String	FIRST_NAME	Employees
LastName	String	LAST_NAME	Employees
Email	String	EMAIL	Employees
PhoneNumber	String	PHONE_NUMBER	Employees
JobId	String	JOB_ID	Employees
DepartmentId	Number	DEPARTMENT_ID	Employees
Picture	BlobDomain	PICTURE	Employees
Salary	Number	SALARY	Employees

⊞ Custom Properties: JobId

⊟ List of Values: JobId

Enable this attribute to display a list of values to use in the user interface.

LOV Resource: v2:Jobs

Select a LOV Resource that will provide the static list of values

Lists of Values:

Name	List Data Source	List Attribute
LOV_JobId	JobsLOV	JobId

Before you begin:

It may be helpful to have an understanding of how the ADF REST framework supports application modules and enables service clients to access rows of data and perform service operations. For more information, see About RESTful Web Services and ADF Business Components.

You may also find it helpful to understand other Oracle ADF features that depend on ADF REST resources. For more information, see Additional Functionality for RESTful Web Services.

You will need to complete this task:

- Enable an LOV for the attribute of the view object, as described in Creating ADF REST Resources Using the Application Module.
- Create the REST resource for the view object with the LOV-enabled attribute for which you want to expose a create operation, as described in Creating ADF REST Resources Using the Application Module.

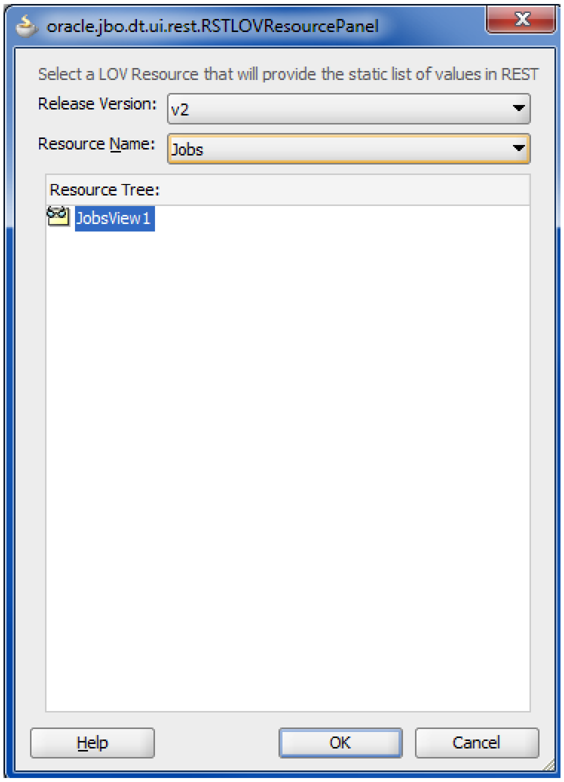
> **Caution:**
>
> When the view object defines an LOV resource, the view object service shape definition must not hide attributes that are required by the LOV-enabled attribute's LOV definition. Required LOV attributes include attributes mapped to the LOV data source, as well as LOV display attributes. For more information about service shaping and ADF REST resources, see How to Hide and Expose Attributes in the ADF REST Resource.

To define a static LOV resource for use with ADF REST resources:

1. In the Applications window, double-click the resource definition file that defines the ADF REST resource you want to edit.
2. In the overview editor for the REST resource, click the **View Object** link.
3. In the overview editor for the view object, click the **Attributes** navigation tab.
4. In the Attributes page, expand the LOV section and click the **Create LOV Resource on Attribute** button.
5. In the Create LOV Resource dialog, select the **Release Version** identifier, select the **Resource Name** for the ADF REST resource that exposes the LOV-enabled attribute, and then select the defining resource from the resource structure tree.

   The **Resource Name** list is populated by the ADF REST resource definitions in the Model project. The REST resource you select must have the same view object definition as the view object that defines the LOV attributes data source. For example, when creating an `Employees` resource, the `EmployeesView` view object defines an LOV attribute `JobsId`. In the Create LOV Resource editor, you would select the ADF REST resource `Jobs` and the root resource `JobsView1` from the `Jobs` resources resource structure tree, as shown in Figure 16-17.

Figure 16-17   Creating a Static LOV Resource for REST Create Operations


oracle.jbo.dt.ui.rest.RSTLOVResourcePanel
Select a LOV Resource that will provide the static list of values in REST
Release Version:
v2
Resource Name:
Jobs
Resource Tree:
JobsView1
Help
OK
Cancel

6. Click **OK** to save the LOV resource definition.

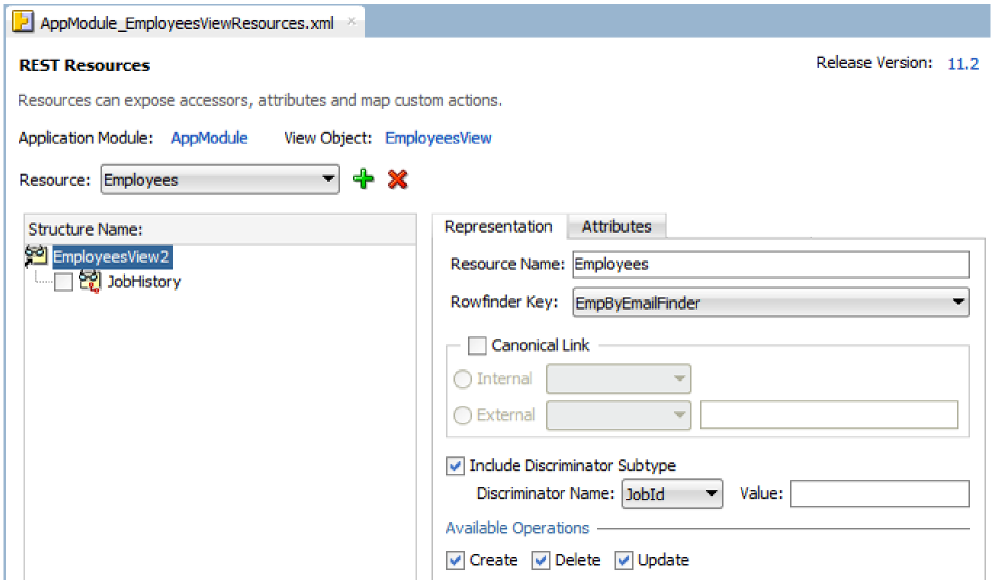
How to Enable Row Finder Keys in the ADF REST Resource

When you do not want service clients to be able to query resources using the primary key for the resource (often a numeric value), the business components developer can create a row finder that specifies a row finder key based on a view object attribute with values that can be uniquely identified by name. One such example may find employees by the first letters of the email attribute.

When a row finder has been explicitly defined on the view object named in the ADF REST resource, the overview editor for the ADF REST resource gives you the option to enable the row finder to filter the resource collection. Row finders are not enabled by default; you must the select the row finder key by name to apply it the current node selection in the resource structure.

In JDeveloper, the overview editor for ADF REST resources displays the list of available row finders in the **Rowfinder Key** dropdown. The list is populated by the row finders that have been defined on the view object definition backing the node selection in the resource structure tree. For example, Figure 16-18 shows the `EmpByEmailFinder` row finder key where `Employees` is the node defining the view instance with the row finder `EmpByEmailFinder`.

Figure 16-18 Row Finder Key Enabled on the Root of the ADF REST Resource



AppModule_EmployeesViewResources.xml

REST Resources

Release Version: 11.2

Resources can expose accessors, attributes and map custom actions.

Application Module: AppModule View Object: EmployeesView

Resource: Employees

Structure Name:
- EmployeesView2
  - □ JobHistory

Representation | Attributes

Resource Name: Employees

Rowfinder Key: EmpByEmailFinder

□ Canonical Link
- ○ Internal
- ○ External

☑ Include Discriminator Subtype

Discriminator Name: JobId Value:

Available Operations

☑ Create ☑ Delete ☑ Update

Before you begin:

It may be helpful to have an understanding of how the ADF REST framework supports application modules and enables service clients to access rows of data and perform service operations. For more information, see About RESTful Web Services and ADF Business Components.

You may also find it helpful to understand other Oracle ADF features that depend on ADF REST resources. For more information, see Additional Functionality for RESTful Web Services.

You will need to complete these tasks:

- Create the desired REST resource, as described in Creating ADF REST Resources Using the Application Module.
- Create the application module data model that contains the view object instances that you want the REST resource to expose, as described in How to Create an Application Module.
- Create the row finders on the view instances that you want the REST resource to expose, as described in Working with Row Finders.

To expose row finder filters:

1. In the Applications window, double-click the resource definition file that defines the ADF REST resource you want to edit.
2. In the overview editor for the REST resource, choose the desired resource from the **Resource** dropdown, expand the **Structure** tree and select the resource node for which you want to expose a row finder.

   If the data model does not define a master-detail hierarchy for the backing view instance of the root resource, then no view link accessor node will appear in the **Structure** tree. For more information about **Structure** nodes, see Table 16-1.

3. Click the **Representation** tab of the overview editor, and then, in the **Rowfinder key**, select the row finder on the current **Structure** tree selection.

   The **Rowfinder key** dropdown may appear disabled if the resource enables a canonical link and the canonical resource itself enables a row finder key. The overview editor for ADF REST resources will disable the row finder key field and automatically reflect the row finder key on the partial resource, as described in What You May Need to Know About Row Finders in ADF REST Resource Requests.

4. Optionally, click **Canonical Link** when you want the row finder selection to filter the canonical resource representing the current node selection.

   If you enable a canonical link, you must also select an internal or external source for the canonical resource.

5. Select another node in the **Structure** tree to expose another row finder.

What You May Need to Know About Configuring LOV Resources for Row Finders

When you want service clients to be able to query resources using a LOV resource, the business components developer can create a finder that specifies a finder key based on a view object attribute with a finder value that can be uniquely identified by name. A link between a regular resource and an LOV resource is configured using the resource description. You must configure a resource that has the LOV definition in order to use the LOV resource. Nested resources are exposed in an order to represent some cascading LOVs that follow a tree structure.

For example, consider there are LOVs for `State` and `City` named `LOV_City` and `LOV_State`. `LOV_City` filters the value that is already selected in `State`. Let us assume there is a resource named `Cities` that has all possible cities in the countries grouped by `State`. The `LOV_City` must filter the Cities resource with a State value already selected. For that you need to define a `Cities` view object with a finder with the value `ByStateFinder` that accepts a parameter `State`. The LOV URL would be `/Cities?finder=ByStateFinder;stateVar={State}`, where State points to the `State` attribute in the address object that has the master LOV.

The following view object definition shows the metadata for the view object that provides the LOV source for the cities and states example.

```
<ViewAttribute
  Name="City"
  IsNotNull="true"
  PrecisionRule="true"
  EntityAttrName="CityName"
  EntityUsage="Address"
  AliasName="CITYNAME"
  LOVName="LOV_City">
  <Properties>
    <SchemaBasedProperties>
      <LOVResource name="Cities" finder="ByStateFinder"/>
    </SchemaBasedProperties>
  </Properties>
</ViewAttribute>
<ViewAttribute
```

```
  Name="State"
  IsNotNull="true"
  PrecisionRule="true"
  EntityAttrName="StateCode"
  EntityUsage="Address"
  AliasName="STATECODE"
  LOVName="LOV_State">
  <Properties>
    <SchemaBasedProperties>
      <LOVResource name="States"/>
    </SchemaBasedProperties>
  </Properties>
</ViewAttribute>
```

In the above example, the cascading LOV is being handled by the LOV with a finder class.

Note that the custom property `LOVResource` identifies the source of that the LOV resource uses for its data. The property name `LOVResource` is specific to ADF REST framework version 5 and later and serves to distinguish the parametrized row finder URL usage to obtain the LOV list from the usage associated with earlier framework versions, where the LOV list URL does not use a row finder. The view object metadata for the LOV source attribute identifies framework versions 1 though 4 by using the property name `ResourceLOV`.

The resource describe associated for these two usages also differs. Compare the static LOV resource associated with the regular view object-backed resource with framework version 5 enabled (attribute metadata property identified as `LOVResource`) to the same resource describe with framework version 4 enabled (attribute metadata property identified as `ResourceLOV`).

Here is the describe with version 5 enabled. The `LOVResource` is added to the `lov` element section of the regular resource (`Address`) and the `childRefForCreate` property identifies the URL name that appears in the `item - links` element section .

```
{
  "Resources" : {
    "Address" : {
      "discrColumnType" : false,
      "attributes" : [ {
       ...
        }, {
          "name" : "City",
          "type" : "string",
          "updatable" : true,
          "mandatory" : true,
          "queryable" : true,
          "precision" : 32,
          "controlType" : "choice",
          "maxLength" : "32",
          "lov" : {
            "childRef" : "CityLOV",
            "childRefForCreate" : "CityLOVForCreate",
            "attributeMap": [ {
              "source" : "CityName",
       "target" : "City"
      }],
      "displayAttributes" : [ "CityName" ],
     }
    }, {
     "name" : "State",
     "type" : "string",
     "updatable" : true,
     "mandatory" : true,
     "queryable" : true,
     "precision" : 2,
     "controlType" : "choice",
     "maxLength" : "2",
     "lov" : {
      "childRef" : "StateLOV",
      "childRefForCreate" : "StateLOVForCreate",
      "attributeMap": [ {
       "source" : "StateCode",
       "target" : "State"
      }],
      "displayAttributes" : [ "StateCode" ],
     }
    }, {
   ...
   },
   "item" : {
    "links" : [ {
     ...
    }, {
     "rel": "lov",
     "href": "http://server/demo/rest/11.0/Address/{id}/lov/
CityLOV",
     "name": "CityLOV",
     "kind": "collection"},
    }, {
     "rel" : "lov",
     "href" : "http://server/demo/rest/11.0/Cities?
finder=ByStateFinder%3BstateVar%3D{State}",
     "name" : "CityLOVForCreate",
     "kind" : "collection"
    }, {
    }, {
     "rel": "lov",
     "href": "http://server/demo/rest/11.0/Address/{id}/lov/
StateLOV",
     "name": "StateLOV",
     "kind": "collection"},
    }, {
     "rel" : "lov",
     "href" : "http://server/demo/rest/11.0/States",
     "name" : "StateLOVForCreate",
     "kind" : "collection"
    }, {
...
```

Here is the describe with version 4 enabled. The ResourceLOV is added to the lov element section of the regular resource (Address) and the lovResourcePath array identifies the URL name that appears in the links element section

```
{
  "Resources" : {
    "Address" : {
      "discrColumnType" : false,
      "attributes" : [ {
       ...
        }, {
          "name" : "City",
          "type" : "string",
          "updatable" : true,
          "mandatory" : true,
          "queryable" : true,
          "precision" : 10,
          "controlType" : "choice",
          "maxLength" : "10",
          "lov" : {
            "childRef" : "CitiesLOV",
            "attributeMap" : [ {
              "source" : "CityName",
              "target" : "City"
            } ],
            "displayAttributes" : [ "CityName" ],
            "lovResourcePath" : [ {
                  "resource" : "Cities",
                  "filter" : "?
finder=FilterByState%3BVStateName%3D{State}"
            } ]
          }
        }, {
      ...
      },
      "links" : [ {
        ...
          "rel" : "lov",
          "href" : "http://server/demo/rest/11.0/Cities",
          "name" : "Cities",
          "kind" : "collection"
        }, {
...
```

What You May Need to Know About Row Finders in ADF REST Resource Requests

At runtime, an ADF REST resource with a row finder key enabled must be accessed in the service client by the row finder key attribute. The row finder supports returning a collection or instance for the row finder-enabled resource using only the row finder key attribute. Thus, service clients may access REST resources without requiring knowledge of database primary key values to end users. For example, compare these two GET requests to retrieve a specific employee in the Employees resource collection:

No row finder:

```
http://localhost:7101/RESTSample_App/rest/11.1/Employees/101
```

With row finder

```
http://localhost:7101/RESTSample_App/rest/11.1/Employees/NSMITH
```

The first URL, with no row finder key defined, requires the ID of the employee to locate the employee resource item. The second URL uses the row finder key value (employee's email address) in the resource path. Once the row finder key has been enabled on the resource at design time, service clients may access the resource by row finder key values using links generated in the request payload by the ADF REST runtime.

You can optionally apply the row finder to a canonical resource that your ADF REST resource defines. If a canonical resource with row finder key is defined on the ADF REST resource with a row finder key enabled, then both sides must be defined by the same key structure. The key attributes must be of the same type and in the same sequence. When you create a canonical resource with a row finder key and then create a partial resource with a canonical link, the overview editor for ADF REST resources will disable the row finder key field and automatically reflect the row finder key on the partial resource. For details about canonical resources, see How to Expose Canonical Resources in the ADF REST Resource.

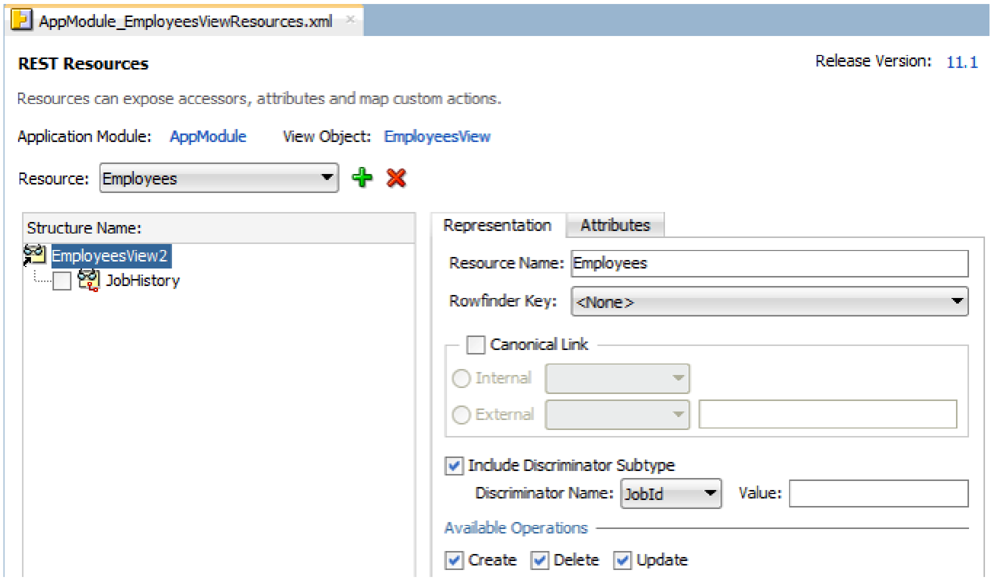
How to Expose Subtype View Objects in the ADF REST Resource

A subtype view object is a view object that extends a base view object in an inheritance hierarchy. The subtype view object inherits its list of attributes from the base view object and contributes its own attributes that are specific to the subtype. Such subtype view objects are called polymorphic view objects. At runtime, polymorphic view objects define subtype view rows comprised of attributes from the base view object and attributes from the defining subtype view object.

In order to query subtype view rows, the subtype view object must specify a discriminator attribute that determines the specific type. For example, an `EmployeesView` base view object with a discriminator attribute `JobId` may instantiate subtype view rows based on the available jobs (Accountant, Sale Representative, and so on), where the attributes of subtype view rows are determined by the specified job ID value. For example, a Sales Representative with the job ID `SA_REP` might contribute the subtype view row attribute `CommissionPct`, whereas the subtype view rows for the other job types would omit this attribute.

The overview editor for REST resources lets you expose subtype view objects for each node of the Structure tree when the view object backing the node defines a discriminator attribute. For example, Figure 16-7 shows the `Employees` resource and the `EmployeesView2` resource node selected in the overview editor. The backing view object `EmployeesView` defines the discriminator attribute `JobId` and the subtype view object `SalesEmployeeView` defines the subtype `SA_REP`. The editor exposes the `SalesEmployeesView` subtype view object by enabling **Include Discriminator Subtype**, specifying the **Discriminator Name** and omitting the discriminator attribute value. Omitting the `JobId` discriminator attribute value ensures that subtype attributes will be returned in service client requests for `Employees` based on the `JobId` value of the requested employee.

Figure 16-19 Defining a Discriminator Attribute in the REST Resource Definition



AppModule_EmployeesViewResources.xml

REST Resources

Release Version: 11.1

Resources can expose accessors, attributes and map custom actions.

Application Module: AppModule View Object: EmployeesView

Resource: Employees

Structure Name:
- ☑ EmployeesView2
  - □ JobHistory

Representation | Attributes

Resource Name: Employees

Rowfinder Key: <None>

□ Canonical Link

○ Internal

○ External

☑ Include Discriminator Subtype

Discriminator Name: JobId Value:

Available Operations

☑ Create ☑ Delete ☑ Update

> **Note:**
>
> Discriminator attributes are defined on the backing view object and support subtype view rows. Subtype view objects are not exposed in the design time as ADF REST resources. The subtype definition is specified on the application module and by the ADF REST runtime.

Before you begin:

It may be helpful to have an understanding of how the ADF REST framework supports application modules and enables service clients to access rows of data and perform service operations. For more information, see About RESTful Web Services and ADF Business Components.

You may also find it helpful to understand how view objects support subtype view rows. For more information, see Working with Polymorphic View Rows.

You may also find it helpful to understand other Oracle ADF features that depend on ADF REST resources. For more information, see Additional Functionality for RESTful Web Services.

You will need to complete these tasks:

- Create the polymorphic view objects, as described in How to Create a View Object with Polymorphic View Rows.
- Create the REST resource for which you want to add or remove a resource, as described in Creating ADF REST Resources Using the Application Module.

To expose subtype view rows in the resource definition:

1. In the Applications window, double-click the resource definition file that defines the ADF REST resource you want to edit.
2. In the overview editor for the REST resource, choose the desired resource from the **Resource** dropdown, expand the **Structure** tree and select the resource node for which you want to expose a subtype view object.

   If the data model does not define a master-detail hierarchy for the backing view instance of the root resource, then no view link accessor node will appear in the **Structure** tree. For more information about **Structure** nodes, see Table 16-1.
3. Click the **Representation** tab of the overview editor, and then, select **Include Discriminator Subtype** to enable subtype view rows on the current **Structure** tree selection.
4. Select the subtype discriminator attribute from the **Discriminator Name** dropdown and omit the value that defines the desired subtype view rows when the subtype view object defines the subtype discriminator attribute value.

   The **Discriminator Name** dropdown is populated by the discriminator attribute definition of the backing view object. For example, an `EmployeesView` view object might define the discriminator attribute `JobId` attribute as the discriminator attribute with a subtype view object `SaleEmployeesView`. In most cases, the subtype view object will define the subtype discriminator attribute value, and you can leave the attribute value empty in the resource definition. For example, where the subtype view object `SalesEmployeesView` defines `SA_REP` (Sales Representative) as the `JobId` discriminator attribute value, a request made for the `Employees` resource will contain the subtype attributes based on the employee's job ID and not by the subtype defined by the resource. Defining the discriminator attribute subtype value in the resource forces the resource items to the same type.

How to Expose Standard HTTP Actions in the ADF REST Resource

Standard data manipulation operations that you expose on the ADF REST resource correspond to actions on the backing view instance of the resource. At runtime, the service client invokes these operations as HTTP actions on the RESTful web service.

Table 16-2 lists the standard operations that are exposed by default in the overview editor for the REST resource. You can expose a different set of operations by deselecting operations from the list for each resource selection you make in the resource tree. Note that the GET method is always exposed for a resource.

Table 16-2 Standard View Instance Data Manipulation Operations

Operation Selection	HTTP Method Name	Operation Description
Create	POST	Creates an ADF Business Components view row or view row set.
Create and **Update** (combined equals Upsert)	POST	Merges payload content with the REST resource using a POST method in combination with a header `Upsert-Mode` set to true. Creates the REST resource if the REST resource does not exist. Merges the payload content with the REST resource if this REST resource exists.

Table 16-2   (Cont.) Standard View Instance Data Manipulation Operations

Operation Selection	HTTP Method Name	Operation Description
Update	PATCH	Updates the specified items of an ADF Business Components view row. The view row must exist for this operation to succeed.
Delete	DELETE	Deletes an ADF Business Components view row or view row set.
not applicable	GET	Reads an ADF Business Components view row or view row set. Note that this operation is available by default when you create a resource.

> **Note:**
>
> Note that Oracle has deprecated the functionality for executing HTTP PUT methods on ADF REST resource requests. In the current release, the describe for ADF REST resources continues to display PUT actions when the backing view object has the Update operation enabled (the operation enables both PUT and PATCH methods); however, ADF REST service clients should avoid making PUT requests (replace all items of the view row) as this functionality will be desupported in a future release.

Before you begin:

It may be helpful to have an understanding of how the ADF REST framework supports application modules and enables service clients to access rows of data and perform service operations. For more information, see About RESTful Web Services and ADF Business Components.

You may also find it helpful to understand other Oracle ADF features that depend on ADF REST resources. For more information, see Additional Functionality for RESTful Web Services.

You will need to complete this task:

- Create the desired REST resource, as described in Creating ADF REST Resources Using the Application Module.

To expose HTTP actions:

1. In the Applications window, double-click the resource definition file that defines the ADF REST resource you want to edit.
2. In overview editor for the REST resource, choose the desired resource from the **Resource** dropdown, expand the **Structure** tree and select the resource accessor node that you want to edit.

   You can expose operations for root resources and child resources. For more information about **Structure** nodes, see Table 16-1.

3. Click the **Representation** tab of the overview editor, and then, in the **Available Operations** section, select the operations to expose for the current **Structure** tree selection.
4. Select another node in the **Structure** tree to expose operations on another resource accessor in the resource structure.

What You May Need to Know About Optimizing ADF REST Resource Get Operations

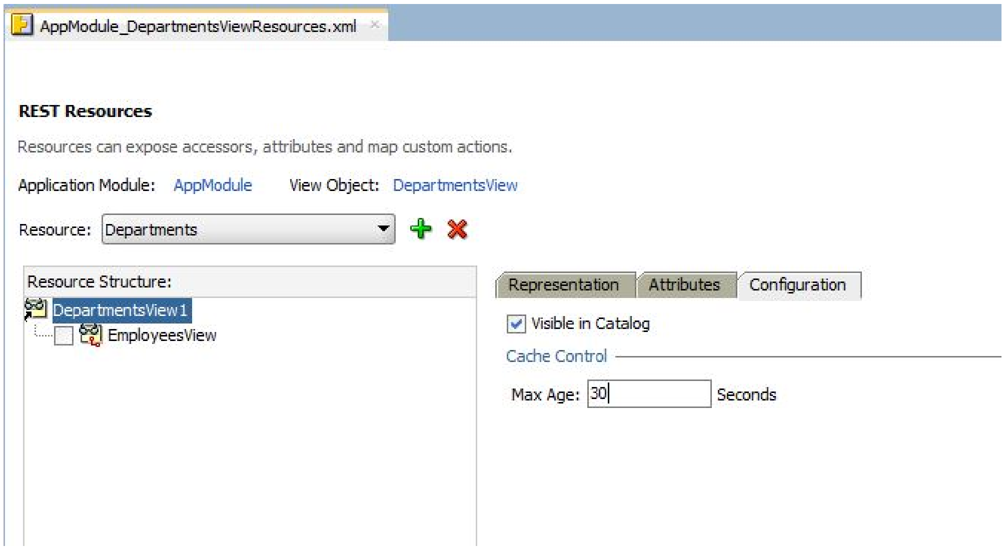
All ADF REST resources support read operations by service clients. To support efficient read operations (HTTP GET requests), the backing view object definition of ADF REST resources should define access mode set to support paging of the collection in an HTTP Request payload. For more information about range paging, see How to Enable Range Paging for a View Object.

How to Control Caching of ADF REST Resources

When you want fine control over the cache duration of ADF REST resource payload resulting from an ADF REST resource request, you can configure cache control in seconds on the root of the resource in the REST resource overview editor. At runtime, the cache setting can help to avoid intermediate proxies to cache/store the resource payload in the HTTP response.

In JDeveloper, the overview editor for REST resources displays the resource cache control setting in the **Configurations** tab for the root node of the resource. When this setting is made on the root resource, it overrides the application-level cache setting that may be configured in the `web.xml` file. For example, the following figure shows the cache maximum age set to 30 seconds on the root of the Departments resource. The option **Visible in Catalog** if selected enables the cache setting to appear in the describe for ADF REST resource requests.

Figure 16-20 Cache Maximum in Seconds Set on the Root of the REST Resource



AppModule_DepartmentsViewResources.xml

REST Resources

Resources can expose accessors, attributes and map custom actions.

Application Module: AppModule   View Object: DepartmentsView

Resource: Departments

Resource Structure:
- ☑ DepartmentsView1
  - □ EmployeesView

☑ Visible in Catalog

Cache Control

Max Age: 30 Seconds

From release 12.1.1.4.0, REST framework supports additional configuration options for `Cache-Control`. See Header Field Definitions - Cache Control. You can specify these additional configuration options in the source editor. When you specify one of the valid values, then that `Cache-Control` header will be added in the payload response.

The valid values are:

- **Cacheability**: indicates if a response can be cached or not. Valid values are: `public`, `private`, and `no-cache`.
  The value `public` cache indicates that the response can be cached by any cache irrespective of the cache status. This value is useful for requests that include an `authorization` header, because requests with authorization header prevents a shared cache from caching the response.

  The value `no-cache` does not mean not to cache the response, but instructs the browser to re-validate before returning the response.
- **NoStore**: indicates that the response should not be cached irrespective of the cacheability setting. It is a boolean flag with values `true` and `false`. When set to `true`, `no-store` is added to the `Cache-Control` header in the response, separated by comma if other settings are present.
- **Revalidation**: indicates that the response must be revalidated if a cache is configured to return a stale response. If you configure `MaxAge` to `0`, the response becomes stale immediately, thus forcing the cache to revalidate every time. Valid value is `must-revalidate`.

See, What You May Need to Know About the Cache-Control Header.

Before you begin:

It may be helpful to have an understanding of how the ADF REST framework supports application modules and enables service clients to access rows of data and perform service operations. For more information, see About RESTful Web Services and ADF Business Components.

You may also find it helpful to understand other Oracle ADF features that depend on ADF REST resources. For more information, see Additional Functionality for RESTful Web Services.

You will need to complete this task:

- Create the REST resource for the view object, as described in Creating ADF REST Resources Using the Application Module.

To configure the cache control maximum age setting:

1. In the Applications window, double-click the resource definition file that defines the ADF REST resource you want to edit.
2. In the overview editor for the REST resource, choose the desired resource from the **Resource** dropdown and then, in the **Structure** tree, select the root resource node.

   For more information about **Structure** nodes, see Table 16-1.
3. Click the **Configuration** tab of the overview editor, and then, click **Visible in Catalog** when you want the describe to show the cache control setting. For **Max Age** enter the maximum age of the payload cache in seconds.

This tab is not displayed for child resource nodes that you select in the **Structure** tree.

What You May Need to Know About the Cache-Control Header

The following table lists few example use cases that REST framework supports with selected subset of options in any valid combination. The developer can choose any combination to generate the desired `Cache-Control` header in the response.

Cache-Control Value	Category	Explanation	Example

`<CacheControl>`
`<Cacheability>no-cache</Cacheability>`
`<NoStore>true</NoStore>`
`<Revalidation>must-revalidate</Revalidation>`
`</CacheControl>`
	Sensitive data	Response is not cached.	Bank account info

`<CacheControl>`
`<Cacheability>public</Cacheability>`
`<MaxAge>30</MaxAge>`
`</CacheControl>`
	Public shared data	Response can be cached by browser and any intermediary caches for up to the specified number of seconds.	LOV. For example country codes

`<CacheControl>`
`<Cacheability>private</Cacheability>`
`<MaxAge>30</MaxAge>`
`</CacheControl>`
	Private data	Response can be cached by the client's browser only for up to the specified number of seconds.	Order history

For example, the following service configuration is defined on `DeptAM_DeptVOResources.xml`:

```
<tree IterBinding="DeptIter" id="departments" AccessorFolder="Always">
  <ServiceConfiguration>
    <CacheControl>
      <Cacheability>private</Cacheability>
      <MaxAge>600</MaxAge>
    </CacheControl>
  </ServiceConfiguration>
  <nodeDefinition .../>
</tree>
```

Any requests against the departments resource have the following `Cache-Control` header:

REQUEST

```
GET http://example.com/demoApi/resources/1.0/departments
```

RESPONSE

```
200 OK
Cache-Control: private,
max-age=600
{
  ...
```

How to Control the Format of the ADF REST Response

You can use REST-Pretty-Print header to remove extra whitespace in the response payload. You can set this attribute for describe and GET, PATCH, and POST payload responses. You must set this property in the adf-config.xml under the startup Properties element.

```
<adf-adfm-config xmlns="http://xmlns.oracle.com/adfm/config">
    <startup>
      <Properties>
        <adf-property name="ORACLE.BC.REST.PRETTYPRINTJSON" value="true"/>
      </Properties>
    </startup>
</adf-adfm-config>
```

In the case of describe, this header reduces the size of the response by nearly 50%.

Versioning the ADF REST Resource

JDeveloper allows you to version the ADF REST resource. It helps create new versions of the ADF REST resource if use cases or backing view objects of ADF REST resource change.

Changes that you make to ADF REST resources during the development cycle can be made without the need to version the resource. However, once you have deployed the RESTful web service, service clients that access the ADF REST resources rely on the stability of the resource collection to satisfy their use cases.

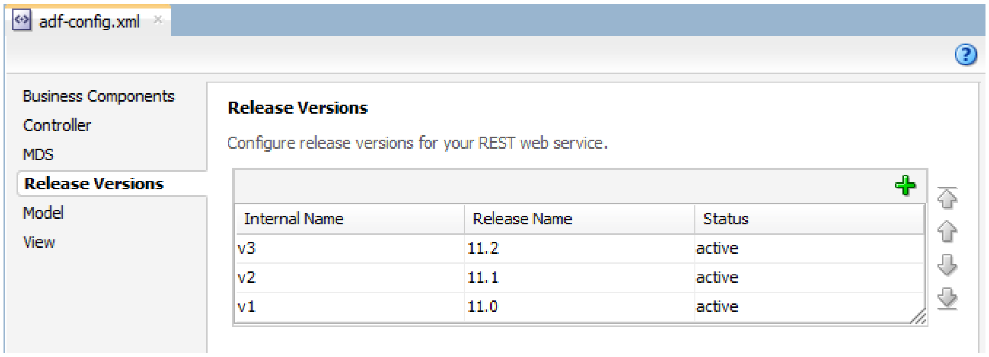
Over time, as use cases change or the backing view objects of the REST resources change, you may need to modify the resource structure of already deployed resources. At that time, modifications to existing ADF REST resources may become backward-incompatible with the deployed REST service and may necessitate creating a new version identifier and ADF REST resource for that version.

How to Version the ADF REST Resource

You use the overview editor for the adf-config.xml file to create new version identifiers and to update the lifecycle status of existing versions. Lifecycle status can be either active, deprecated, or desupported. If the service client requests a resource that does not exist for the requested version, the ADF REST runtime will fallback to a previous version with status active or deprecated

Note that the order of the release name identifiers in the release version list is significant. The ADF REST runtime recognizes the top-most release name as the most recent version. Subsequent release names below the top name in the list form a sequential list of versions. As Figure 16-21 shows, `11.0` is the first version and must appear at the bottom of the list, with subsequent versions, listed above, where `11.2` is the latest version.

Figure 16-21   Managing ADF REST Resource Versions



Release Versions

Configure release versions for your REST web service.

Internal Name	Release Name	Status
v3	11.2	active
v2	11.1	active
v1	11.0	active

> **Note:**
>
> To alter the position of a release identifier in the sequential list of versions, you can use the arrows in the Release Versions page of the overview editor to move the identifier up (more recent version) or down (previous version) in the list.

Before you begin:

It may be helpful to have an understanding of how the ADF REST framework supports application modules and enables service clients to access rows of data and perform service operations. For more information, see About RESTful Web Services and ADF Business Components.

You may also find it helpful to understand when editing a resource may introduce backward-incompatible changes that require creating a resource with a new version. For more information, see What You May Need to Know About Versioning ADF REST Resources.

You may also find it helpful to understand how requests by service clients depend on the version identifier of the resource. For more information, see What Happens At Runtime: Invoking ADF REST Resource Versions.

You may also find it helpful to understand other Oracle ADF features that depend on ADF REST resources. For more information, see Additional Functionality for RESTful Web Services.

To manage ADF REST versions:

1. In the Applications window, in the Application Resources pane, expand the **Descriptors** and **ADF META-INF** nodes, and then double-click the **adf-config.xml** file.
2. In the overview editor for the `adf-config.xml` file, click the **Release Versions** navigation tab, then click the **Create New Release Version** icon button to create a new resource version identifier.

3. When you want to change the order of the versions to affect the runtime, select a version from the table and use the arrows to the right of the table to change its hierarchy in the list of versions.

   The ADF REST runtime will recognized the version at the top of the list as the most recent version of the resource.

4. Optionally, select the status to conform to the desired resource lifecycle: **active**, **desupported**, **deprecated**.

   After you create the version and assign it to REST resources, the version identifier must not be deleted from the `adf-config.xml` file. The proper way to effectively remove a version is to change its status to **desupported**.

What You May Need to Know About Versioning ADF REST Resources

The following are examples of changes to the ADF REST resource structure that should be considered backward-incompatible with already deployed RESTful web services:

- Renaming exposed attributes
- Removing or hiding previously exposed attributes
- Removing previously exposed child resources
- Disabling standard operations
- Disabling LOV view accessors or changing an LOV to a different view object

Under certain circumstances, you may modify the ADF REST structure without versioning the resource. The following changes are considered to be backward compatible and therefore do not necessitate creating a new version of the already deployed ADF REST resource.

- Exposing new attributes and the backing view object does not define the attributes as mandatory

  Note that exposing mandatory-defined attributes requires versioning the resource.
- Exposing standard operations
- Exposing new child resources
- Defining a LOV view accessor for already exposed attributes

What Happens At Runtime: Invoking ADF REST Resource Versions

Version identifiers that you specify will form the URL that the service client uses to invoke operations of the REST resource. For example, the following GET request specifies the URL parameters `11.0` and `11.1` to access two different versions of the `Departments` resource:

```
http://localhost:7101/RESTSample_App/rest/11.0/Departments
```

```
http://localhost:7101/RESTSample_App/rest/11.1/Departments
```

When you use the overview editor for the `adf-config.xml` file to update the lifecycle status of existing versions, you tag each version as either `active`, `deprecated`, or `desupported`. The status `deprecated` has no runtime impact and is provided for documentation purposes, but the status `desupported` will cause the deployed resource to become inaccessible.

Note that the order of the release name identifiers in the adf-config.xml file's release version list is significant. The ADF REST runtime recognizes the top-most release name as the most recent version. Release names below the top name in the list form a sequential list of versions. For example, the following GET request retrieves version 11.1 of the Departments resource, where 11.1 is at the top of the release versions list:

```
http://localhost:7101/RESTSample_App/rest/11.1/Departments
```

If the service client requests a resource that does not exist for the requested version, the ADF REST runtime will fallback to a previous version with status active or deprecated. Thus, if the Departments resource versions 11.0 and 11.1 exist but 11.2 does not, the request for version 11.2 will return the next more recent resource, version 11.1.

Versioning the ADF REST Runtime Framework

JDeveloper allows you to specify the default ADF REST runtime framework that you want the service client to use when processing requests on ADF REST resources. Declaring a default framework version ensures that clients interact with the appropriate level of functionality.

Oracle JDeveloper 12c releases, starting with release 12.2.1.2.0, will introduce ADF REST framework enhancements to support new runtime functionality on service clients. Whether or not your application will benefit from these enhancements depends on the use cases of your application. To support your ability to evaluate framework enhancements before exposing them to service clients, each iteration of the ADF REST framework is identified by a version number, such as version 1, 2, and 3, where version 1 is associated with your application by default. Your application can either remain at the current framework version (for example, the base framework version 1) or you can decide to opt into a newer version (for example, framework version 2) to enhance your application use cases. This ensures that the ADF REST runtime will process the request using the framework version that offers the desired level of functionality for your service clients. Being able to specify a framework version to process requests on your application, allows you to opt into those features when you are ready.

As new ADF REST runtime framework versions are introduced, you may want to update your application to make use of runtime functionality offered in a later framework. To accomplish this, the ADF REST design time allows you to associate a restFrameworkVersion property in the adf-config.xml file to define the default framework version for each version of your application's ADF REST resources. For example, you may have a resource version 11.0, which is by default associated with the base functionality of framework version 1. Then starting with Oracle JDeveloper 12.2.1.2.0 and ADF REST framework version 2, you can choose to create a new resource version 11.1 and associate it with framework version 2. Alternatively, you can opt into framework version 2 on your existing resource version instead of creating a new resource version. By specifying a default framework version that you associate with individual resource versions, you ensure that your application will expose the appropriate level of functionality to service clients.

> **Note:**
>
> Starting in JDeveloper release 12.2.1.2.0, versions 1, 2, and 3 of the ADF REST runtime framework exist, where version 1 is the base version and includes the ability to support multiple framework versions on the ADF REST service client. Starting in JDeveloper release 12.2.1.4.0, versions 4, 5, 6, and 7 also exist. For complete details about versions of the framework, see What You May Need to Know About Versioning the ADF REST Runtime Framework.

How to Version the ADF REST Runtime Framework

You use the overview editor for the `adf-config.xml` file in Source view to declare the default ADF REST framework versions for the resource versions of the ADF REST application. When the service client requests a resource, the ADF REST runtime will process the request using the declared framework version that is associated with the resource version specified on the request URL.

Before you begin:

It may be helpful to have an understanding of how the ADF REST framework supports application modules and enables service clients to access rows of data and perform service operations. See About RESTful Web Services and ADF Business Components.

You may find it helpful to understand the role of release versions that you create to manage the release iterations of your application's ADF REST resources. For more information, see Versioning the ADF REST Resource.

You may also find it helpful to understand how an ADF REST framework version may introduce backward-incompatible changes that will require creating a new resource version and declaring a default framework version for it. See What You May Need to Know About Versioning ADF the REST Framework.

You may also find it helpful to understand how the processing of requests by service clients is affected by the ADF REST framework declared for the resource version. See What Happens At Runtime: Invoking an ADF Rest Framework Version.

You may also find it helpful to understand other Oracle ADF features that depend on ADF REST resources. See Additional Functionality for RESTful Web Services.

To declare default ADF REST framework versions:

1. In the Applications window, in the Application Resources pane, expand the **Descriptors** and **ADF META-INF** nodes, and then double-click the **adf-config.xml** file.
2. In the overview editor for the `adf-config.xml` file, click the **Release Versions** navigation tab to view the existing list of release versions that the application uses.
3. Optionally, click the **Create New Release Version** icon button to create a new resource version identifier to associate with the new framework version. For example, you may want to create a new release version when the new ADF REST framework version will introduce backward-incompatible changes to the service client and you wish to preserve the original functionality for the existing release version that you can associate with the existing framework version.

4. To specify a default framework version to process requests for the resources of a particular release version, click the **Source** tab and edit the `<version>` element to add the `restFrameworkVersion` property with a specific framework version number. Starting in JDeveloper 12.2.1.2.0, versions 1, 2, and 3 are available. Starting in JDeveloper 12.2.1.4.0, versions 4, 5, 6, and 7 are also available. Make no other changes to the XML and specifically do not change the order of the release versions, where the top-most release version defines the most recent release version.

```
<versions>
    <version name="11.2" displayName="11.2" restFrameworkVersion="3"/>
    <version name="11.1" displayName="11.1" restFrameworkVersion="2"/>
    <version name="11.0" displayName="11.0" restFrameworkVersion="1"/>
</versions>
```

   In the above example, requests for ADF REST resources named version `11.0` will use framework version `1` to process the request, while requests for resources named `11.1`, will use the functionality offered by framework version `2`, and requests for resources named `11.2`, will use the functionality offered by framework version `3` when processing the request.

What You May Need to Know About Versioning the ADF REST Framework

One reason that you may want to version the ADF REST framework for your ADF REST application is to opt into new functionality offered by a later version of the ADF REST runtime framework. Currently, Oracle offers the following framework versions.

> **Note:**
>
> Each ADF REST framework version after version 1 introduces functionality that the previous framework versions does not support. Thus, when you choose to opt into a later framework version, the REST API of your application may introduce backward incompatible changes on the service client consuming the REST API. This topic explains the changes for each framework version.

Framework Version 1

The ADF REST framework identified as version 1 is available starting with JDeveloper release 12.2.1.0.0, when ADF REST was initially introduced. This is the base version which specified a particular JSON payload format. In this release and all subsequent releases, the ability to support multiple framework version was included, by allowing the `restFrameworkVersion` property in the `adf-config.xml` file.

Version 1 is the default version that the ADF REST runtime will use to process requests for service clients when no other version is specified. Therefore, when you want to support service clients at the base level of functionality, your application does not need to declare the version using the `restFrameworkVersion` property in the `adf-config.xml` file.

Note that the query-by-example resource query syntax supported in the base framework version (version 1) is not compatible with later versions of the ADF REST framework. Beginning with version 2 of the ADF REST framework, a more advanced query syntax is offered instead.

Framework Version 2

The ADF REST runtime framework identified as version 2 is available starting with JDeveloper release 12.2.1.2.0. The purpose of this new version is to introduce an expanded query expression syntax for ADF REST requests. Version 2 of the ADF REST framework will interpret the `q` query parameter value differently than the way framework version 1 does, and therefore introduces a backward incompatible change to service clients that rely on framework version 1. Only when framework version 2 (or later) is specified for the resource request will the ADF REST runtime support the use of the expanded expression syntax to process the request.

In version 1, filtering resource collections using the `q` query parameter is limited to a query-by-example syntax, as follows.

```
GET /rest/11.0/Departments?q=Dname SA*;Loc BOSTON
```

Whereas, starting in version 2, the new advanced query syntax supports filtering resource collections using rowmatch query expressions, as follows.

```
GET /rest/11.1/Departments?q=Dname like 'SA*' or Loc = 'BOSTON'
```

> **Note:**
>
> To support both rowmatch expressions (offered in framework version 2 and later) and the query-by-example syntax (version 1 only), you would need to associate framework version 2 (or later) with a new release version identifier that you define for the REST resources in the `adf-config.xml` file. For example, in the above URL samples, you can preserve the original functionality for release version 11.0 and expose the new functionality for version 11.1. This assumes that the `adf-config.xml` file for the application has declared framework version 2 the default using the `restFrameworkVersion=″2″` property on release version `11.1`. Alternatively, if your service clients will no longer require the functionality of your current framework version, you may associate the new framework version with your existing release version identifier. Therefore, you are not required to increment the release version to make use of a new framework version.

For an explanation of the enhanced query syntax offered by rowmatch expressions, see What You May Need to Know About the Advanced Query Syntax in ADF REST Framework Version 2.

Framework Version 3

The ADF REST runtime framework identified as version 3 is available starting with JDeveloper release 12.2.1.2.0. The purpose of this version is to add support for retrieving nested child resources with payload attributes that may be used by the service client to determine whether more resource items would be returned in a subsequent ADF REST request. To support this functionality, the payload structure in framework version 3 now represents nested child resources as a resource collection, instead of an array of items, as was true in version 1 and 2. Therefore, version 3 introduces a backward incompatible change to service clients that rely on framework version 1 or version 2. If you decide to opt into version 3, you will expose functionality that allows GET operations to use the ?expand and ?fields query parameter to return a nested child resource as a resource collection with the hasMore attribute. In affect, this change supports pagination of nested child resources that would otherwise require more than one request to fetch.

When you want to add support for framework version 3 to your application, the same guidelines described for framework version 2 (see above section) apply for preserving the existing level of functionality using the restFrameworkVersion property in the adf-config.xml file.

For an example of the new payload structure for nested child resources introduced in version 3, see Fetching Nested Child Resources and Querying With Filtering Attributes (Partial Get). For details about paginating a resource collection using the hasMore attribute, see Paging a Resource Collection.

Framework Version 4

The ADF REST runtime framework identified as version 4 is available starting with JDeveloper release 12.2.1.4.0. In addition to HTTP status codes and error messages, it is possible to obtain exception details in the response when your request is enabled to use ADF REST framework version 4 and the request is made for either application/vnd.oracle.adf.error+json or application/json media types. With framework version 4, the response will be in the form an exception detail payload which provides the following benefits to the service client:

- If multiple errors occur in a single request, the details of each error are presented in a hierarchical structure.
- An application-specific error code may be present that identifies the ADF exception corresponding to each error.
- An error path may be present that identifies the location of each error in the request payload structure.

> **Note:**
>
> The exception detail may or may not present certain details, such as the application-specific error code and the request payload’s error path.

For example, compare the error response for a POST submitted with a payload that contains the following incorrectly formatted date field when framework version 3 (or earlier) is enable and when framework version 4 (or later) is enabled.

```
{    "EmpNum" : 5027,
     "EmpName" : "John",
     "EmpHireDate" : "not a date"
}
```

Standard Error Response

Without framework version 4, no response payload is generated and instead only a single error message that does not reference the request payload will be returned in the response.

```
"An instance of type oracle.jbo.domain.Date cannot be created from
string not a date. The string value must be in format YYYY-MM-
DDTHH:MI:SS.sss+hh:mm."
```

Exception Payload Error Response

With framework version 4 (or later) enabled, the following exception detail payload is generated for the response. The payload includes the usual HTTP status code and formats the details of one or more exceptions in an array structure.

```
{    "title" : "Bad Request",
    "status" : "400",
    "o:errorDetails" : [ {
     "detail" : "An instance of type oracle.jbo.domain.Date cannot be
created from string not a date.
               The string value must be in format YYYY-MM-
DDTHH:MI:SS.sss+hh:mm.",
       "o:errorCode" : "26099",
       "o:errorPath" : "/EmpHireDate"
    } ]
 }
```

Framework Version 5

The ADF REST runtime framework identified as version 5 is available starting with JDeveloper release 12.2.1.4.0. In versions 4 and earlier you might pick a nested resource as an LOV target. For example: `/rest/v1/States/California/Cities`. In versions 5 and later you use filtering to access the nested LOV resource instead, for example: `/rest/v1/Cities?finder=ByState;name=California`. With this framework version, the row context LOV URLs are no longer returned in the payload and in the describe. Only LOV resource URLs that point to top-level resources are described and included in the payloads.

See What You May Need to Know About Configuring LOV Resources for Row Finders and Retrieving LOV-Enabled Attribute Values with Framework Version 5 and Later.

This framework version 5 also provides support for request and response parameters beyond the current supported types of string, number, date, and boolean. The supported Java types include `java.util.List` and `java.util.Map`. For the ADF REST catalog describe, if the response parameter is `java.util.List`, then specify the request parameter as `array` and if the response parameter is `java.util.Map` then specify the request parameter as `object`.

```
{
  "name" : "sales",
  "parameters" : [ {
    "name" : "prices",
    "type" : "array",
    "typeProperties": {
      "itemType": "number"
    },
    "mandatory" : false
  }, {
    "name" : "quantities",
    "type" : "object",
    "typeProperties": {
      "itemType": "array",
      "typeProperties": {
        "itemType": "integer"
      }
    },
    "mandatory" : false
  } ],
  "resultType" : "object",
  "typeProperties": {
    "itemType": "number"
  },
  "method" : "POST",
  "requestType" : [ "application/vnd.oracle.adf.action+json" ],
  "responseType" : [ "application/vnd.oracle.adf.actionresult+json",
"application/json" ]
}
```

Framework Version 6

The ADF REST runtime framework identified as version 6 is available starting with JDeveloper release 12.2.1.4.0. The purpose of this framework version is to easily differentiate between the resource fields and the item context information like links and headers. A new element `@context` is introduced in this version and all the information for an item is moved under `@context` section. The `changeIndicator` value is moved to `ETag`, which is under `headers`. A new context information key is included under `@context` that contains the unique identifier of the specific resource item as a string. The `@context` section also contains warnings in the response payload for create/upsert and update actions. See Fetching a Resource with Grouped Context Information .

The new payload for a resource item in a response payload and collection response payload will be similar to the one below:

```
{
   "field1": "value1",
   "field2": "value2",
     ...
   "@context" : {
      "key" : "AB8765BCD",
      "headers" : {
         "ETag" : "ACED..."
       ...
      },
      "links": [ {
         "rel": "self",
         "href":
          ...
      } ]
      "warnings": [ {
         "detail": "Warning from overridden validateEntity method in DeptImpl :
DeptName = ABC"
      }, {
         "detail": "Attribute set with value 92 for DeptNum in Dept failed",
         "o:errorCode": "27011",
         "o:errorPath": "/DeptNum"
      }, {
        "detail": "Warning from overridden afterCommit method in DeptViewImpl"
      }, {
        "detail": "Warning from overridden afterCommit method in DeptViewImpl"
      } ]
  }
}
```

Framework Version 7

The ADF REST runtime framework identified as version 7 is available starting with JDeveloper release 12.2.1.4.0. This framework version supports top-level LOVs, and fully removes row-level LOV resource descriptions in the ADF REST describe. Framework versions earlier than framework version 5 support nested LOVs in the context of a row in the REST resource. With framework version 7 enabled, use instead static LOV resources and row finder URL links in the describe, as described in Retrieving LOV-Enabled Attribute Values with Framework Version 5 and Later.

Additionally, version 7 removes LOV links from the resource item payload links section of the response. The goal is to enforce the use of parametrized row finders as implemented by version 5 and later by removing LOV links. For example, compare this response for a resource item request made with version 7 enabled to the same request with an earlier version.

```
{
   "EmpId" : 101,
   "EmpName" : "Bob",
   "@context" : {
      "key" : "1",
      "links" : [ {
        "rel" : "self",
        "href" : "http://server/demo/rest/12.0/Employees/101",
        "name" : "Employees",
        "kind" : "item"
      }, {
        "rel" : "canonical",
        "href" : "http://server/demo/rest/12.0/Employees/101",
        "name" : "Employees",
        "kind" : "item"
      } ]
   }
}
```

Here is the same response with an earlier version with the LOV resource link returned.

```
{
   "EmpId" : 101,
   "EmpName" : "Bob",
   "@context" : {
      "key" : "1",
      "links" : [ {
        "rel" : "self",
        "href" : "http://server/demo/rest/12.0/Employees/101",
        "name" : "Employees",
        "kind" : "item"
      }, {
        "rel" : "canonical",
        "href" : "http://server/demo/rest/12.0/Employees/101",
        "name" : "Employees",
        "kind" : "item"
      }, {
        "rel" : "lov",
        "href" : "http://server/demo/rest/12.0/Employees/10/lov/
JobsLOV",
        "name" : "JobsLOV",
        "kind" : "collection"
      } ]
   }
}
```

What You May Need to Know About the Advanced Query Syntax in ADF REST Framework Version 2

The ADF REST runtime framework identified as version 2 was offered starting with JDeveloper release 12.2.1.2.0. The purpose of this framework version is to introduce an expanded query expression syntax for ADF REST requests. This version will interpret the q query parameter value differently than the way framework version 1 does, and therefore introduces a backward incompatible change to service clients that rely on framework version 1.

When you decide to opt into framework version 2 (or later), the ADF REST runtime will process fetch requests for the q query parameter using the expanded expression syntax, whereas requests using the query-by-example syntax will become invalid and will return an error. However, you may decide not to opt into the functionality offered in framework version 2 by leaving the default for your release version set to framework version 1. Or, you may preserve the base functionality by creating a new release version identifier that you associate with framework version 2, while leaving the existing release identifier in the adf-config.xml file as framework version 1.

In version 1, filtering resource collections using query parameters is limited to a query-by-example syntax, which separates expressions using a semi-colon, as follows:

```
GET /rest/11.0/Departments?q=Dname SA*;Loc BOSTON
```

Whereas, starting in version 2, a new advanced query syntax supports filtering resource collections using rowmatch expressions, as follows:

```
GET /rest/11.1/Departments?q=Dname like 'SA*' or Loc = 'BOSTON'
```

Such rowmatch expressions include the case-sensitive name of a resource item, followed by an operator and one or more operand values (depending on the operator used). The filter can be as simple as a single expression, or it can combine expressions using the and and or conjunctions with matching sets of parentheses for grouping.

Benefits of the Advanced Query Syntax Offered in Framework Version 2 and Later

The advantages of rowmatch expression include the following.

- They may use supported operators:

  DepartmentNumber = 20

  DepartmentNumber <> 20

  DepartmentNumber <= 20

  DepartmentNumber < 20

  DepartmentNumber >= 20

DepartmentNumber >20

DepartmentNumber between 20 and 40

DepartmentNumber not between 20 and 40

DepartmentNumber in (20, 30, 40)

DepartmentName like '%S%'

DepartmentName like 'RE%'

DepartmentName not like 'RE%'

Location is null

Location is not null

- They may involve multiple attributes:

  DepartmentNumber = 10 or DepartmentName like 'RESEARCH'

  DepartmentNumber > 10 and DepartmentNumber < 40

  DepartmentNumber < 20 or DepartmentNumber > 30

  (DepartmentNumber = 10 or DepartmentNumber = 30) and (DepartmentName like 'SALES')

  DepartmentNumber BETWEEN 20 and 40) and (Location like 'DAL%')

  (DepartmentNumber > 0 and DepartmentNumber < 100) and (DepartmentName <> 'SALES') and (Location not like 'NEW%')

  (DepartmentNumber = 10 or DepartmentNumber = 30) and (DepartmentName = 'ACCOUNTING' or DepartmentName = 'SALES')

  (DepartmentNumber = 10 and DepartmentName like 'ACC%') or (DepartmentNumber = 20 and DepartmentName like 'RES%')

  DepartmentName='ACCOUNTING' or (DepartmentName like 'R%' and Location like '%ALLA%')

  (DepartmentName like 'R%' and Loc like '%ALLA%') or DepartmentName='ACCOUNTING'

  (DepartmentName like 'R%' or Loc like '%ALLA%') or DepartmentName='ACCOUNTING'

  (DepartmentNumber between 20 and 40) and DepartmentNumber is not null

- They may involve attributes of nested child resources:

  Deptno > 5 and Emps.Job = 'MANAGER'

  Emps.Job = 'MANAGER' and Deptno > 5

  Deptno > 5 and (Emps.Job = 'MANAGER')

  (Emps.Job = 'MANAGER') and Deptno > 5

  (Deptno > 5) and (Emps.Job = 'MANAGER')

  (Deptno = 10 and Emps.Job = 'PRESIDENT') or (Deptno = 20 and Emps.Job = 'MANAGER')

  Deptno > 5 and Emps.Job = 'MANAGER' and Emps.Sal >= 2500

  Deptno > 5 and (Emps.Job = 'ANALYST' or Emps.Sal >= 4000)

  (Deptno > 5 and Emps.Job = 'ANALYST') or Emps.Sal >= 4000

Emps.Job = 'ANALYST' or Emps.Job = 'SALESMAN'

Deptno > 5 and (Emps.Job = 'ANALYST' or Emps.Job = 'SALESMAN')

Deptno > 5 and Emps.Job = 'MANAGER' and Emps.DirectReports.Sal >= 2000

Deptno > 5 and (Emps.Job = 'MANAGER' or Emps.DirectReports.Sal >= 2000)

Deptno > 10 and (Emps.Job = 'MANAGER' and (Loc = 'NEW YORK' or Emps.Mgr=7698))

Deptno > 10 and (Emps.Job = 'MANAGER' or (Loc = 'NEW YORK' or Emps.Mgr=7698))

Deptno > 10 and (Emps.Job = 'MANAGER' or (Loc = 'NEW YORK' or Emps.Mgr=7698)) or Deptno = 40

Deptno > 10 and (Emps.Job = 'MANAGER' or (Loc = 'NEW YORK' or Emps.Mgr=7698)) or (Deptno = 40)

Deptno > 10 and (Emps.Job = 'MANAGER' or (Emps.DirectReports.Sal > 2000 and (Emps.DirectReports.Comm = 500 or Emps.DirectReports.Deptno > 10)))

Deptno > 10 and (Emps.Job = 'MANAGER' and (Emps.DirectReports.Sal >= 2000 and (Emps.DirectReports.Comm = 500 or Emps.DirectReports.Deptno > 10)))

- They may involve the UPPER function:

  UPPER(DepartmentName) = 'RESEARCH'

  UPPER(DepartmentName) = UPPER('research')

  UPPER(DepartmentName) like 'RES%' and UPPER(Location) like 'DAL%'

  UPPER(DepartmentName) like UPPER('research')

Overview of the Advanced Query Syntax Offered in Framework Version 2 and Later

The following are some examples of rowmatch expressions.

- To test whether a value is `null` you must use the `is null` or the `is not null` keywords:

  ```
  AssignedToId is null
  ```

  ```
  AssignedToId is not null
  ```

- For equality use the = sign, and for inequality use either the `!=` or the <> operators.

  ```
  AssignedToId = 100000000089003
  ```

  ```
  Priority != 1
  ```

  ```
  Priority <> 1
  ```

  ```
  ActivityType != 'RS'
  ```

  ```
  ActivityType <> 'RS'
  ```

- For relational comparisons, use the familiar <, <=, >, or <> operators, along with `between` or `not between`.

  ```
  Priority <= 2
  ```

  ```
  Priority < 3
  ```

  ```
  Priority <> 1
  ```

```
Priority > 1
```

```
Priority >= 1
```

```
TotalLoggedHours >= 12.75
```

```
Priority between 2 and 4
```

```
Priority not between 2 and 4
```

- For string matching, you can use the `like` operator, employing the percent sign `%` as the wildcard character to obtain "starts with", "contains", or "ends with" style filtering, depending on where you place your wildcard(s):

  ```
  RecordName like 'TT-%'
  ```

  ```
  RecordName like '%-TT'
  ```

  ```
  RecordName like '%-TT-%'
  ```

- To test whether a field's value is in a list of possibilities, you can use the `in` operator:

  ```
  ActivityType in ('OC','IC','RS')
  ```

- You can combine expressions using the conjunctions `and` and `or` along with matching sets of parentheses for grouping to create more complex filters like:

  ```
  (AssignedToId is null) or ( (Priority <= 2) and (RecordName like
  'TT-99%'))
  ```

  ```
  (AssignedToId is not null) and ( (Priority <= 2) or (RecordName like
  'TT-99%'))
  ```

  When using the `between` or `in` clauses, you must surround them by parentheses when you join them with other clauses using `and` or `or` conjunctions.

What Happens At Runtime: Invoking an ADF REST Framework Version

After Oracle JDeveloper introduces a new ADF REST framework version (such as framework version 2 offered in JDeveloper release 12.2.1.2.0 and later), you may want to create a new resource version and associate it with the new framework version to ensure that service clients make use of the appropriate level of functionality.

For example, the following URL specifies resource version 11.2, where the application’s `adf-config.xml` file associates this resource version with version 2 of the ADF REST framework. Because ADF REST framework version 2 supports rowmatch filter expressions, as a result, the ADF REST runtime will process the request with the appropriate functionality and fetch the departments with names that begin with SA (for example, SALES) or have a Location of BOSTON.

```
http://server/demo/rest/11.2/Departments?q=DepartmentName like 'SA*' or
Location = 'BOSTON'
```

Note that a service client may override the default framework version and process individual requests using a specified framework version that gets passed in a custom header, `REST-Framework-Version`. In this case, the ADF REST runtime will ignore the application default framework version declaration. If the custom header is omitted on the request, then the ADF REST runtime always uses the application’s default framework version, as defined in the `adf-config.xml` file. When the application does not define a default framework version and the request on the service client omits the version header, then version 1 of the ADF REST framework is assumed. For information about processing requests using the framework versions, see Working with ADF REST Framework Versions.

The following table contrasts what happens at runtime when a request uses functionality supported (or not supported) by a particular ADF REST framework version. In particular, note the URLs that contrast the query-by-example syntax supported only in version 1 with rowmatch expressions supported in version 2 (and later). The column for the `REST-Framework-Version` header specifies whether the request passes a version header on the request. When the request includes the version header, the ADF REST runtime processes the request with the passed framework value and overrides the default `adf-config.xml` file declaration.

Table 16-3 Example Requests Processed By ADF REST Framework Versions

URI	REST-Framework-Version Header	Resolved Version	Success / Error Notes
GET /rest/11.0/Departments		1	Success. Uses default framework version for resource version 11.0.
GET /rest/11.0/Departments	1	1	Success. Uses the requested framework version, which matches the default declaration in the `adf-config.xml` file.
GET /rest/11.0/Departments	2	2	Success. Uses the requested framework version, which overrides the default declaration in the `adf-config.xml` file.
GET /rest/11.1/Departments		2	Success. Uses the default framework version for resource version 11.1.
GET /rest/11.0/Departments?q=Dname SA*;Loc BOSTON		1	Success. Uses the default framework version for resource version 11.0, which is defined as version 1. Query by example syntax is supported in framework version 1.
GET /rest/11.1/Departments?q=Dname SA*;Loc BOSTON		2	Error is returned. Uses the default framework version for resource version 11.1, which is defined a version 2. Query by example syntax is not supported in framework version 2.

Table 16-3 (Cont.) Example Requests Processed By ADF REST Framework Versions

URI	REST-Framework-Version Header	Resolved Version	Success / Error Notes
GET /rest/11.0/ Departments? q=Dname LIKE 'SA*' OR Loc = 'BOSTON'	2	2	Success. Uses the requested framework version, which overrides the default declaration in the `adf-config.xml` file. Rowmatch expressions are supported in framework version 2.
GET /rest/11.0/ Departments? q=Dname LIKE 'SA*' OR Loc = 'BOSTON'		1	Error is returned. Uses the default framework version for resource version 11.1, which is defined as version 1. Rowmatch expressions are not supported in framework version 1.
GET /rest/11.1/ Departments? q=Dname LIKE 'SA*' OR Loc = 'BOSTON'	1	1	Error is returned. Uses the requested framework version, which overrides default declaration in the `adf-config.xml` file. Rowmatch expressions are not supported in framework version 1.

Granting Client Access to the ADF REST Resource

Use the Configure ADF Security wizard and in the wizard select RESTWebService.jpr as the web project in the wizard to manage ADF REST resource security.

You can enable security for the ADF REST resources when you want to require user authentication to invoke actions on the REST request. Security is handled like all Fusion web applications and is enabled when you run the Configure ADF Security wizard and select the **RESTWebService.jpr** as the web project in the wizard. Enabling ADF Security on the web service project, means the deployed service will be inaccessible by default and users will require security grants to application roles that confer specific access permission.

Permission that you can grant to users of the RESTful web service is defined by the following classes:

- `oracle.adf.share.security.authorization.RestServicePermission` which defines the authorization grants to the standard actions defined in a REST resource.

Table 16-4 lists the actions defined by the ADF REST permission class that you may grant to your defined application roles.

Table 16-4 Secured Actions of ADF REST Resources

Grantable Action	Effect on the REST Resource
`describe`	Grants describing the collections and instances of the REST resource.
`get`	Grants returning the collection or instance for the REST resource.
`update`	Grants updating a collection or instance for the REST resource.
`delete`	Grants deleting a collection or instance for the REST resource.
`create`	Grants creating a collection or instance for the REST resource.

The general process to authorize users to invoke actions of the ADF REST resource is as follows.

1. Enable the desired standard actions on the REST resource.
2. Run the ADF Security wizard on the **RESTWebService** project.
3. Create one or more application roles and test users to test security in JDeveloper.
4. Create a resource grant for the REST resource that you associate with the desired application role.
5. Map the application roles and their resource grants to your test users.

How to Create a Resource Grant for ADF REST Resources

You can use the overview editor for security policies to create a grant that will secure operations that you have exposed on the resource. JDeveloper will match your resource type definition to the OPSS permission class `oracle.adf.share.security.authorization.RestServicePermission`, as described in Table 16-4.

Before you begin:

It may be helpful to have an understanding of how the ADF REST framework supports application modules and enables service clients to access rows of data and perform service operations. See About RESTful Web Services and ADF Business Components.

You may also find it helpful to understand other Oracle ADF features that depend on ADF REST resources. For more information, see See Additional Functionality for RESTful Web Services.

You will need to complete these tasks:

- Create the desired REST resource, as described in How to Create ADF REST Resources from View Object Instances.
- Enable the standard operations that you want the REST resource to secure, as described in How to Expose Standard HTTP Actions in the ADF REST Resource.

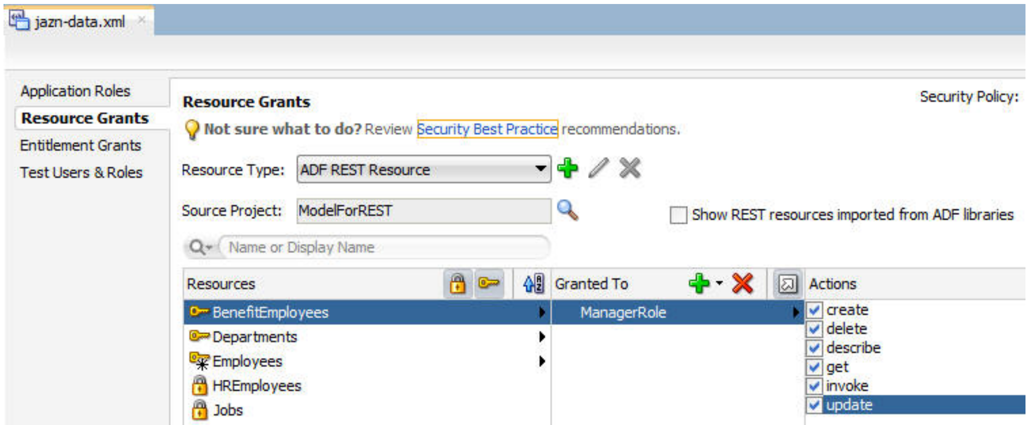
- Run the Configure ADF Security wizard to enforce security and make the REST resource methods protected by default. In the Authentication Type page of the wizard instead of selecting **ViewController**, as described in Enabling ADF Security, select the default generated **RESTWebService** project from the **Web Project** dropdown to configure ADF Security for a ADF Business Components model project that defines the ADF REST resources.
- Create one or more test users and map them to application roles that you create for the purpose of granting resource permissions, as described in How to Associate Test Users and Application Roles.

To create the ADF REST resource grant:

1. In the main menu, choose **Application** and then **Secure > Resource Grants**.
2. In the Resource Grants page of the overview editor for security policies, choose **ADF REST Resource** from the **Resource Type** dropdown.
3. Locate the Model project that defines the resource and add it to the **Source Project** field.
4. In the **Resources** column, select the resource.
5. In the **Granted To** column, click the **Add Grantee** icon and choose **Add Application Role** and then, in the Select Application Roles dialog, click the **Add** icon and choose the application role to receive the resource grants. For example, if you created a role `ManagerRole`, select the role and click **OK**.
6. With the role selected in the **Granted To** column, select the desired actions from the **Actions** list.

   The overview editor displays the resource grant made to the application role, as shown in Figure 16-22.

Figure 16-22 Creating a Grant for an ADF REST Resource



Resource Grants

Not sure what to do? Review Security Best Practice recommendations.

Resource Type: ADF REST Resource

Source Project: ModelForREST

☐ Show REST resources imported from ADF libraries

Name or Display Name

Resources	Granted To	Actions
BenefitEmployees	ManagerRole	☑ create
Departments		☑ delete
Employees		☑ describe
HREmployees		☑ get
Jobs		☑ invoke
		☑ update

Deploying the ADF REST Resource

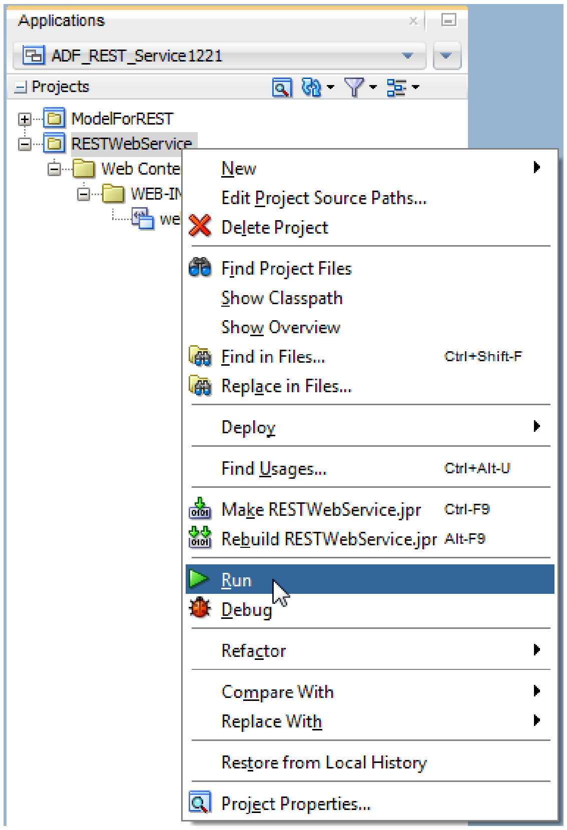
You can deploy the RESTful web service to either the Integrated Oracle WebLogic Server of JDeveloper or to the standalone Oracle WebLogic Server.

You can deploy the RESTful web service to either JDeveloper's Integrated WebLogic Server or to standalone Oracle WebLogic Server.

How to Deploy the ADF REST Resource to Integrated WebLogic Server

You can deploy the RESTful web service to JDeveloper's Integrated WebLogic Server using the default **Run** option that you select on the **RESTWebService** project, as shown in Figure 16-23.

Figure 16-23 Using the Default Run Option on RESTWebService Project



Once you have deployed the service to Integrated WebLogic Server, you can click the target URL in the Log window to access the service in the HTTP Analyzer tool in JDeveloper and to perform testing using the syntax that establishes the request (GET or POST, for example). For details about testing RESTful web services, see Testing the ADF REST Resources Using Integrated WebLogic Server.

Before you begin:

It may be helpful to have an understanding of how the ADF REST framework supports application modules and enables service clients to access rows of data and perform service operations. For more information, see About RESTful Web Services and ADF Business Components.

You may also find it helpful to understand other Oracle ADF features that depend on ADF REST resources. For more information, see Additional Functionality for RESTful Web Services.

You will need to complete these tasks:

- Create the desired REST resource, as described in Creating ADF REST Resources Using the Application Module.
- Optionally, create an Integrated WebLogic Server connection and configure the default domain, as described in Working with the Default Domain. If you do not create the connection, JDeveloper will prompt you to do so.
- Optionally, edit the REST servlet's URL pattern and context root as you want it to appear in the target URL used to access the deployed RESTful web service. For more information about editing the URL pattern, see What You May Need to Know About Modifying the Target URL Used by Service Clients.

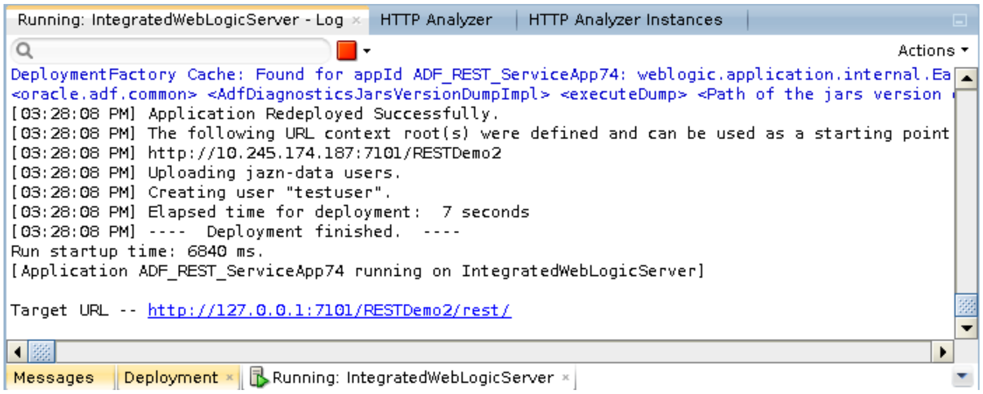
To deploy the web service to Integrated WebLogic Server:

1. In the Applications window, select the **RESTWebService** project node, right-click and choose **Run**.

   If you have not already configured Integrated WebLogic Server, JDeveloper prompts you to configure the default domain.

   JDeveloper deploys the application and starts the web service. During this time, the output from these processes is displayed in the **Running: IntegratedWebLogicServer** tab of the Log window. After the web service has started, the target URL appears at the bottom of the log, as shown in Figure 16-24.

   Figure 16-24   Log Window Displays Target URL

   Running: IntegratedWebLogicServer - Log
HTTP Analyzer
HTTP Analyzer Instances
Actions
Messages
Deployment
Running: IntegratedWebLogicServer

   ```
   DeploymentFactory Cache: Found for appId ADF_REST_ServiceApp74: weblogic.application.internal.Ea
   <oracle.adf.common> <AdfDiagnosticsJarsVersionDumpImpl> <executeDump> <Path of the jars version
   [03:28:08 PM] Application Redeployed Successfully.
   [03:28:08 PM] The following URL context root(s) were defined and can be used as a starting point
   [03:28:08 PM] http://10.245.174.187:7101/RESTDemo2
   [03:28:08 PM] Uploading jazn-data users.
   [03:28:08 PM] Creating user "testuser".
   [03:28:08 PM] Elapsed time for deployment:  7 seconds
   [03:28:08 PM] ----  Deployment finished.  ----
   Run startup time: 6840 ms.
   [Application ADF_REST_ServiceApp74 running on IntegratedWebLogicServer]

   Target URL -- http://127.0.0.1:7101/RESTDemo2/rest/
   ```

2. After you deploy the web service, in the **Running: IntegratedWebLogicServer** tab of the Log window, click the target URL to access the web service in the HTTP Analyzer and to perform testing on the deployed service in JDeveloper.

   For example, the **Running: IntegratedWebLogicServer** tab of the Log window might display the following target URL that you can click, where 7101 is the port number and what follows is the service context root and the REST servlet URL pattern:

   `http://<hostname>:7101/RESTSample_App-context-root/rest`

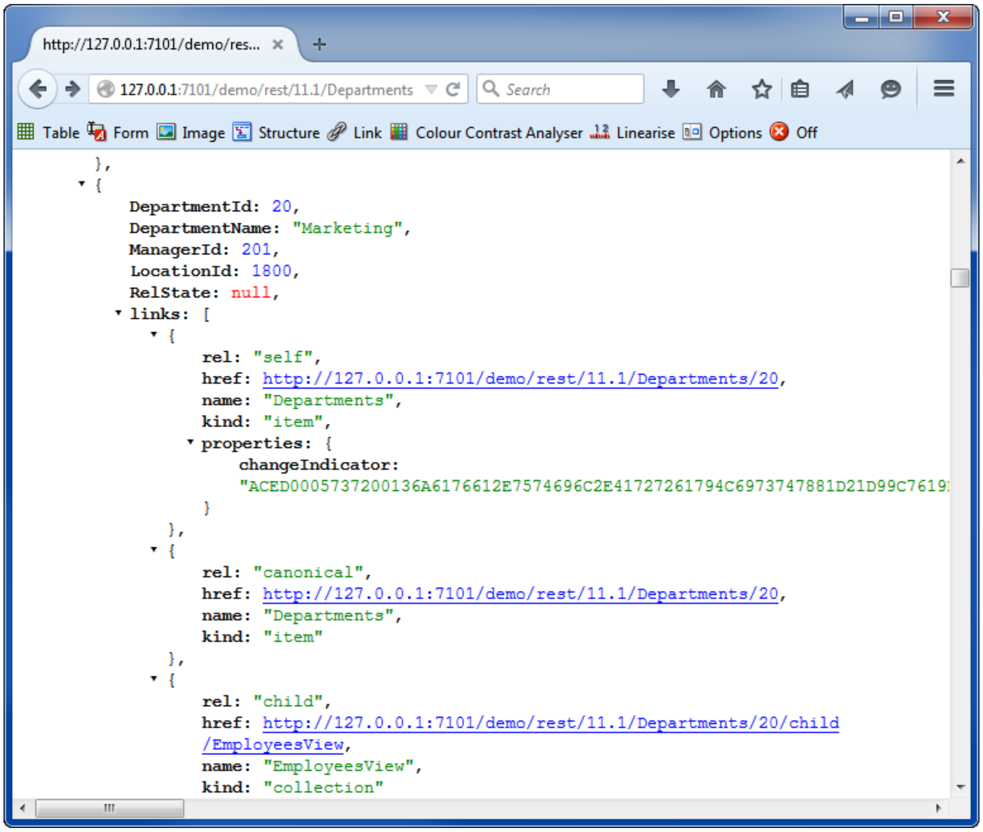
3. Alternatively, to display the web service using a web browser, paste the target URL the Log window into the browser address field, modify the URL to identify the ADF REST resource, and submit the HTTP Get request.

   For example, a Get request for version `11.1` of the `Departments` resource looks like this:

```
http://localhost:7101/RESTSample_App-context-root/rest/11.1/
Departments
```

Be sure to include the version number that you configured for the resource in the URL. In this example, `11.1` was configured as the version for the `Departments` resource, as shown in Figure 16-25.

Figure 16-25   Using the Browser to Make a GET REST Request



```
        },
      ▾ {
            DepartmentId: 20,
            DepartmentName: "Marketing",
            ManagerId: 201,
            LocationId: 1800,
            RelState: null,
          ▾ links: [
              ▾ {
                    rel: "self",
                    href: http://127.0.0.1:7101/demo/rest/11.1/Departments/20,
                    name: "Departments",
                    kind: "item",
                  ▾ properties: {
                        changeIndicator:
                        "ACED0005737200136A6176612E7574696C2E41727261794C6973747881D21D99C7619
                    }
                },
              ▾ {
                    rel: "canonical",
                    href: http://127.0.0.1:7101/demo/rest/11.1/Departments/20,
                    name: "Departments",
                    kind: "item"
                },
              ▾ {
                    rel: "child",
                    href: http://127.0.0.1:7101/demo/rest/11.1/Departments/20/child
                    /EmployeesView,
                    name: "EmployeesView",
                    kind: "collection"
```

How to Configure the ADF REST Resources Deployment Profile For Standalone Oracle WebLogic Server Deployment

Before you can deploy the RESTful web service to standalone Oracle WebLogic Server, you must configure the ADF REST resources deployment profile to declare the RESTWebService project as a dependent module of the ADF REST application EAR file. This project contains a `web.xml` file and a `weblogic.xml` file that ensure access to the REST resources by service clients.

In JDeveloper, you use the Application Assembly page of the Edit EAR Deployment Profile Properties dialog to add the module dependency to the application. Then, when you deploy the ADF REST application, the two files of the selected RESTWebService project will be assembled as a WAR file in the deployed EAR file.

Before you begin:

It may be helpful to have an understanding of how the ADF REST framework supports application modules and enables service clients to access rows of data and perform service operations. For more information, see About RESTful Web Services and ADF Business Components.

You may also find it helpful to understand other Oracle ADF features that depend on ADF REST resources. For more information, see Additional Functionality for RESTful Web Services.

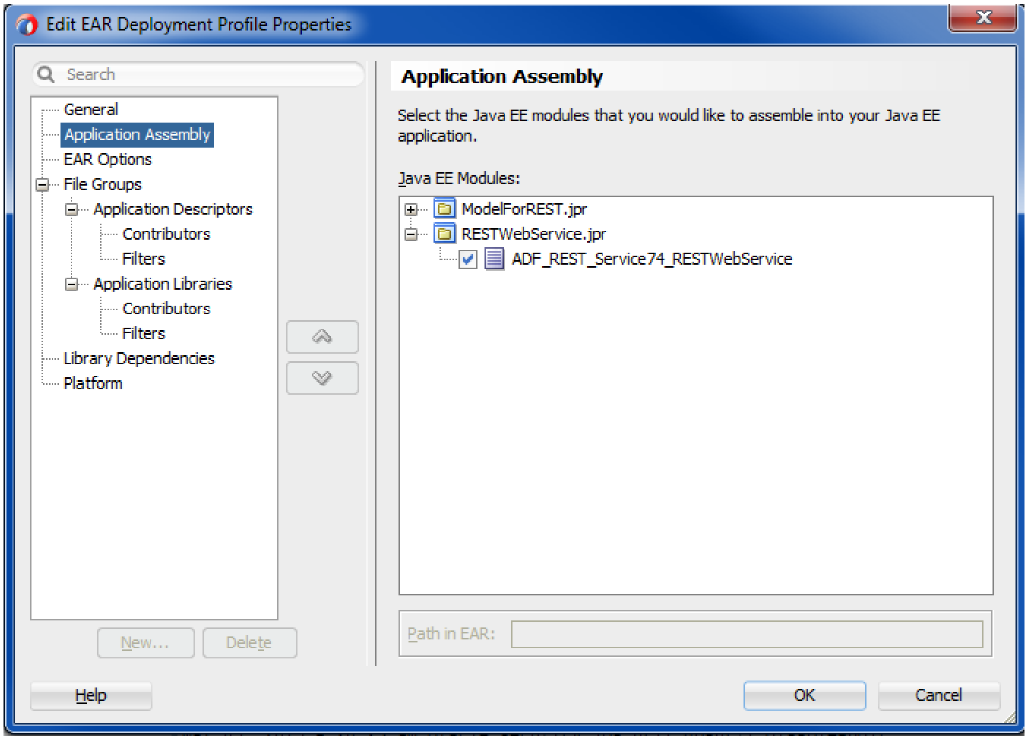
You will need to complete these tasks:

- Create the desired REST resource, as described in Creating ADF REST Resources Using the Application Module.

To configure the deployment profile:

1. Add the RESTWebService project as a dependency for the application's EAR deployment profile:
   a. In the Applications window, select the application and then choose **Application - Application Properties** from the main menu.
   b. In the Application Properties dialog, choose **Deployment** and select the application module deployment profile and click **Edit Profile**.
   c. In the Edit EAR Deployment Profile Properties dialog, choose **Application Assembly** and expand the **RESTWebService.jpr** module and then select the project as shown in Figure 16-26 and click **OK**.

Figure 16-26 Adding a Deployment Dependency on RESTWebService Project



Edit EAR Deployment Profile Properties

Application Assembly

Select the Java EE modules that you would like to assemble into your Java EE application.

Java EE Modules:

- ModelForREST.jpr
- RESTWebService.jpr
  - ☑ ADF_REST_Service74_RESTWebService

Path in EAR:

How to Deploy the ADF REST Resource to Standalone Oracle WebLogic Server

You can deploy the RESTful web service to standalone Oracle WebLogic Server.

Before you begin:

It may be helpful to have an understanding of how the ADF REST framework supports application modules and enables service clients to access rows of data and perform service operations. For more information, see About RESTful Web Services and ADF Business Components.

You may also find it helpful to understand other Oracle ADF features that depend on ADF REST resources. For more information, see Additional Functionality for RESTful Web Services.

You will need to complete these tasks:

- Create the desired REST resource, as described in Creating ADF REST Resources Using the Application Module.
- Modify the deployment profile to add the RESTWebService project as a dependency and deploy the web service, as described in How to Configure the ADF REST Resources Deployment Profile For Standalone Oracle WebLogic Server Deployment.
- When the application uses the customization features provided by Oracle Metadata Services (MDS) framework and shaping definitions are applied to ADF REST resources, it is necessary to modify the MAR profile for the customization features, as described in What You May Need to Know About Deploying ADF REST Resources With MDS Customization Support.
- Optionally, edit the REST servlet's URL pattern and context root as you want it to appear in the target URL used to access the deployed RESTful web service. For more information about editing the URL pattern, see What You May Need to Know About Modifying the Target URL Used by Service Clients.

To deploy the web service to standalone Oracle WebLogic Server:

1. In the Applications window, select the application and then choose **Application - Deploy - *deployment_profile* to *server_name*** from the main menu.

   JDeveloper deploys the application and starts the web service. During this time, the output from these processes is displayed in the **Deployment** tab of the Log window. After the web service has started, the target URL is also displayed in the Log window.

2. After you deploying the web service, open the Deployment - Log window and click the target URL to access the web service and to perform testing on the deployed service.

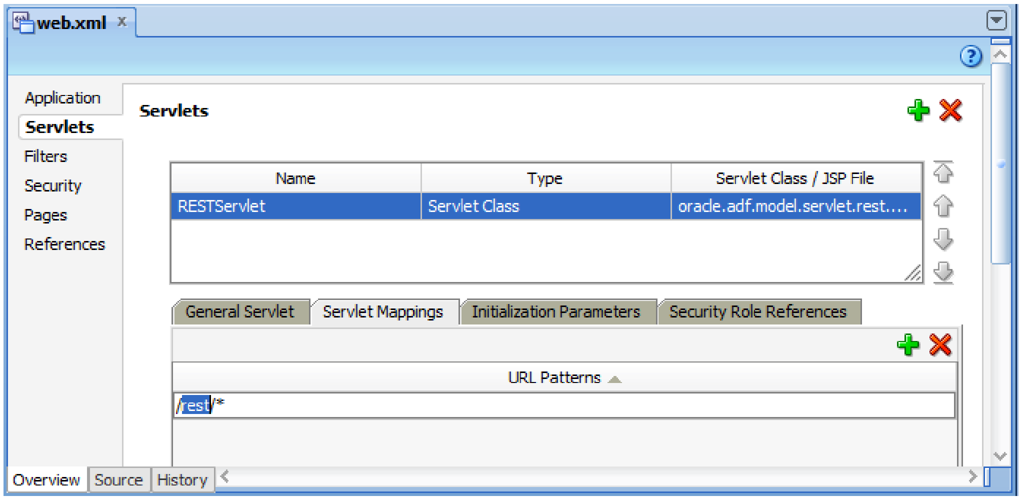
What You May Need to Know About Modifying the Target URL Used by Service Clients

You can edit the REST servlet's URL pattern as you want it to appear in the URL that service clients use to access the service. By default the servlet mapping uses `rest` in the URL pattern:

```
http://localhost:7101/RESTSample_App-context-root/rest/11.1/Departments
```

To edit the servlet mapping pattern, in the Applications window, expand the **RESTWebService** project and double-click the `web.xml` file, and then in the editor for the `web.xml` file, edit the URL pattern for the REST servlet, as shown in Figure 16-27.

Figure 16-27 Editing the URL Pattern for the REST Servlet Mapping

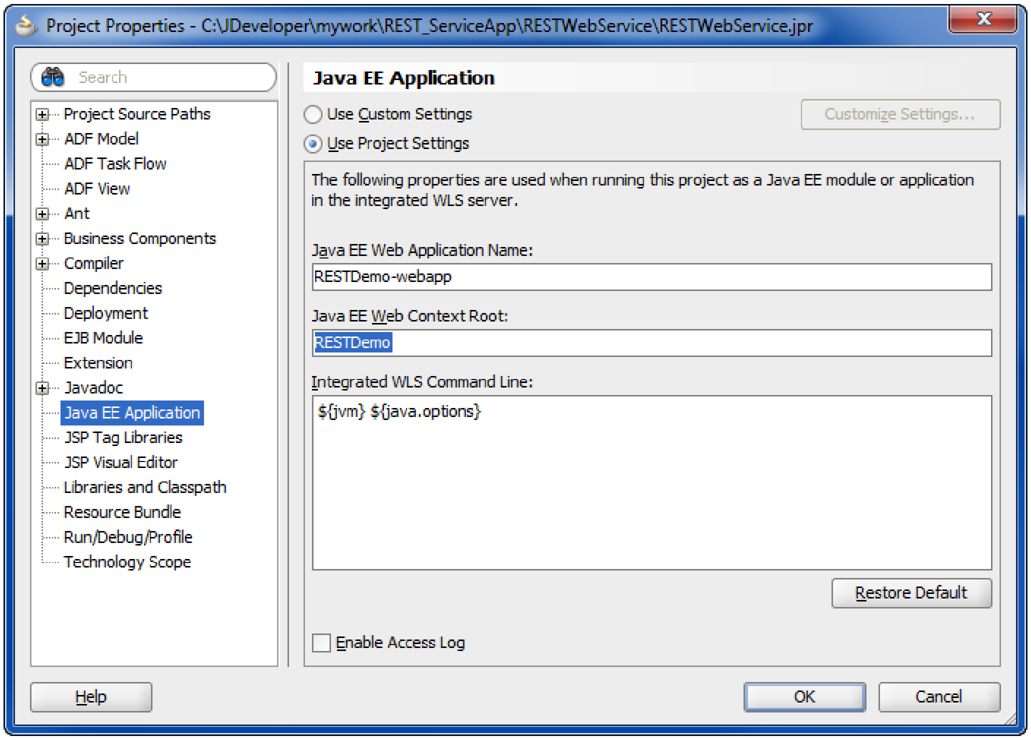


Name	Type	Servlet Class / JSP File
RESTServlet	Servlet Class	oracle.adf.model.servlet.rest....

URL Patterns
/rest/*

You may also edit the context root as you want it to appear in the URL to access the service. By default, the context root is based on the application name and may be shortened as desired. To edit the context root, in the Applications window, right-click the **RESTWebService** project and choose **Project Properties**, and then, in the Project Properties dialog, click **Java EE Application** and edit the **Java EE Context Root** as shown in Figure 16-28 and click **OK**.

Figure 16-28 Editing the Default Context Root on the RESTWebService Project



Project Properties - C:\JDeveloper\mywork\REST_ServiceApp\RESTWebService\RESTWebService.jpr

Java EE Application

○ Use Custom Settings

● Use Project Settings

The following properties are used when running this project as a Java EE module or application in the integrated WLS server.

Java EE Web Application Name: RESTDemo-webapp

Java EE Web Context Root: RESTDemo

Integrated WLS Command Line:

```
${jvm} ${java.options}
```

□ Enable Access Log

What You May Need to Know About Deploying ADF REST Resources With MDS Customization Support

If your Model project defines view object shaping definitions and the application supports end user customizations using features of the Oracle Metadata Services (MDS) framework, it is necessary to create a MAR profile that includes the `/persdef/` name in the `user` directory.

This is necessary because MDS customizations define the namespace path shared by the ADF REST shaping definitions:

```
<namespace path="/persdef/" metadata-store-usage="reposX"/>
```

In this case, the `adf-config.xml` file should define `deploy-target="true"` for the corresponding `metadata-store-usage` element.

For details about how to package the customized metadata, see How to Package and Deploy Customized Applications.

Testing the ADF REST Resources Using Integrated WebLogic Server

JDeveloper allows you to test an ADF REST resource deployed on the Integrated WebLogic Server. Use the HTTP Analyzer to create an HTTP request, web browser for GET requests, or cURL command line tool.

You use JDeveloper to deploy the RESTful web service to Integrated WebLogic Server and then access the ADF REST resources using the target URL generated by the deployment process. Using the generated URL, you can make requests to test the ADF REST resources of the ADF Model project.

You can use any of the following techniques to test the deployed RESTful web service by modifying the URL of the deployed service to return an expected payload:

- In the HTTP Analyzer in JDeveloper, using the **Create HTTP Request** option
- In a web browser (typically limited to GET requests)
- In the cURL command line tool from a command window

  Note that cURL is a third-party tool that you can install and configure to make REST requests using a command window.

Testing the ADF resource using the deployed RESTful web service requires knowledge of the REST API syntax enabled by the ADF REST runtime. However, you can use the instructions in this document to make a basic GET request to view a resource collection.

For example, a GET request that returns all versions of all resources looks like this:

```
http://<server>/<context-root>/rest
```

For example, a GET request that returns all resources of a specific version looks like this:

```
http://<server>/<context-root>/rest/<versionID>/describe
```

For example, a GET request that returns a specific version of a specific resource collection looks like this:

```
http://<server>/<context-root>/rest/<versionID>/<resourceName>
```

Before you begin:

It may be helpful to have an understanding of how the ADF REST framework supports application modules and enables service clients to access rows of data and perform service operations. See About RESTful Web Services and ADF Business Components.

You may also find it helpful to understand the ADF REST framework and the syntax for making API calls using HTTP methods. See Consuming ADF RESTful Web Services.

You may also find it helpful to understand other Oracle ADF features that depend on ADF REST resources. See Additional Functionality for RESTful Web Services.

You will need to complete these tasks:

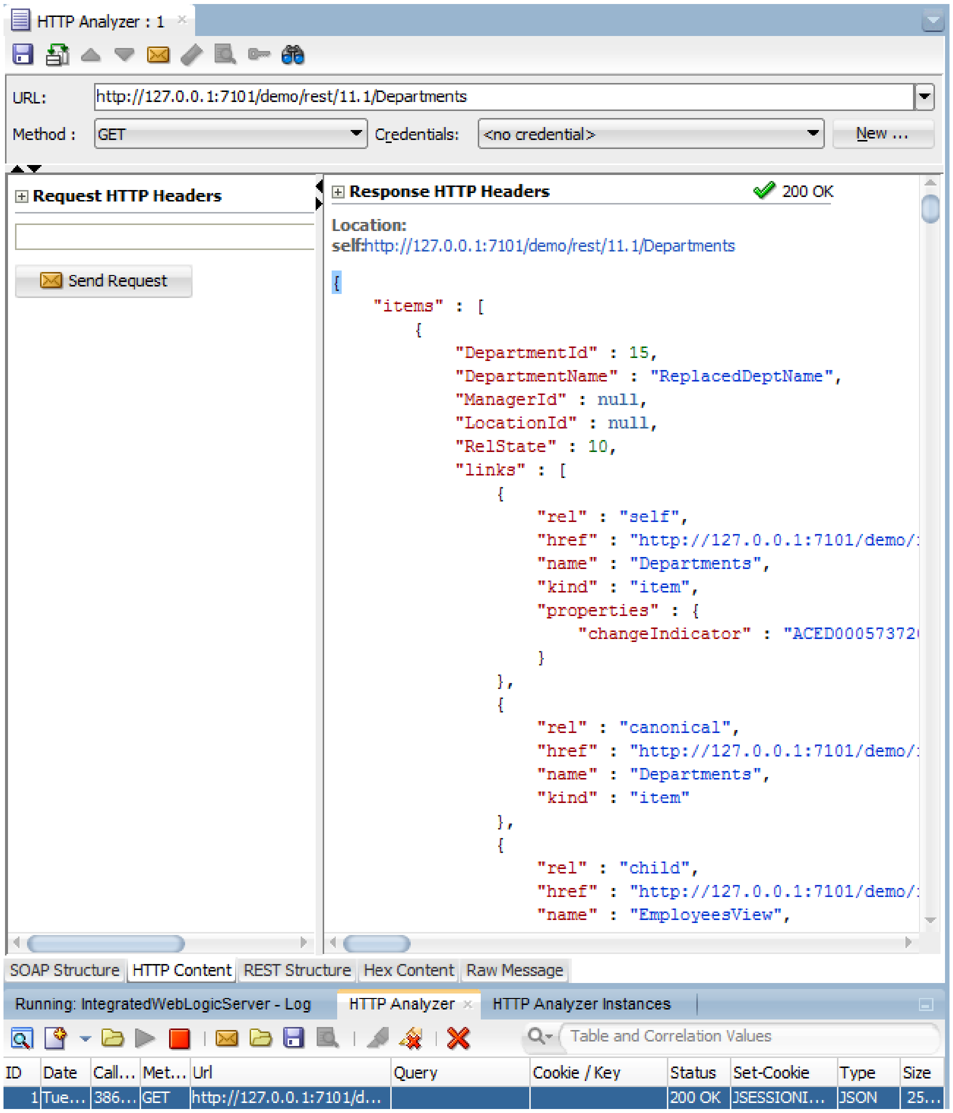
- Create the desired REST resource, as described in Creating ADF REST Resources Using the Application Module.
- Deploy the ADF REST resources to Integrated WebLogic Server, as described in How to Deploy the ADF REST Resource to Integrated WebLogic Server.
- Download the command line tool cURL when you want to use the tool to test the RESTful web service operations against REST resources.

To test the web service using HTTP GET:

1. To test using JDeveloper's HTTP Analyzer, choose **Tools - HTTP Analyzer** from the main menu, then choose **Create New HTTP Request** in the HTTP Analyzer Log window and then, in the HTTP Analyzer, select the HTTP Content tab and paste the target URL that you copied from the Log window into the **URL** field, modify the URL to identify the REST resource, then choose the desired **Method** and click **Send Request**.

   Be sure to include the version number that you configured for the resource in the URL. In this example, `11.1` is the release name version identifier used to access the `Departments` resource, as shown in Figure 16-29.

Figure 16-29   Using the HTTP Analyzer to Make a GET REST Request



URL: http://127.0.0.1:7101/demo/rest/11.1/Departments

Method : GET   Credentials: <no credential>

Location:
self:http://127.0.0.1:7101/demo/rest/11.1/Departments

```
{
    "items" : [
        {
            "DepartmentId" : 15,
            "DepartmentName" : "ReplacedDeptName",
            "ManagerId" : null,
            "LocationId" : null,
            "RelState" : 10,
            "links" : [
                {
                    "rel" : "self",
                    "href" : "http://127.0.0.1:7101/demo/:
                    "name" : "Departments",
                    "kind" : "item",
                    "properties" : {
                        "changeIndicator" : "ACED00057372
                    }
                },
                {
                    "rel" : "canonical",
                    "href" : "http://127.0.0.1:7101/demo/:
                    "name" : "Departments",
                    "kind" : "item"
                },
                {
                    "rel" : "child",
                    "href" : "http://127.0.0.1:7101/demo/:
                    "name" : "EmployeesView",
```

ID	Date	Call...	Met...	Url	Query	Cookie / Key	Status	Set-Cookie	Type	Size
1	Tue...	386...	GET	http://127.0.0.1:7101/d...			200 OK	JSESSIONI...	JSON	25...

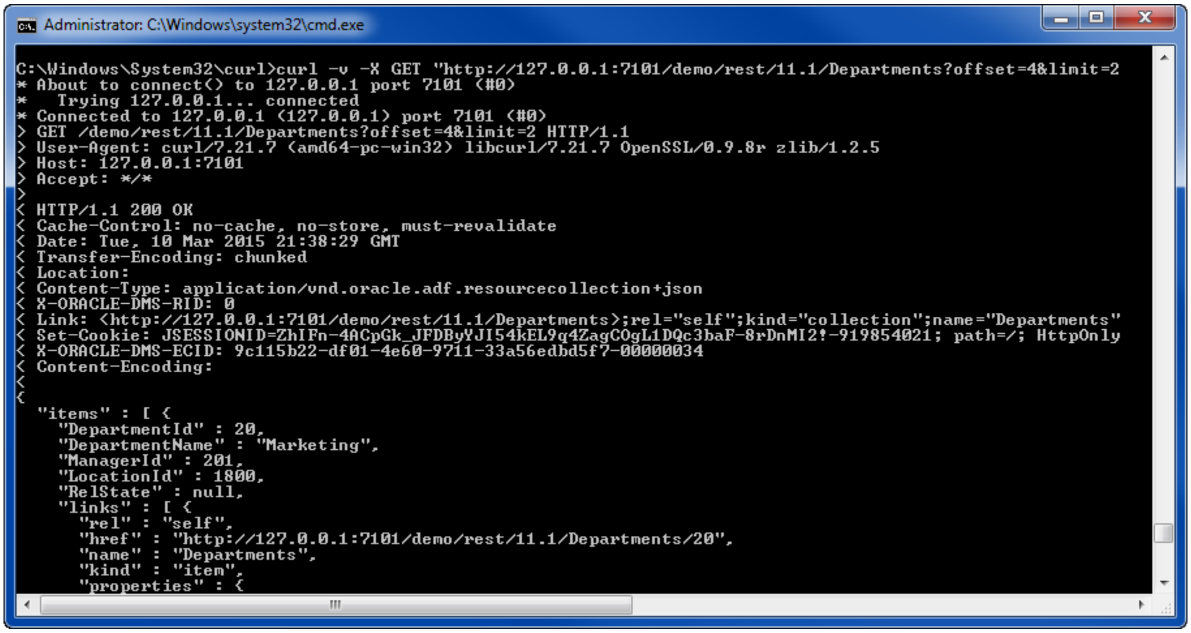
Note that the HTTP Analyzer supports SOAP, standard HTTP, and REST type requests. However, the WADL that you might normally supply to the HTTP Analyzer to define a generic RESTful web service does not support the describe format of ADF REST resources. Therefore when using the HTTP Analyzer to test the deployed ADF REST resource, you must create a new HTTP type request instead of a REST request.

2. To test using the third-party cURL command line tool, open a command prompt and enter a cURL command for the REST resource similar to the following, where `-v` is the option to see what cURL sends and `-X` is the request option:

   ```
   curl -v -X GET http://127.0.0.1:7101/RESTDemo/rest/11.1/Departments
   ```

   Be sure to include the version number that you configured for the resource in the URL. In this example, `11.1` is the release name version identifier used to access the `Departments` resource, as shown in Figure 16-30.

Figure 16-30   Using cURL to Make a REST Request



```
C:\Windows\System32\curl>curl -v -X GET "http://127.0.0.1:7101/demo/rest/11.1/Departments?offset=4&limit=2
* About to connect() to 127.0.0.1 port 7101 (#0)
*   Trying 127.0.0.1... connected
* Connected to 127.0.0.1 (127.0.0.1) port 7101 (#0)
> GET /demo/rest/11.1/Departments?offset=4&limit=2 HTTP/1.1
> User-Agent: curl/7.21.7 (amd64-pc-win32) libcurl/7.21.7 OpenSSL/0.9.8r zlib/1.2.5
> Host: 127.0.0.1:7101
> Accept: */*
>
< HTTP/1.1 200 OK
< Cache-Control: no-cache, no-store, must-revalidate
< Date: Tue, 10 Mar 2015 21:38:29 GMT
< Transfer-Encoding: chunked
< Location:
< Content-Type: application/vnd.oracle.adf.resourcecollection+json
< X-ORACLE-DMS-RID: 0
< Link: <http://127.0.0.1:7101/demo/rest/11.1/Departments>;rel="self";kind="collection";name="Departments"
< Set-Cookie: JSESSIONID=ZhIFn-4ACpGk_JFDByYJI54kEL9q4ZagCOgL1DQc3baF-8rDnMI2!-919854021; path=/; HttpOnly
< X-ORACLE-DMS-ECID: 9c115b22-df01-4e60-9711-33a56edbd5f7-00000034
< Content-Encoding:
<
{
  "items" : [ {
    "DepartmentId" : 20,
    "DepartmentName" : "Marketing",
    "ManagerId" : 201,
    "LocationId" : 1800,
    "RelState" : null,
    "links" : [ {
      "rel" : "self",
      "href" : "http://127.0.0.1:7101/demo/rest/11.1/Departments/20",
      "name" : "Departments",
      "kind" : "item",
      "properties" : {
```

17 Extending Business Components Functionality

This chapter describes techniques that you can use to incorporate custom code with all types of ADF Business Components and to extend the ADF Business Components framework behavior.

This chapter includes the following sections:

About Extending Business Components Functionality

You can extend base classes of ADF Business Components to incorporate custom code.

One of the powerful features of framework-based development is the ability to extend the base framework to change a built-in feature to behave differently or to add a new feature that can be used by all of your applications.

The base classes of the **ADF Business Components** framework may be extended to incorporate custom code with all types of components in the framework and to extend the ADF Business Components framework behavior.

When used without customization, your business component is completely defined by its XML document and it will be fully functional without custom Java code or even a Java class file for the component. If you have no need to extend the built-in functionality of a component in ADF Business Components, and no need to write any custom code to handle its built-in events, you can use the component in this XML-only fashion. However, when you do extend base classes of the ADF Business Components framework, you can still work with XML documents in JDeveloper.

Once you have created framework extension classes, any new business component you create can be based on your *customized* framework class instead of the base one. And, you can always update the definitions of existing components to use the new framework extension class as well.

Additional Functionality for Extending Business Components

You may find it helpful to understand other **Oracle ADF** features before you start working with the ADF Business Components framework. Following are links to other functionality that may be of interest.

- For details about creating a reusable library to make your framework extension layer classes easier to package, see Reusing Application Components .
- For details about using the customization features provided by Oracle Metadata Services (MDS) to create applications that can be customized and subsequently deployed by a customer, see Customizing Applications with MDS .
- For a quick reference to the most common code that you will typically write, use, and override in your custom classes, see Most Commonly Used ADF Business Components Methods.
- For API documentation related to the `oracle.jbo` package, see the following Javadoc reference document:
  - *Java API Reference for Oracle ADF Model*

Creating ADF Business Components Extension Classes

An ADF Business Components extension class augments or modifies behavior of built-in features and also allows you to provide a generic workaround for a bug.

An ADF Business Components framework extension class is a Java class you write that extends one of the framework's base classes to:

- Augment a built-in feature with additional, generic functionality
- Change how a built-in feature works, or even to
- Workaround a bug you encounter in a generic way

Before you begin to develop application-specific business components, Oracle recommends that you consider creating a complete layer of framework extension classes and set up your project-level preferences to use that layer by default. You might not have any custom code in mind to put in these framework extension classes initially, but this practice will help when customization becomes practical.

This way, substantial inconvenience can be avoided if you discover mid-project that all of your **entity objects**, for example, require a new generic feature, augmented built-in feature, or a generic bug workaround.

> **Note:**
>
> To experiment with the examples in this chapter, use the `SummitADF_Examples` workspace, as described in Running the Standalone Samples from the SummitADF_Examples Workspace. For information about how to obtain and install the Summit ADF standalone sample applications, see Setting Up the Summit Sample Applications for Oracle ADF.

How To Create a Framework Extension Class

When you need to add custom code to extend the base functionality of the ADF Business Components framework, you can enable a custom Java class for any of the key types of business components you create. You enable the generation of custom classes for a component on the Java page of its respective overview editor in JDeveloper. When you enable this option, JDeveloper creates a Java source file for a custom class related to the component whose name follows a configurable naming standard. This class, whose name is recorded in the component's XML document, provides a place where you can write the custom Java code required by that component.

To create a framework extension class:

1. Identify a project to contain the framework extension class.

   You can create it in the same project as your business service components if you believe it will only be used by components in that project. Alternatively, if you believe you might like to reuse the framework extension class across multiple Fusion web applications, create a separate model project to contain the framework extension classes.

2. Ensure that the **BC4J Runtime** library is in the project's libraries list.

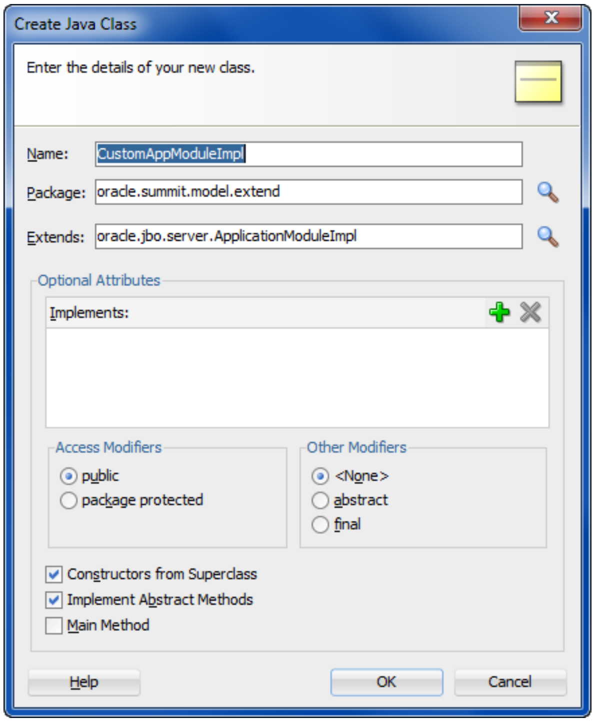
   Use the **Libraries and Classpath** page of the **Project Properties** dialog to verify this and to add the library if missing.

3. In the Applications window, right-click the project in which you want to create the extension class and choose **New** and then **Java Class**.

4. In the Create Java Class dialog, specify the appropriate framework base class from the `oracle.jbo.server` package in the **Extends** field.

   Figure 17-1 illustrates what it would look like to create a custom framework extension class named `CustomAppModuleImpl` in the `com.yourcompany.fwkext` package to customize the functionality of the base **application module** component. To quickly find the base class you're looking for, use the **Browse** button next to the **Extends** field that launches the JDeveloper **Class Browser**. Using its **Search** tab, you can type in part of the class name (including using `*` as a wildcard) to quickly subset the list of classes to find the one you're looking for.

Figure 17-1 Creating a Framework Extension Class for an Application Module



Enter the details of your new class.

Name: CustomAppModuleImpl

Package: oracle.summit.model.extend

Extends: oracle.jbo.server.ApplicationModuleImpl

Optional Attributes

Implements:

Access Modifiers

● public
○ package protected

Other Modifiers

● <None>
○ abstract
○ final

☑ Constructors from Superclass
☑ Implement Abstract Methods
□ Main Method

When you click **OK**, JDeveloper creates the custom framework extension class for you in the directory of the project's source path corresponding to the package name you've chosen.

> **Note:**
>
> Some ADF Business Component classes exist in both a server-side and a remote-client version. For example, if you use the JDeveloper **Class Browser** and type `ApplicationModuleImpl` into the **Match Class Name** field on the **Search** tab, the list will show two `ApplicationModuleImpl` classes: one in the `oracle.jbo.server` package and the other in the `oracle.jbo.client.remote` package. When creating framework extension classes, use the base ADF Business Components classes in the `oracle.jbo.server` package.

What Happens When You Create a Framework Extension Class

After creating a new framework extension class, it will not automatically be used by your application. You must decide which components in your project should make use of it. The following sections describe the available approaches for basing your business components on your own framework extension classes.

What You May Need to Know About Customizing Framework Extension Bases Classes

To make your framework extension layer classes easier to package as a reusable library, create them in a separate project from the projects that use them.

A common set of customized framework base classes in a package name of your own choosing like *com.yourcompany.fwkext*, each importing the oracle.jbo.server.* package, would consist of the following classes:

- public class CustomEntityImpl extends EntityImpl
- public class CustomEntityDefImpl extends EntityDefImpl
- public class CustomViewObjectImpl extends ViewObjectImpl
- public class CustomViewRowImpl extends ViewRowImpl
- public class CustomApplicationModuleImpl extends ApplicationModuleImpl
- public class CustomDBTransactionImpl extends DBTransactionImpl2
- public class CustomDatabaseTransactionFactoryImpl extends DatabaseTransactionFactory

For details about using the custom DBTransactionImpl2 and DatabaseTransactionFactory classes, see Configuring an Application Module to Use a Custom Database Transaction Class.

For completeness, you may also want to create customized framework classes for the following classes as well, note however that overriding anything in these classes would be a fairly rare requirement.

- public class CustomViewDefImpl extends ViewDefImpl
- public class CustomEntityCache extends EntityCache
- public class CustomApplicationModuleDefImpl extends ApplicationModuleDefImpl

How to Base a Business Component on a Framework Extension Class

You can set the base classes for any business component using the **Java** page of any ADF Business Components wizard or editor.

Before you begin:

- Create the framework extension class, as described in How To Create a Framework Extension Class.
- If you created your framework extension classes in a separate project, visit the **Dependencies** page of the Project Properties dialog for the project containing your business components and select **Build Output** to add the framework extension project as a project dependency.
- If you have packaged your framework extension classes in a Java archive (JAR) file, create a named library definition to reference its JAR file and also list that library in the library list of the project containing your business components. To create a library if missing, use the Manage Libraries dialog available from the

Tools > Manage Libraries main menu item. To verify or adjust the project's library list, use the **Libraries** page of the Project Properties dialog.

After you ensure the framework classes are available to reference, you can create the business component. Every ADF Business Components wizard and editor displays the same **Class Extends** button on the **Java** page so you can use the technique to choose your desired framework extension base class(es) both for new components or existing ones.

There is no fixed limit on how many levels of framework extension classes you create. For example, after creating a company-level CustomAppModuleImpl to use for all application modules in all Fusion web applications that your company creates, some later project team might encounter the need to further customize that framework extension class. That team could create a SomeProjectCustomAppModuleImpl class that extends the CustomAppModuleImpl and then include the project-specific custom application module code in there as shown in the following example.

```
public class SomeProjectCustomAppModuleImpl
       extends CustomAppModuleImpl {
  /*
   * Custom application module code specific to the
   * "SomeProject" project goes here.
   */
}
```

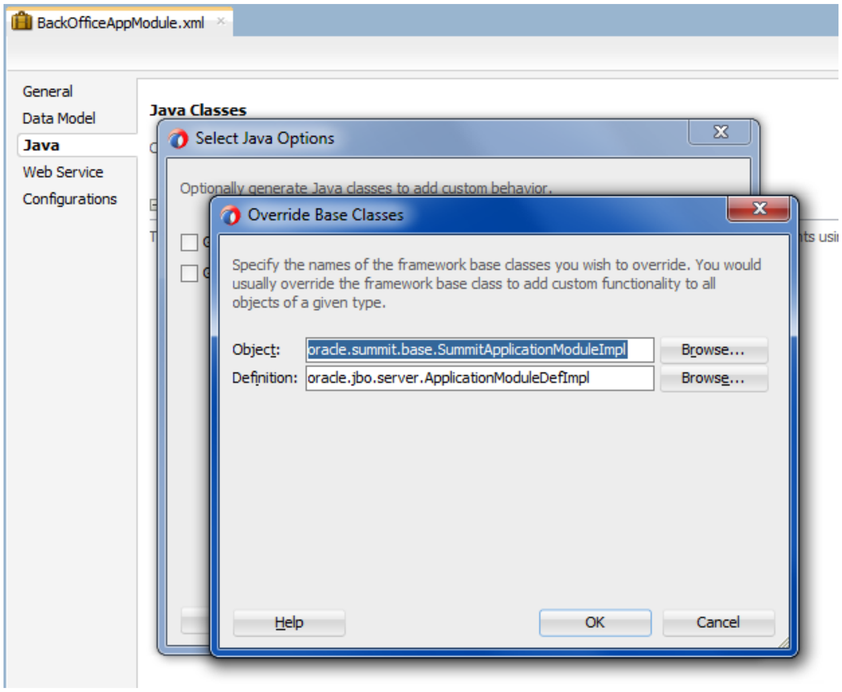
Then, any application modules created as part of the implementation of this specific project can use the SomeProjectCustomAppModuleImpl as their base class instead of the CustomAppModuleImpl.

To create a business component based on a framework extension class:

1. In the Applications window, double-click the desired component.
2. In the overview editor, click the **Java** navigation tab and then click the **Edit Java options** button.
3. In the Select Java Options dialog, click **Classes Extends**.
4. In the Override Base Classes dialog, enter the fully qualified name of the framework base classes you wish to override. You can also use the **Browse** button to use the JDeveloper Class Browser to find the classes quickly.

   When you use the Class Browser to select a custom base class for the component, the list of available classes is automatically filtered to show only classes that are appropriate. For example, when clicking **Browse** in Figure 17-2 to select an application module **Object** base class, the list will only show classes available in the current project's library list which extend the oracle.jbo.server.ApplicationModule class either directly or indirectly. If you don't see the class you're looking for, either you extended the incorrect base class or you have chosen the wrong component class name to override.

Figure 17-2 Specifying a Custom Base Class for a New Application Module



Optionally generate Java classes to add custom behavior.

Specify the names of the framework base classes you wish to override. You would usually override the framework base class to add custom functionality to all objects of a given type.

Object: oracle.summit.base.SummitApplicationModuleImpl

Definition: oracle.jbo.server.ApplicationModuleDefImpl

How to Define Framework Extension Classes for All New Components

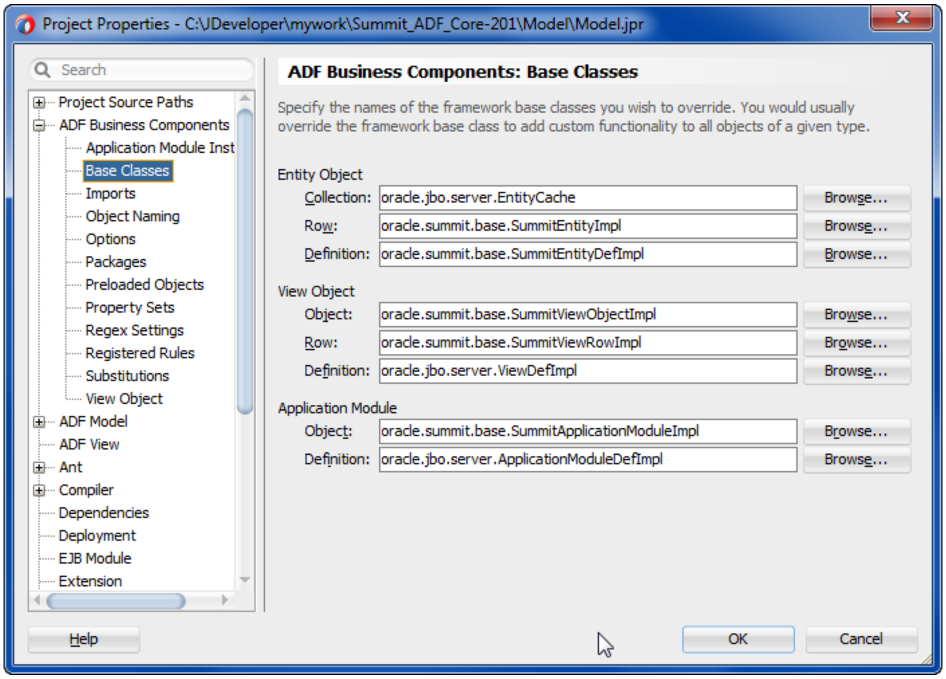
If you decide to use a specific set of framework extension classes as a standard for a given project, you can use the Project Properties dialog to define your preferred base classes for each component type. Setting these preferences for base classes does not affect any existing components in the project, but the component wizards will use the preferences for any new components created.

To define project-level preferences for framework extension classes:

1. In the Applications window, right-click the project that will contain the extension classes and choose **Project Properties**.
2. In the Project Properties dialog, expand **ADF Business Components > Base Classes** and enter the fully qualified name of the base class in the **Application Module Object** class name field.

   For example, to indicate that any new application modules created in the project should use the `CustomAppModuleImpl` class by default, enter the fully qualified name of that class in the ***componentName*** **Object** class name field as shown in Figure 17-3.

Figure 17-3 Setting Project-Level Preferences for Business Component Base Classes



Project Properties - C:\JDeveloper\mywork\Summit_ADF_Core-201\Model\Model.jpr

ADF Business Components: Base Classes

Specify the names of the framework base classes you wish to override. You would usually override the framework base class to add custom functionality to all objects of a given type.

Entity Object
Collection: oracle.jbo.server.EntityCache
Row: oracle.summit.base.SummitEntityImpl
Definition: oracle.summit.base.SummitEntityDefImpl

View Object
Object: oracle.summit.base.SummitViewObjectImpl
Row: oracle.summit.base.SummitViewRowImpl
Definition: oracle.jbo.server.ViewDefImpl

Application Module
Object: oracle.summit.base.SummitApplicationModuleImpl
Definition: oracle.jbo.server.ApplicationModuleDefImpl

How to Define Framework Extension Classes for All New Projects

When you want to apply the same base class preferences to each new project that you create in JDeveloper, you can define the preferences at a global level using the Preferences dialog. Base classes that you specify at the global level will not alter your existing projects containing business components.

To define global preferences for framework extension classes:

1. In the main menu, choose **Tools** and then **Preferences**.
2. In the Preferences dialog, expand **ADF Business Components > Base Classes** in the tree.
3. On the Business Components page, enter the fully qualified name of that class in ***componentName*** **Object** class name field.

   The page displays the same options for specifying the preferred base classes for each component type as shown in How to Define Framework Extension Classes for All New Components.

What Happens When You Base a Component on a Framework Extension Class

When a business component you create extends a custom ADF Business Components framework extension class, JDeveloper updates its XML document to reflect the custom class name you've chosen.

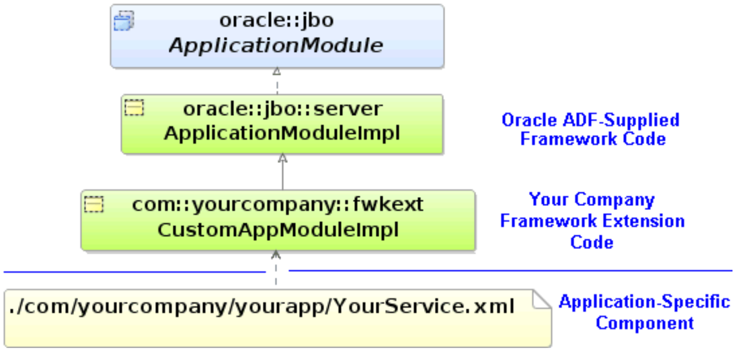
XML-Only Components

For example, assume you've created the YourService application module in the com.yourcompany.yourapp package, with a custom application module base class of CustomAppModuleImpl. If you have opted to leave the component as an XML-only component with no custom Java file, its XML document (YourService.xml) will look similear to the one shown in the following example. The value of the ComponentClass attribute of the AppModule tag is read at runtime to identify the Java class to use to represent the component.

```
<AppModule
   Name="YourService"
   ComponentClass="com.yourcompany.fwkext.CustomAppModuleImpl" >
  <!-- etc. -->
</AppModule>
```

Figure 17-4 illustrates how the XML-only YourService application module relates to your custom extension class. At runtime, it uses the CustomAppModuleImpl class which inherits its base behavior from the ApplicationModuleImpl class.

Figure 17-4 XML-Only Component Reference an Extended Framework Base Class


oracle::jbo
ApplicationModule
oracle::jbo::server
ApplicationModuleImpl
com::yourcompany::fwkext
CustomAppModuleImpl
./com/yourcompany/yourapp/YourService.xml
Oracle ADF-Supplied Framework Code
Your Company Framework Extension Code
Application-Specific Component

Components with Custom Java Classes

If your component requires a custom Java class, as you've seen in previous chapters you open the **Java** page of the component editor and check the appropriate checkbox to enable it. For example, when you enable a custom application module class for the YourServer application module, JDeveloper creates the appropriate YourServiceImpl.java class. As shown in the following example, it also updates the component's XML document to reflect the name of the custom component class.

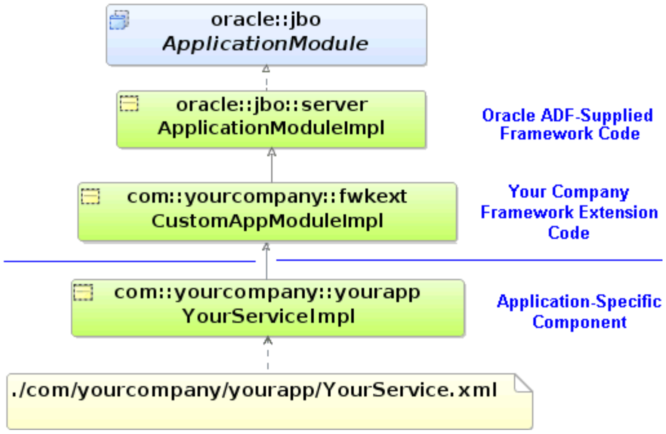
```
<AppModule
   Name="YourService"
   ComponentClass="com.yourcompany.yourapp.YourServiceImpl" >
  <!-- etc. -->
</AppModule>
```

JDeveloper also updates the component's custom Java class to modify its extends clause to reflect the new custom framework base class, as shown in the following example.

```
package com.yourcompany.yourapp;
import com.yourcompany.fwkext.CustomAppModuleImpl;
// ---------------------------------------------------------------------
// ---    File generated by Oracle ADF Business Components Design Time.
// ---    Custom code may be added to this class.
// ---    Warning: Do not modify method signatures of generated methods.
// ---------------------------------------------------------------------
public class YourServiceImpl extends CustomAppModuleImpl {
  /**This is the default constructor (do not remove)   */
  public YourServiceImpl() {}
  // etc.
}
```

Figure 17-5 illustrates how the `YourService` application module with its custom `YourServiceImpl` class is related to your framework extension class. At runtime, it uses the `YourServiceImpl` class which inherits its base behavior from the `CustomAppModuleImpl` framework extension class which, in turn, extends the base `ApplicationModuleImpl` class.

Figure 17-5   Component with Custom Java Extending Customized Framework Base Class


oracle::jbo
ApplicationModule
oracle::jbo::server
ApplicationModuleImpl
Oracle ADF-Supplied Framework Code
com::yourcompany::fwkext
CustomAppModuleImpl
Your Company Framework Extension Code
com::yourcompany::yourapp
YourServiceImpl
Application-Specific Component
./com/yourcompany/yourapp/YourService.xml

What You May Need to Know About Updating the Extends Clause in Custom Component Java Files

If you have a business component with a custom Java class and later decide to base the component on an ADF Business Components framework extension class, use the **Class Extends** button in the Select Java Options dialog to change the component's base class. You can open the dialog from the **Java** page of the component's overview editor. Doing this updates the component's XML document to reflect the new base class, and *also* modifies the `extends` clause in the component's custom Java class.

> **Note:**
>
> If you manually update the `extends` clause without using the component editor, the component's XML document will not reflect the new inheritance and the next time you open the editor, your manually modified `extends` clause will be overwritten with what the component editor believes is the correct component base class.

How to Package Your Framework Extension Layer in a JAR File

Use the **Create Deployment Profile: JAR File** dialog to create a JAR file containing the classes in your framework extension layer. This is available in the **New Gallery** in the **General > Deployment Profiles** category.

Give the deployment profile a name like `FrameworkExtensions` and click **OK**. By default the JAR file will include all class files in the project. Since this is exactly what you want, when the **JAR Deployment Profile Properties** dialog appears, you can just click **OK** to finish.

> **Note:**
>
> Do not use the **ADF Library JAR** archive type to package your framework extension layer. You create the ADF Library JAR file when you want to package reusable components to share in the JDeveloper Resource Catalog. For details about working with business components and the ADF Library JAR archive type, see Packaging a Reusable ADF Component into an ADF Library.

Finally, to create the JAR file, right-click the project node in the Applications window and choose **Deploy** - ***YourProfileName*** - **to JAR File** on the context menu. A **Deployment** tab appears in the JDeveloper **Log window** that should display feedback like:

```
----  Deployment started.  ----    Feb 14, 2013 1:42:39 PM
Running dependency analysis...
Wrote JAR file to ...\FrameworkExtensions\deploy\FrameworkExtensions.jar
Elapsed time for deployment:  2 seconds
----  Deployment finished.  ----    Reb 14, 2013 1:42:41 PM
```

How to Create a Library Definition for Your Framework Extension JAR File

JDeveloper uses named libraries as a convenient way to organize the one or more JAR files that comprise reusable component libraries.

To define a library for your framework extensions JAR file:

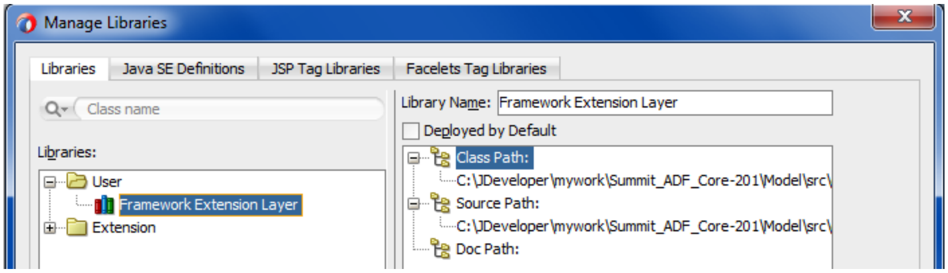
1. In the main menu, choose **Tools** and then **Manage Libraries**.

2. In the **Manage Libraries** dialog, select the **Libraries** tab and then select **User** and click **New**.
3. In the **Create Library** dialog that appears, name the library "Framework Extension Layer" and select the **Class Path** node and click **Add Entry**.
4. Use the **Select Path Entry** dialog to select the JAR file that contains the class files for the framework extension components, then click **Select**.
5. Select the **Source Path** node and click **Add Entry.**
6. Use the **Select Path Entry** dialog that appears to select the directory where the source files for the framework extension classes reside, then click **Select**.

   For example, select `..\FrameworkExtensions\src` for the JAR file `FrameworkExtensions.jar`.
7. Click **OK**.

When finished, you will see your new "Framework Extension Layer" user-defined library, as shown in Figure 17-6. You can then add this library to the library list of any project where you will be building business services, and your custom framework extension classes will be available to reference as the preferred component base classes.

Figure 17-6 New User-Defined Library for Your Framework Extensions Layer



Library Name: Framework Extension Layer

□ Deployed by Default

Class Path:
C:\JDeveloper\mywork\Summit_ADF_Core-201\Model\src\

Source Path:
C:\JDeveloper\mywork\Summit_ADF_Core-201\Model\src\

Doc Path:

Customizing Framework Behavior with Extension Classes

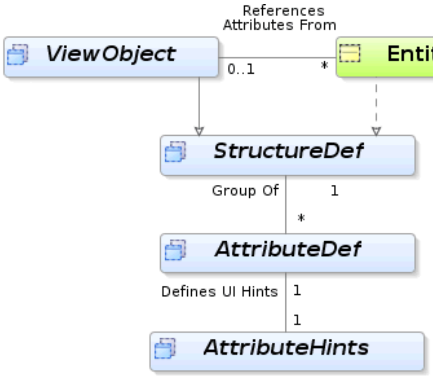
ADF Business Components framework extension code is utilized by all components of a specific type. Use ADF Business Components APIs to access component meta-data at runtime within the code, to write generic functionality.

One of the common tasks you'll perform in your framework extension classes is implementing custom application functionality. Since framework extension code is written to be used by all components of a specific type, the code you write in these classes often needs to work with component attributes in a generic way. To address this need, ADF Business Components provides API's that allow you to access component metadata at runtime. It also provides the ability to associate custom metadata properties with any component or attribute. You can write your generic framework extension code to leverage runtime metadata and custom properties to build generic functionality, which if necessary, only is used in the presence of certain custom properties.

How to Access Runtime Metadata For View Objects and Entity Objects

Figure 17-7 illustrates the three primary interfaces ADF Business Components provides for accessing runtime metadata about **view objects** and entity objects. The `ViewObject` interface extends the `StructureDef` interface. The class representing the entity definition (`EntityDefImpl`) also implements this interface. As its name implies, the `StructureDef` defines the structure and the component and provides access to a collection of `AttributeDef` objects that offer runtime metadata about each attribute in the view object row or entity row. Using an `AttributeDef`, you can access its companion `AttributeHints` object to reference hints like the display label, format mask, tooltip, etc.

Figure 17-7 Runtime Metadata Available for View Objects and Entity Objects



How to Implement Generic Functionality Using Runtime Metadata

In ViewObject Interface Methods for Working with the View Object's Default RowSet you learned that for read-only view objects the `findByKey()` method and the `setCurrentRowWithKey` builtin operation only work if you override the `create()` method on the view object to call `setManageRowsByKey(true)`. This can be a tedious detail to remember if you create a lot of read-only view objects, so it is a great candidate for automating in a framework extension class for view objects.

Assume a `FrameworkExtensions` project contains a `SummitViewObjectImpl` class that is the base class for all view objects in the application. This framework extension class for view objects extends the base `ViewObjectImpl` class and overrides the `create()` method as shown in the following example to automate this task. After calling the `super.create()` to perform the default framework functionality when a view object instance is created at runtime, the code tests whether the view object is a read-only view object with at least one attribute marked as a key attribute. If this is the case, it invokes `setManageRowsByKey(true)`.

The `isReadOnlyNonEntitySQLViewWithAtLeastOneKeyAttribute()` helper method determines whether the view object is read-only by testing the combination of the following conditions:

- `isFullSql()` is `true`

This method returns true if the view object's SQL query is completely specified by the developer, as opposed to having the select list derived automatically based on the participating entity usages.

- `getEntityDefs()` is `null`

  This method returns an array of `EntityDefImpl` objects representing the view object's entity usages. If it returns `null`, then the view object has no entity usages.

It goes on to determine whether the view object has any key attributes by looping over the `AttributeDef` array returned by the `getAttributeDefs()` method. If the `isPrimaryKey()` method returns true for any attribute definition in the list, then you know the view object has a key.

```
public class SummitViewObjectImpl extends ViewObjectImpl {
  protected void create() {
    super.create();
    if (isReadOnlyNonEntitySQLViewWithAtLeastOneKeyAttribute()) {
      setManageRowsByKey(true);
    }
  }
  boolean isReadOnlyNonEntitySQLViewWithAtLeastOneKeyAttribute() {
    if (getViewDef().isFullSql() && getEntityDefs() == null) {
      for (AttributeDef attrDef : getAttributeDefs()) {
        if (attrDef.isPrimaryKey()) {
          return true;
        }
      }
    }
    return false;
  }
  // etc.
}
```

How to Implement Generic Functionality Driven by Custom Properties

When you create application modules, view objects, and entity objects you can select the **General** navigation tab in the overview editor for these business components and expand the **Custom Properties** section to define custom metadata properties for any component. These are name/value pairs that you can use to communicate additional declarative information about the component to the generic code that you write in framework extension classes. You can use the `getProperty()` method in your code to conditionalize generic functionality based on the presence of, or the specific value of, one of these custom metadata properties.

For example, the `SummitViewObjectImpl` framework extension class overrides the view object's `insertRow()` method as shown in the following example to conditionally force a row to be inserted and to appear as the last row in the row set. If any view object extending this framework extension class defines a custom metadata property named `InsertNewRowsAtEnd`, then this generic code executes to insert new rows at the end. If a view object does not define this property, it will have the default `insertRow()` behavior.

```
public class SummitViewObjectImpl extends ViewObjectImpl {
  private static final String INSERT_NEW_ROWS_AT_END = "InsertNewRowsAtEnd";
  public void insertRow(Row row) {
    super.insertRow(row);
    if (getProperty(INSERT_NEW_ROWS_AT_END) != null) {
      row.removeAndRetain();
```

```
        last();
        next();
        getDefaultRowSet().insertRow(row);
      }
    }
    // etc.
  }
```

In addition to defining component-level custom properties, you can also define properties on view object attributes, entity object attributes, and domains. At runtime, you access them using the `getProperty()` method on the `AttributeDef` interface for a given attribute.

What You May Need to Know About the Kinds of Attributes

In addition to providing information about an attribute's name, Java type, SQL type, and many other useful pieces of information, the `AttributeDef` interface contains the `getAttributeKind()` method that you can use to determine the kind of attribute it represents. This method returns a `byte` value corresponding to one of the public constants in the `AttributeDef` interface listed in Table 17-1.

Table 17-1 Entity Object and View Object Attribute Kinds

Public `AttributeDef` Constant	Attribute Kind Description
`ATTR_PERSISTENT`	Persistent attribute
`ATTR_TRANSIENT`	Transient attribute
`ATTR_ENTITY_DERIVED`	View object attribute mapped to an entity-level transient attribute
`ATTR_SQL_DERIVED`	SQL-Calculated attribute
`ATTR_DYNAMIC`	Dynamic attribute
`ATTR_ASSOCIATED_ROWITERATOR`	Accessor attribute returning a `RowSet` of set of zero or more `Rows`
`ATTR_ASSOCIATED_ROW`	Accessor attribute returning a single `Row`

What You May Need to Know About Custom Properties

You may find it handy to programmatically set custom property values at runtime. While the `setProperty()` API to perform this function is by design not available to clients on the `ViewObject`, `ApplicationModule`, or `AttributeDef` interfaces in the `oracle.jbo` package, code that you write *inside* custom Java classes of your business components can use it.

Creating Generic Extension Interfaces

ADF Business Components allows you to create custom interfaces that all components can implement. It helps for example, to expose methods from an application module.

In addition to creating framework extension classes, you can create custom interfaces that all of your components can implement by default. The client interface is very useful for exposing methods from your application module that might be invoked by UI clients, for example. This section considers an example for an application module, however, the same functionality is possible for a custom extended view object and view row interface as well. For information about client interfaces, see Publishing Custom Service Methods to UI Clients and Working Programmatically with an Application Module's Client Interface.

Assume that you have a `CustomApplicationModuleImpl` class that extends `ApplicationModuleImpl` and that you want to expose two custom methods like this:

```
public void doFeatureOne(String arg);
public int anotherFeature(String arg);
```

Perform the following steps to create a custom extension interface `CustomApplicationModule` and have your `CustomApplicationModuleImpl` class implement it.

1. Create a custom interface that contains the methods you would like to expose globally on your application module components. For this scenario, that interface would look like this:

   ```
   package devguide.advanced.customintf.fwkext;
   /**
    * NOTE: This does not extend the
    * ====  oracle.jbo.ApplicationModule interface.
    */
   public interface CustomApplicationModule  {
     public void doFeatureOne(String arg);
     public int anotherFeature(String arg);
   }
   ```

   Notice that the interface does *not* extend the `oracle.jbo.ApplicationModule` interface.

2. Modify your `CustomApplicationModuleImpl` application module framework extension class to implement this new `CustomApplicationModule` interface.

   ```
   package devguide.advanced.customintf.fwkext;
   import oracle.jbo.server.ApplicationModuleImpl;
   public class CustomApplicationModuleImpl
          extends ApplicationModuleImpl
          implements CustomApplicationModule {
     public void doFeatureOne(String arg) {
       System.out.println(arg);
     }
     public int anotherFeature(String arg) {
       return arg == null ? 0 : arg.length();
     }
   }
   ```

3. Rebuild your project.

   The ADF Business Components overview editors will only "see" your interfaces after they have been successfully compiled.

After you have implemented your `CustomApplicationModuleImpl` class, you can create a new application module which exposes the global extension interface and is based on your custom framework extension class. For this purpose you use the overview editor for application modules.

To create a custom application module interface:

1. In the Applications window, double-click the application module for which you want to create the custom interface.

   For example, you might create a new `ProductModule` application module which exposes the global extension interface `CustomApplicationModule` and is based on the `CustomApplicationModuleImpl` framework extension class.

2. In the overview editor, select the **Java** navigation tab and then click the **Edit Java options** icon.

   The Java Classes page should show an existing Java class for the application module identified as **Application Module Class**.

   By default, JDeveloper generates the Java class for application modules you create. However, if you disabled this feature, click the **Edit Java options** button in the Java Classes section and select **Generate Application Module Class**. Click **OK** to add a Java class to the project from which you will create the custom interface.

3. In the Select Java Options dialog, click **Class Extends**.

4. In the Override Base Classes dialog, specify the name of the framework base class you want to override and click **OK**.

   For example, you might select `CustomApplicationModuleImpl` as the base class for the application module.

5. In the Java Classes page of the overview editor, expand the **Client Interface** section and click the **Edit application module client interface** button.

6. In the Edit Client Interface dialog, click the **Interfaces** button.

7. In the Select Interfaces to Extend dialog, select the desired custom application module interface from the available list and click **OK**.

   For example, you might shuttle the `CustomApplicationModule` interface to the **Selected** list to be one of the custom interfaces that clients can use with your component.

8. In the Edit Client Interfaces dialog, ensure that at least one method appears in the **Selected** list.

   > **Note:**
   >
   > You need to select at least one method in the **Selected** list in the Edit Client Interfaces dialog, even if it means redundantly selecting one of the methods on the global extension interface. Any method will do in order to get JDeveloper to generate the custom interface.

9. Click **OK**.

   The Java Classes page displays the new custom interface for the application module identified as **Application Module Client Interface**.

When you dismiss the Edit Client Interfaces dialog and return to the application module overview editor, JDeveloper generates the application module custom interface. For example, the custom interface `ProductModule` automatically extends *both* the base `ApplicationModule` interface and your `CustomApplicationModule` extension interface like this:

```
package devguide.advanced.customintf.common;
import devguide.advanced.customintf.fwkext.CustomApplicationModule;

import oracle.jbo.ApplicationModule;
// ---------------------------------------------------------------------
// ---    File generated by ADF Business Components Design Time.
// ---------------------------------------------------------------------
public interface ProductModule
       extends CustomApplicationModule, ApplicationModule {
  void doSomethingProductRelated();
}
```

Once you've done this, then client code can cast your `ProductModule` application module to a `CustomApplicationModule` interface and invoke the generic extension methods it contains in a strongly typed way.

> **Note:**
>
> The basic steps are the same for exposing methods on a `ViewObjectImpl` framework extension class, as well as for a `ViewRowImpl` extension class.

Invoking Stored Procedures and Functions

You can write custom code in the custom Java classes of ADF Business Components to invoke database stored procedures and functions.

You can write code in the custom Java classes for your business components to invoke database stored procedures and functions. Here you'll consider some simple examples based on procedures and functions in a PL/SQL package; however, using the same techniques, you also can invoke procedures and functions that are not part of a package.

Consider the PL/SQL package shown in the following example.

```
create or replace package invokestoredprocpkg as
  procedure proc_with_no_args;
  procedure proc_with_three_args(n number, d date, v varchar2);
  function  func_with_three_args(n number, d date, v varchar2) return varchar2;
  procedure proc_with_out_args(n number, d out date, v in out varchar2);
end invokestoredprocpkg;
```

The following sections explain how to invoke each of the example procedures and functions in this package.

> **Note:**
>
> The example in this section refers to the `oracle.summit.model.invokingstoredprocedure` package in the `SummitADF_Examples` application workspace.

How to Invoke Stored Procedures with No Arguments

If you need to invoke a stored procedure that takes no arguments, you can use the `executeCommand()` method on the `DBTransaction` interface (in the `oracle.jbo.server` package as shown in the following example.

```
// In InvokingStoredProcAppModuleImpl.java
public void callProcWithNoArgs() {
  getDBTransaction().executeCommand(
    "begin invokestoredprocpkg.proc_with_no_args; end;");
}
```

How to Invoke Stored Procedure with Only IN Arguments

Invoking stored procedures that accept only `IN`-mode arguments — which is the default PL/SQL parameter mode if not specified — requires using a JDBC `PreparedStatement` object. The `DBTransaction` interface provides a `createPreparedStatement()` method to create this object for you in the context of the current database connection. You could use a helper method like the one shown in the following example to simplify the job of invoking a stored procedure of this kind using a `PreparedStatement`. Importantly, by using a helper method, you can encapsulate the code that closes the JDBC `PreparedStatement` after executing it. The code performs the following basic tasks:

1. Creates a JDBC `PreparedStatement` for the statement passed in, wrapping it in a PL/SQL `begin`...`end` block.
2. Loops over values for the bind variables passed in, if any.
3. Sets the value of each bind variable in the statement.

   Notice that since JDBC bind variable API's use one-based numbering, the code adds one to the zero-based for loop index variable to account for this.
4. Executes the statement.
5. Closes the statement.

```
protected void callStoredProcedure(String stmt, Object[] bindVars) {
  PreparedStatement st = null;
  try {
    // 1. Create a JDBC PreparedStatement for
    st = getDBTransaction().createPreparedStatement("begin "+stmt+";end;",0);
    if (bindVars != null) {
      // 2. Loop over values for the bind variables passed in, if any
      for (int z = 0; z < bindVars.length; z++) {
        // 3. Set the value of each bind variable in the statement
        st.setObject(z + 1, bindVars[z]);
      }
    }
    // 4. Execute the statement
    st.executeUpdate();
  }
  catch (SQLException e) {
    throw new JboException(e);
  }
  finally {
    if (st != null) {
      try {
        // 5. Close the statement
        st.close();
      }
      catch (SQLException e) {}
    }
  }
}
```

With a helper method like this in place, calling the `proc_with_three_args` procedure shown in the previous example would look like this:

```
// In StoredProcTestModuleImpl.java
public void callProcWithThreeArgs(Number n, Date d, String v) {
  callStoredProcedure("callStoredProcedure.proc_with_three_args(?,?,?)",
                      new Object[]{n,d,v});
}
```

Notice the question marks used as JDBC bind variable placeholders for the arguments passed to the function. JDBC also supports using named bind variables, but using these simpler positional bind variables is also fine since the helper method is just setting the bind variable values positionally.

How to Invoke Stored Function with Only IN Arguments

Invoking stored functions that accept only `IN`-mode arguments requires using a JDBC `CallableStatement` object in order to access the value of the function result after executing the statement. The `DBTransaction` interface provides a `createCallableStatement()` method to create this object for you in the context of the current database connection. You could use a helper method like the one shown in the following example to simplify the job of invoking a stored function of this kind using a `CallableStatement`. The helper method encapsulates both the creation and clean up of the JDBC statement being used.

The code performs the following basic tasks:

1. Creates a JDBC `CallableStatement` for the statement passed in, wrapping it in a PL/SQL `begin`...`end` block.
2. Registers the first bind variable for the function return value.
3. Loops over values for the bind variables passed in, if any.
4. Sets the value of each user-supplied bind variable in the statement.

   Notice that since JDBC bind variable API's use one-based numbering, and since the function return value is already the first bind variable in the statement, the code adds *two* to the zero-based for loop index variable to account for these.
5. Executes the statement.
6. Returns the value of the first bind variable.
7. Closes the statement.

```
// Some constants
public static int NUMBER = Types.NUMERIC;
public static int DATE = Types.DATE;
public static int VARCHAR2 = Types.VARCHAR;

protected Object callStoredFunction(int sqlReturnType, String stmt,
                                    Object[] bindVars) {
```

```
  CallableStatement st = null;
  try {
    // 1. Create a JDBC CallabledStatement
    st = getDBTransaction().createCallableStatement(
           "begin ? := "+stmt+";end;",0);
    // 2. Register the first bind variable for the return value
    st.registerOutParameter(1, sqlReturnType);
    if (bindVars != null) {
      // 3. Loop over values for the bind variables passed in, if any
      for (int z = 0; z < bindVars.length; z++) {
        // 4. Set the value of user-supplied bind vars in the stmt
        st.setObject(z + 2, bindVars[z]);
      }
    }
    // 5. Set the value of user-supplied bind vars in the stmt
    st.executeUpdate();
    // 6. Return the value of the first bind variable
    return st.getObject(1);
  }
  catch (SQLException e) {
    throw new JboException(e);
  }
  finally {
    if (st != null) {
      try {
        // 7. Close the statement
        st.close();
      }
      catch (SQLException e) {}
    }
  }
}
```

With a helper method like this in place, calling the `func_with_three_args` procedure shown in the previous example would look like this:

```
// In InvokingStoredProcAppModuleImpl.java
public String callFuncWithThreeArgs(Number n, Date d, String v) {
  return (String)callStoredFunction(VARCHAR2,
                           "invokestoredprocpkg.func_with_three_args(?,?,?)",
                           new Object[]{n,d,v});
}
```

Notice the question marks are used as JDBC bind variable placeholders for the arguments passed to the function. JDBC also supports using named bind variables, but using these simpler positional bind variables is also fine since the helper method is just setting the bind variable values positionally.

How to Call Other Types of Stored Procedures

Calling a stored procedure or function like `invokestoredprocpkg.proc_with_out_args` that includes arguments of `OUT` or `IN OUT` mode requires using a `CallableStatement` as in the previous section, but is a little more challenging to generalize into a helper method. The following example illustrates the JDBC code necessary to invoke the `invokestoredprocpkg.proc_with_out_args` procedure.

The code performs the following basic tasks:

1. Defines a PL/SQL block for the statement to invoke.

2. Creates the `CallableStatement` for the PL/SQL block.
3. Registers the positions and types of the `OUT` parameters.
4. Sets the bind values of the `IN` parameters.
5. Executes the statement.
6. Creates a JavaBean to hold the multiple return values

   The `DateAndStringBean` class contains bean properties named `dateVal` and `stringVal`.
7. Sets the value of its `dateVal` property using the first `OUT` param.
8. Sets value of its `stringVal` property using second `OUT` param.
9. Returns the result.
10. Closes the JDBC `CallableStatement`.

```
public DateAndStringBean callProcWithOutArgs(Number n, String v) {
  CallableStatement st = null;
  try  {
    // 1. Define the PL/SQL block for the statement to invoke
    String stmt = "begin invokestoredprocpkg.proc_with_out_args(?,?,?); end;";
    // 2. Create the CallableStatement for the PL/SQL block
    st = getDBTransaction().createCallableStatement(stmt,0);
    // 3. Register the positions and types of the OUT parameters
    st.registerOutParameter(2,Types.DATE);
    st.registerOutParameter(3,Types.VARCHAR);
    // 4. Set the bind values of the IN parameters
    st.setObject(1,n);
    st.setObject(3,v);
    // 5. Execute the statement
    st.executeUpdate();
    // 6. Create a bean to hold the multiple return values
    DateAndStringBean result = new DateAndStringBean();
    // 7. Set value of dateValue property using first OUT param
    result.setDateVal(new Date(st.getDate(2)));
    // 8. Set value of stringValue property using 2nd OUT param
    result.setStringVal(st.getString(3));
    // 9. Return the result
    return result;
  } catch (SQLException e)  {
    throw new JboException(e);
  } finally  {
    if (st != null) {
      try {
        // 10. Close the JDBC CallableStatement
        st.close();
      }
      catch (SQLException e) {}
    }
  }
}
```

The `DateAndString` bean used in the previous example is a simple JavaBean with two bean properties like this:

```
package oracle.summit.model.invokingstoredprocedure;

import java.io.Serializable;
import oracle.jbo.domain.Date;

public class DateAndStringBean implements Serializable {
  Date dateVal;
  String stringVal;
  public void setDateVal(Date dateVal) {this.dateVal=dateVal;}
  public Date getDateVal() {return dateVal;}
  public void setStringVal(String stringVal) {this.stringVal=stringVal;}
  public String getStringVal() {return stringVal;}
}
```

> **Note:**
>
> In order to allow the custom method to be a legal candidate for inclusion in an application module's custom service interface (if desired), the bean needs to implement the `java.io.Serializable`. interface. Since this is a "marker" interface, this involves simply adding the `implements Serializable` keywords without needing to code the implementation of any interface methods.

Accessing the Current Database Transaction

When trying to integrate third-party code with ADF Business Components, use a helper method to access the JDBC Connection object.

Since ADF Business Components abstracts all of the lower-level database programming details for you, you typically won't need *direct* access to the JDBC `Connection` object. There is no guarantee at runtime that your application will use the exact same **application module instance** or JDBC `Connection` instance across different web page requests. Since inadvertently holding a reference to the JDBC Connection object in this type of pooled services environment can cause unpredictable behavior at runtime, by design, ADF Business Components has no direct API to obtain the JDBC `Connection`. This is an intentional attempt to discourage its direct use and inadvertent abuse.

However, on occasion it may come in handy when you're trying to integrate third-party code with ADF Business Components, so you can use a helper method like the one shown in the following example to access the connection.

```
/**
 * Put this method in your XXXImpl.java class where you need
 * to access the current JDBC connection
 */
private Connection getCurrentConnection() throws SQLException {
 /* Note that we never execute this statement, so no commit really happens */
 PreparedStatement st = getDBTransaction().createPreparedStatement("commit",1);
 Connection conn = st.getConnection();
 st.close();
 return conn;
}
```

Caution:

Never cache the JDBC connection obtained using the helper method from the above example anywhere in your own code. Instead, call the helper method for each connection to avoid inadvertently holding a reference to a JDBC connection that might be used in a later request by another user. Due to the pooled services nature of the Oracle ADF runtime environment, you must not close the reference you are holding; however, do ensure that you close your statement.

Customizing Business Components Error Messages

You can use ADF Business Components custom message bundle to provide an alternative message string for an error code. This way, you can customize in-built error messages.

You can customize any of the builtin ADF Business Components error messages by providing an alternative message string for the error code in a custom message bundle.

Note:

The example in this section refers to the `oracle.summit.model.custommessages` package in the `SummitADF_Examples` application application workspace.

How to Customize Base ADF Business Components Error Messages

Assume you want to customize the builtin error message:

```
JBO-27014: Attribute OrderFilled in SOrd is required
```

If you have requested the Oracle Application Development Framework (Oracle ADF) source code from Oracle Worldwide Support, you can look in the `CSMessageBundle.java` file in the `oracle.jbo` package to see that this error message is related to the combination of the following lines in that message bundle file:

```
public class CSMessageBundle extends CheckedListResourceBundle {
  // etc.
  public static final String EXC_VAL_ATTR_MANDATORY = "27014";
  // etc.
  private static final Object[][] sMessageStrings = {
    // etc.
    {EXC_VAL_ATTR_MANDATORY, "Attribute {2} in {1} is required"},
    // etc.
  }
}
```

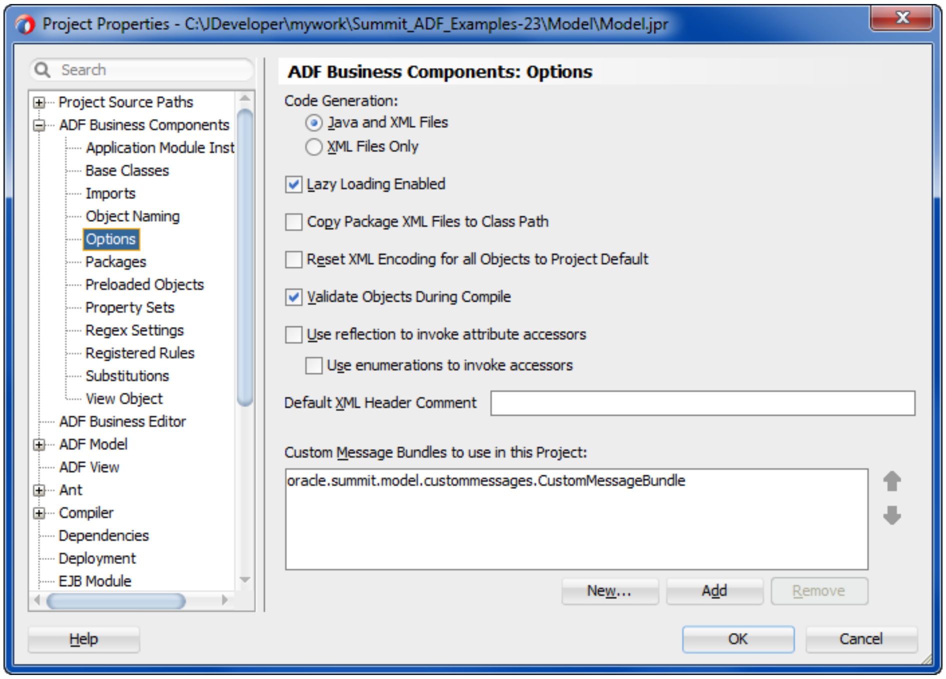
The numbered tokens `{2}` and `{1}` are error message placeholders. In this example the `{l}` is replaced at runtime with the name of the entity object and the `{2}` with the name of the attribute.

To create a custom message bundle file:

1. In the Applications window, right-click the project that you want to add the message bundle file to and choose **Project Properties**.
2. In the Project Properties dialog, select **Business Components > Options** and click **New**.
3. In the Create MessageBundle Class dialog, enter a name and package for the custom message bundle and click **OK**.

> **Note:**
>
> If the fully qualified name of your custom message bundle file does not appear in the **Custom Message Bundles to use in this Project** list, click the **Remove** button, then click the **Add** button to add the new message bundle file created. When the custom message bundle file is correctly registered, its fully qualified class name should appear in the list, as shown in Figure 17-8.

Figure 17-8 Project Properties Displays Message Resource Bundles



Project Properties - C:\JDeveloper\mywork\Summit_ADF_Examples-23\Model\Model.jpr

ADF Business Components: Options

Code Generation:
● Java and XML Files
○ XML Files Only

☑ Lazy Loading Enabled

☐ Copy Package XML Files to Class Path

☐ Reset XML Encoding for all Objects to Project Default

☑ Validate Objects During Compile

☐ Use reflection to invoke attribute accessors

☐ Use enumerations to invoke accessors

Default XML Header Comment

Custom Message Bundles to use in this Project:

oracle.summit.model.custommessages.CustomMessageBundle

4. In the Project Properties dialog, click **OK** to dismiss the **Project Properties** dialog and open the new custom message bundle class in the source editor.
5. Edit the two-dimensional `String` array in the custom message bundle class to contain any customized messages you'd like to use.

   The followig sample llustrates a custom message bundle class that overrides the error message string for the `JBO-27014` error to inform the user that a value must be provided for the named attribute. A second custom error message displays when a database constraint with the name `ORD_ORDER_FILLED_CK` is violated to inform the user of the expected Order Filled value. For more details about customizing error messages for constraints, see How to Customize Error Messages for Database Constraint Violations.

```
package oracle.summit.model.custommessages;

import java.util.ListResourceBundle;

public class CustomMessageBundle extends ListResourceBundle {
  private static final Object[][] sMessageStrings
    = new String[][] {
       {"27014","You must provide a value for {2}"},
       {"S_ORD_ORDER_FILLED_CK", "The order filled value must be Y or N"}
     };


  /* Return String Identifiers and corresponding Messages in a
   * two-dimensional array.
   */
  protected Object[][] getContents() {
    return sMessageStrings;
  }
}
```

What Happens When You Customize Base ADF Business Components Error Messages

After adding this message to your custom message bundle file, if you test the application using the Oracle ADF Model Tester and try to blank out the value of a mandatory attribute, you'll now see your custom error message instead of the default one:

```
JBO-27014: You must provide a value for Order Filled
```

You can add as many messages to the message bundle as you want. Any message whose error code key matches one of the built-in error message codes will be used at runtime instead of the default one in the `oracle.jbo.CSMessageBundle` message bundle.

How to Display Customize Error Messages as Nested Exceptions

When you customize ADF Business Components error messages, you will also need to customize the display of nested error messages. To accomplish this, you must create and register a custom error handler class.

When your business method throws an error, the ADF Model data binding layer intercepts the error and invokes the registered custom error handler class. In general, the error handler class is responsible for formatting the exception to be readable. During this process, the default error handler `DCErrorHandlerImpl` normally skips the top-level `JboException`, as this object is a wrapper over other business exceptions and does not have any business significance.

Although skipping the top-level exception is the desired behavior in the case of ADF Business Components errors, the default behavior will result in skipping the custom message you set for replacing the `SQLException`. To avoid this situation, while displaying each item in a nested exception, your custom error handler class must override `DCErrorHandlerImpl::skipException(Exception *ex*)` to decide whether to display the corresponding exception to the user in the final list or not.

Before you begin:

It may be helpful to have an understanding of application modules. For more information, see Customizing Business Components Error Messages.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Extending Business Components.

You will need to complete this task:

> Create the error message in the resource bundle, as described in How to Customize Base ADF Business Components Error Messages.

To provide custom messages for SQLExceptions in your project:

1. Create an error handler class that extends the default error handler `DCErrorHandlerImpl` interface provided by the ADF Model data binding layer.
2. In the error handler class, override the default error handler behavior for the `DCErrorHandlerImpl::skipException(Exception *ex*)` method, as shown in the following example.

   This overridden method is necessary to display each item in a nested exception, such as the ones returned for database-level error messages. You must implement logic to check for specifics exception types and, based on the business scenario, determine whether to display it in the list.
3. You can then register the custom error handler in your project's `DataBindings.cpx` file, as described in Customizing Error Handling.

The following example shows a custom implementation of the error handler that skips the `SQLIntegrityConstraintViolationException` from displaying in the error final list displayed to the user. You can choose to skip other database-level error message resulting from errors, such as unique constraint violations or foreign key constraint violations.

```
package view;

import java.sql.SQLIntegrityConstraintViolationException;

import oracle.adf.model.BindingContext;
import oracle.adf.model.RegionBinding;
import oracle.adf.model.binding.DCBindingContainer;
import oracle.adf.model.binding.DCErrorHandlerImpl;

import oracle.adf.model.binding.DCErrorMessage;

import oracle.jbo.DMLConstraintException;
import oracle.jbo.JboException;

public class CustomErrorHandler extends DCErrorHandlerImpl {

    public CustomErrorHandler() {
        super();
    }


    /**
     * If an exception is a RowValException or a TxnValException
     * and they have nested exceptions, then do not display
     * it.
     */
    @Override
    protected boolean skipException(Exception ex) {

        if (ex instanceof DMLConstraintException) {
            return false;
        } else if (ex instanceof SQLIntegrityConstraintViolationException) {
            return true;
        }
        return super.skipException(ex);
    }
)
```

How to Customize Error Messages for Database Constraint Violations

If you enforce constraints in the database, you might want to provide a custom error message in your Fusion web application to display to the end user when one of those constraints is violated. For example, assume a constraint called S_ORD_ORDER_FILLED_CK gets added to the application's S_ORD table using the following DDL statement shown in the following example.

```
alter table s_ord add (
  constraint S_ORD_ORDER_FILLED_CK
      check (ORDER_FILLED IN ('Y', 'N'))
);
```

To define a custom error message in your application, you add a message to a custom message bundle with the constraint name as the message key. The following example shows the CustomMessageBundle.java class when it defines a message with the key S_ORD_ORDER_FILLED_CK which matches the name of the database constraint name defined in previous example.

```
package oracle.summit.model.custommessages;

import java.util.ListResourceBundle;

public class CustomMessageBundle extends ListResourceBundle {
  private static final Object[][] sMessageStrings
    = new String[][] {
       {"27014","You must provide a value for {2}"},
       {"S_ORD_ORDER_FILLED_CK", "The order filled value must be Y or N"}
     };

  protected Object[][] getContents() {
    return sMessageStrings;
  }
}
```

How to Implement a Custom Constraint Error Handling Routine

If the default facility for assigning a custom message to a database constraint violation does not meet your needs, you can implement your own custom constraint error handling routine. Doing this requires creating a custom framework extension class for the ADF Business Components transaction class, which you then configure your application module to use at runtime.

Creating a Custom Database Transaction Framework Extension Class

To write a custom framework extension class for the ADF Business Components transaction, create a class like the `CustomDBTransactionImpl` shown in the following example. This example overrides the transaction object's `postChanges()` method to wrap the call to `super.postChanges()` with a `try/catch` block in order to perform custom processing on any `DMLConstraintException` errors that might be thrown. In this simple example, the only custom processing being performed is a call to `ex.setExceptions(null)` to clear out any nested detail exceptions that the `DMLConstraintException` might have. Instead of this, you could perform any other kind of custom exception processing required by your application, including throwing a *custom* exception, provided your custom exception extends `JboException` directly or indirectly.

```
package oracle.summit.model.custommessages;

import oracle.jbo.DMLConstraintException;
import oracle.jbo.server.DBTransactionImpl2;
import oracle.jbo.server.TransactionEvent;

public class CustomDBTransactionImpl  extends DBTransactionImpl2 {
  public void postChanges(TransactionEvent te) {
    try {
      super.postChanges(te);
    }
    /*
     * Catch the DML constraint exception
     * and perform custom error handling here
     */
    catch (DMLConstraintException ex) {
      ex.setExceptions(null);
      throw ex;
    }
  }
}
```

Configuring an Application Module to Use a Custom Database Transaction Class

In order for your application module to use a custom database transaction class at runtime, you must:

1. Provide a custom implementation of the `DatabaseTransactionFactory` class that overrides the `create()` method to return an instance of the customized transaction class.
2. Configure the value of the `TransactionFactory` property to be the fully qualified name of this custom transaction factory class.

The following example shows a custom database transaction factory class that does this. It returns a new instance of the `CustomDBTransactionImpl` class when the framework calls the `create()` method on the database transaction factory.

```
package oracle.summit.model.custommessages;

import oracle.jbo.server.DBTransactionImpl2;
import oracle.jbo.server.DatabaseTransactionFactory;
```

```
public class CustomDatabaseTransactionFactory
                     extends DatabaseTransactionFactory {
  public CustomDatabaseTransactionFactory() {
  }
  /**
   * Return an instance of our custom CustomDBTransactionImpl class
   * instead of the default implementation.
   *
   * @return instance of custom CustomDBTransactionImpl implementation.
   */
  public DBTransactionImpl2 create() {
    return new CustomDBTransactionImpl();
  }
}
```

To complete the job, use the Properties tab of the overview editor for application module configurations (on the `bc4j.xcfg` file) to assign the value `oracle.summit.model.custommessages.CustomDatabaseTransactionFactory` to the `TransactionFactory` property. To display the application module configuration overview editor, double-click the application module in the Applications window and, in the overview editor, select the **Configurations** navigation tab. Then, in the Configurations page of the overview editor, click the configuration hyperlink. In the application module configuration overview editor, select the **Properties** tab and click **Add Property** to select the following property from the Add Property dialog and click **OK**.

- `TransactionFactory`

Then, in the **Properties** list enter the value for the `TransactionFactory` property:

- `TransactionFactory` = `oracle.summit.model.custommessages.CustomDatabaseTransactionFactory`

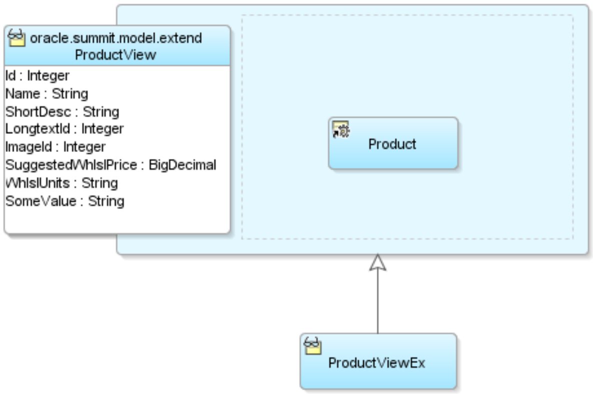
When you run the application using this configuration, your custom transaction class will be used.

Creating Extended Components Using Inheritance

ADF Business Components supports inheritance. It allows you to create a new business component by extending an existing one, which inherits all properties of the original, and thence customize the new one.

Whenever you create a new business component, if necessary, you can extend an existing one to create a customized version of the original. As shown in Figure 17-9, the `ProductViewEx` view object extends the `ProductView` view object to add a bind variable named `bv_ProductName` and to customize the `WHERE` clause to reference that bind variable.

Figure 17-9 ADF Business Components Can Extend Another Component



While the figure shows a view object example, this component inheritance facility is available for all component types. When one component extends another, the extended component inherits all of the metadata and behavior from the parent it extends. In the extended component, you can add new features or customize existing features of its parent component both through metadata and Java code.

> **Note:**
>
> The example in this section refers to the `oracle.summit.model.extend` package in the `SummitADF_Examples` application workspace.

How To Create a Component That Extends Another

To create an extended component, use the component wizard in the **New Gallery** for the type of component you want to create. For example, to create an extended view object, you use the Create View Object wizard. On the **Name** page of the wizard — in addition to specifying a name and a package for the new component — provide the fully qualified name of the component that you want to extend in the **Extends** field. To pick the component name from a list, use the **Browse** button next to the **Extends** field. Then, continue to create the extended component in the normal way using the remaining panels of the wizard.

How To Extend a Component After Creation

After you define an extended component, JDeveloper lets you change the parent component from which an extended component inherits. You can use the overview editor for the component to accomplish this.

To change the parent component after creation:

1. In the Applications window, double-click the component.
2. In the overview editor, click the **General** navigation tab and then click the **Refactor object extends** button next to the **Extends** field.

3. In the Select Parent dialog, choose the desired component to extend from the package list and click **OK**.

To change the extended component to not inherit from any parent, select the **None** checkbox in the Select Parent dialog. This has the same effect as if you deleted the component and re-created to accomplish this.

What Happens When You Create a Component That Extends Another

Business components that you create in an ADF Business Component data model project are comprised of an XML document and an optional Java class. When you create a component that extends another, JDeveloper reflects this component inheritance in both the XML document and in any generated Java code for the extended component.

Attributes in an Extended Component Inherited from the Parent Component

When you create an extended component, the extended component inherits the attributes and attribute properties of the parent component. This connection is maintained because the inherited attributes are not defined in the extended component, they are simply passed through from the parent component. Therefore, if after creating an extended component you subsequently modify an attribute in the parent component, the extended component's attribute reflects that modification.

For example, say you create a `ProductViewEx` extended view object that extends a `ProductView` view object. Then you decide to set the **Display** UI hint for the `ShortDesc` attribute to **Hide** in the `ProductView` parent view object. When you do this, the `ShortDesc` attribute is hidden in the `ProductViewEx` extended view object as well.

However, if you override an attribute in the extended object, this connection is severed because the overridden attribute is redefined in the extended object. This allows you to make a change in an extended object without impacting the parent object or any other objects derived from that parent object. Note that the override applies only to the selected attribute, the other inherited attributes are unaffected.

For example, say you have a `ProductView` view object and two extended view objects, `ProductViewEx1` and `ProductViewEx2`. After setting the **Display** UI hint for the `ShortDesc` attribute to **Hide** in the `ProductView` parent view object, you decide to set it to **Show** in the `ProductViewEx1` extended view object. When you override `ShortDesc` in `ProductViewEx1` and set the **Display** UI hint to **Show**, the change is only reflected in the `ProductViewEx1` extended view object, not in the `ProductViewEx2` extended view object or the `ProductView` parent view object.

Attributes Added to the Extended Component's XML Descriptor

JDeveloper notes the name of the parent component in the new component's XML document by adding an `Extends` attribute to the root component element. Any new declarative features you add or any aspects of the parent component's definition you've overridden appear in the extended component's XML document. In contrast, metadata that is purely inherited from the parent component is not repeated for the extended component.

The following example shows what the `ProductViewEx.xml` XML document for the `ProductViewEx` view object looks like. Notice the `Extends` attribute on the `ViewObject` element, the `Variable` element related to the additional bind variable added in the extended view object, and the overridden value of the Where attribute for the WHERE clause that was modified to reference the theProductName bind variable.

```
<ViewObject
  xmlns="http://xmlns.oracle.com/bc4j"
  Name="ProductViewEx"
  InheritPersonalization="true"
  BindingStyle="OracleName"
  CustomQuery="false"
  ComponentClass="oracle.summit.model.extend.ProductViewExImpl"
  RowClass="oracle.summit.model.extend.ProductViewExRowImpl"
  RowInterface="oracle.summit.model.extend.common.ProductViewExRow"
  ClientRowProxyName="oracle.summit.model.extend.client.ProductViewExRowClient"
  ComponentInterface="oracle.summit.model.extend.common.ProductViewEx"
  ClientProxyName="oracle.summit.model.extend.client.ProductViewExClient"
  Extends="oracle.summit.model.extend.ProductView"
  Where="UPPER(PRODUCT_NAME) LIKE UPPER(:theProductName)||'%'"
...
  <Variable
     Name="bv_ProductName"
     Kind="where"
     Type="java.lang.String"/>
...
</ViewObject>
```

Java Classes Generated for an Extended Component

If you enable custom Java code for an extended component, JDeveloper automatically generates the Java classes to extend the respective Java classes of its parent component. In this way, the extended component can override any aspect of the parent component's programmatic behavior as necessary. If the parent component is an XML-only component with no custom Java class of its own, the extended component's Java class extends whatever base Java class the parent would use at runtime. This could be the default ADF Business Components framework class in the oracle.jbo.server package, or could be your own framework extension class if you have specified that in the **Extends** dialog of the parent component.

In addition, if the extended component is an application module or view object and you enable client interfaces on it, JDeveloper automatically generates the extended component's client interfaces to extend the respective client interfaces of the parent component. If the respective client interface of the parent component does not exist, then the extended component's client interface directly extends the appropriate base ADF Business Components interface in the oracle.jbo package.

What You May Need to Know About Extending Components

Parent Classes and Interfaces for Extended Components

Since an extended component is a customized version of its parent, code you write that works with the *parent* component's Java classes or its client interfaces works without incident for either the parent component *or* any customized version of that parent component.

For example, assume you have a base ProductView view object with custom Java classes and client interfaces like:

- class `ProductViewImpl`
- row class `ProductViewRowImpl`
- client interface `ProductView`
- row client interface `ProductViewRow`

If you create a `ProductViewEx` view object that extends `ProductView`, then you can use the base component's classes and interface to work both with `ProductView` and `ProductViewEx`.

The following example illustrates a test client program that works with the `ProductView`, `ProductViewRow`, `ProductViewEx`, and `ProductViewExRow` client interfaces. A few interesting things to note about the example are the following:

1. You can use parent `ProductView` interface for working with the `ProductViewEx` view object that extends it.
2. Alternatively, you can cast an instance of the `ProductViewEx` view object to its own more specific `ProductViewEx` client interface.
3. You can test if row `ProductViewRow` is actually an instance of the more specific `ProductViewExRow` before casting it and invoking a method specific to the `ProductViewExRow` interface.

```
package oracle.summit.model.extend;

import oracle.jbo.ApplicationModule;
import oracle.jbo.AttributeDef;
import oracle.jbo.Row;
import oracle.jbo.StructureDef;
import oracle.jbo.ValidationException;
import oracle.jbo.ViewObject;
import oracle.jbo.client.Configuration;

import oracle.summit.model.extend.common.ProductView;
import oracle.summit.model.extend.common.ProductViewEx;
import oracle.summit.model.extend.common.ProductViewExRow;
import oracle.summit.model.extend.common.ProductViewRow;


public class TestClient {
    public static void main(String[] args) {
    String        amDef = "oracle.summit.model.extend.AppModule";
    String        config = "AppModuleLocal";
    ApplicationModule am =
                         Configuration.createRootApplicationModule(amDef,config);
    ProductView products = (ProductView)am.findViewObject("ProductView1");
    products.executeQuery();
    ProductViewRow product = (ProductViewRow)products.first();
    printAllAttributes(products,product);
    testSomethingOnProductsRow(product);
    products = (ProductView)am.findViewObject("ProductViewEx1");
    ProductViewEx productsByName = (ProductViewEx)products;
    productsByName.setbv_ProductName("bunny");
    productsByName.executeQuery();
    product = (ProductViewRow)productsByName.first();
    printAllAttributes(productsByName,product);
    testSomethingOnProductsRow(product);
    am.getTransaction().rollback();
    Configuration.releaseRootApplicationModule(am,true);
```

```
  }
  private static void testSomethingOnProductsRow(ProductViewRow product) {
    try {
      if (product instanceof ProductViewExRow) {
        ProductViewExRow productByName = (ProductViewExRow)product;
        productByName.someExtraFeature("Test");
      }
      product.setName("Q");
      System.out.println("Setting the Name attribute to 'Q' succeeded.");
    }
    catch (ValidationException v) {
      System.out.println(v.getLocalizedMessage());
    }
  }
  private static void printAllAttributes(ViewObject vo, Row r) {
    String viewObjName = vo.getName();
    System.out.println("Printing attribute for a row in VO '"+
                       viewObjName+"'");
    StructureDef def = r.getStructureDef();
    StringBuilder sb = new StringBuilder();
    int numAttrs = def.getAttributeCount();
    AttributeDef[] attrDefs = def.getAttributeDefs();
    for (int z = 0; z < numAttrs; z++) {
      Object value = r.getAttribute(z);
      sb.append(z > 0 ? "  " : "")
        .append(attrDefs[z].getName())
        .append("=")
        .append(value == null ? "<null>" : value)
        .append(z < numAttrs - 1 ? "\n" : "");
    }
    System.out.println(sb.toString());
  }
}
```

Running the test client produces the following results.

```
Printing attribute for a row in VO 'ProductView1'
Id=10011
  Name=Bunny Boot
  ShortDesc=Beginners ski boot
  LongtextId=518
  ImageId=1001
  SuggestedWhlslPrice=150
  WhlslUnits=<null>
  SomeValue=I am from the Product Impl Class
Setting the Name attribute to 'Q' succeeded.
```

Note:

In this example, `ProductView` is an entity-based view object based on the `Product` entity object. The `Product` entity object includes a transient `SomeValue` attribute that returns the string "I am from the Product Impl class". You'll learn more about why this was included in the example in Substituting Extended Components in a Delivered Application.

Generated Classes for Extended Components

When you create an extended component, the **Class Extends** button on the Java page of the extended component's wizard is disabled. Additionally, in the application module editor's Java page, when you click **Edit java options**, the **Class Extends** button in the Java dialog appears disabled. This is due to the fact that JDeveloper automatically extends the appropriate class of its parent component, so it does not make sense to allow you to select a different class.

Business Component Types

Entity Objects
When you create an extended entity object, you can introduce new attributes, new associations, new validators, and new custom code. You can override certain declarative aspects of existing attributes as well as overriding any method from the parent component's class. You may not remove from the extended entity object any attributes that the base class defines from the extended entity object.

View Objects
When you create an extended view object, you can introduce new attributes, new **view links**, new bind variables, and new custom code. You can override certain declarative aspects of existing attributes as well as overriding any method from the parent component's class. You may not remove from the extended view object any attributes that the base class defines.

Application Modules
When you create an extended application module, you can introduce new view object instances or new nested application module instance and new custom code. You can also override any method from the parent component's class from the extended view object.

New Attributes in an Extended Component

If you add new attributes in an extended entity object or view object, the attribute index numbers are computed relative to the parent component. For example, consider the `ProductView` view object mentioned in Parent Classes and Interfaces for Extended Components. If you enable a custom view row class, it might have attribute index constants defined in the `ProductViewRowImpl.java` class like this:

```
public class ProductViewRowImpl extends SummitViewRowImpl
                             implements ProductViewRow {

  /**
   * AttributesEnum: generated enum for identifying attributes and accessors.
   * Do not modify.
   */
  public enum AttributesEnum {...}

  public static final int ID = AttributesEnum.Id.index();
  public static final int NAME = AttributesEnum.Name.index();
  public static final int SHORTDESC = AttributesEnum.ShortDesc.index();
  public static final int LONGTEXTID = AttributesEnum.LongtextId.index();
  public static final int IMAGEID = AttributesEnum.ImageId.index();
  public static final int SUGGESTEDWHLSLPRICE =
                              AttributesEnum.SuggestedWhlslPrice.index();
  public static final int WHLSLUNITS = AttributesEnum.WhlslUnits.index();
  public static final int SOMEVALUE = AttributesEnum.SomeValue.index();
  //etc.
}
```

When you create an extended view object like `ProductViewEx`, if that view object adds an additional attribute like `SomeExtraAttr` and has a custom view row class enabled, then its attribute constants will be computed relative to the maximum value of the attribute constants in the parent component:

```
public class ProductViewExRowImpl extends ProductViewRowImpl
                                   implements ProductViewExRow {

  public static final int MAXUSAGECONST = 1;

  public enum AttributesEnum {
      SomeExtraAttr {
          public Object get(ProductViewExRowImpl obj) {
              return obj.getSomeExtraAttr();
          }

          public void put(ProductViewExRowImpl obj, Object value) {
              obj.setAttributeInternal(index(), value);
          }
      }

      private static AttributesEnum[] vals = null;
      private static int firstIndex =
          ViewDefImpl.getMaxAttrConst("oracle.summit.model.extend.ProductView");

      public abstract Object get(ProductViewExRowImpl object);

      public abstract void put(ProductViewExRowImpl object, Object value);

      public int index() {
          return AttributesEnum.firstIndex() + ordinal();
      }

      public static int firstIndex() {
          return firstIndex;
      }

      public static int count() {
          return AttributesEnum.firstIndex() +
                                 AttributesEnum.staticValues().length;
      }

      public static AttributesEnum[] staticValues() {
          if (vals == null) {
              vals = AttributesEnum.values();
          }
          return vals;
      }
  }
  public static final int SOMEEXTRAATTR = AttributesEnum.SomeExtraAttr.index();
...
}
```

Substituting Extended Components in a Delivered Application

ADF Business Components supports application customization without needing to have access to the application’s source code. It completely negates maintenance expenses and human errors involved in customizations.

If you deliver packaged applications that can require on-site customization for each potential client of your solution, ADF Business Components offers a useful feature to simplify that task.

> **Note:**
>
> The example in this section refers to the `ExtendedProject` project in the `SummitADF_Examples` application workspace.

All too often, on-site application customization is performed by making direct changes to the source code of the delivered application. This approach demonstrates its weaknesses whenever you deliver patches or new feature releases of your original application to your clients. Any customizations they had been applied to the base application's source code need to be painstakingly reapplied to the patched or updated version of the base application. Not only does this render the application customization a costly, ongoing maintenance expense, it can introduce subtle bugs due to human errors that occur when reapplying previous customizations to new releases.

ADF Business Components offers a superior, component-based approach to support application customization that doesn't require changing — or even having access to — the base application's source code. To customize your delivered application, your customers can:

1. Import one or more packages of components from the base application into a new project.
2. Create new components to effect the application customization, extending appropriate parent components from the base application as necessary.
3. Define a list of global component substitutions, naming their customized components to substitute for your base application's appropriate parent components.

When the customer runs your delivered application with a global component substitution list defined, their customized application components are used by your delivered application without changing any of its code. When you deliver a patched or updated version of the original application, their component customizations apply to the updated version the next time they restart the application without needing to reapply any customizations.

How To Substitute an Extended Component

To define global component substitutions, use the Project Properties dialog in the project where you have created extended components based on the imported components from the base application.

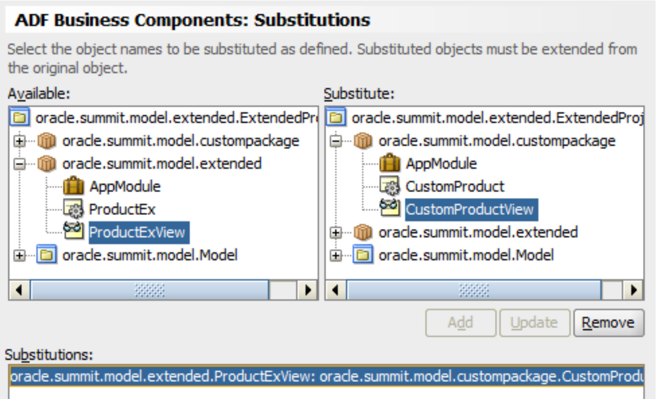
> **Note:**
>
> You can only substitute a component in the base application with an extended component that inherits directly or indirectly from the base one.

To substitute an extended component:

1. In the Applications window, right-click the project that you want to add the extended component to and choose **Project Properties**.
2. In the Project Properties dialog, select **ADF Business Components > Substitutions** and select the base application's component in the **Available** list.
3. In the **Substitute** list, select the customized, extended component to substitute .
4. Click **Add**.

   For example, assume that you have created the view object `CustomizedProduct` in a package that extends the base view object `Products`. To substitute the `CustomizedProducts` view object for the legacy `Products` view object, you would select these view objects as shown in Figure 17-10 to define the component substitution.

Figure 17-10   Defining Business Component Substitutions



ADF Business Components: Substitutions

Select the object names to be substituted as defined. Substituted objects must be extended from the original object.

Available:
oracle.summit.model.extended.ExtendedPr
oracle.summit.model.custompackage
oracle.summit.model.extended
AppModule
ProductEx
ProductExView
oracle.summit.model.Model
Substitute:
oracle.summit.model.extended.ExtendedProj
oracle.summit.model.custompackage
AppModule
CustomProduct
CustomProductView
oracle.summit.model.extended
oracle.summit.model.Model
Add
Update
Remove
Substitutions:

oracle.summit.model.extended.ProductExView: oracle.summit.model.custompackage.CustomProdu

What Happens When You Substitute

When you define a list of global component substitutions in a project named *`ExtendedProject`*, the substitution list is saved in the *`ExtendedProject`*`.jpx` in the root directory of the source path.

The file will contain `Substitute` elements as shown in the following example, one for each component to be substituted.

```
<JboProject
   Name="ExtendedProject"
   SeparateXMLFiles="true"
   PackageName="oracle.summit.model.extended" >
   <Containee
      Name="custompackage"
      FullName="oracle.summit.model.custompackage"
      ObjectType="JboPackage" >
   </Containee>
   <Containee
      Name="extended"
      FullName="oracle.summit.model.extended"
      ObjectType="JboPackage" >
   </Containee>
   <AppContainee
      Name="Model"
      FullName="oracle.summit.model.Model"
      ObjectType="JboProject">
   <Substitutes>
      <Substitute
        OldName="oracle.summit.model.extended.ProductEx"
        NewName="oracle.summit.model.custompackage.CustomProduct" />
   </Substitutes>
</JboProject>
```

How to Enable the Substituted Components in the Base Application

To have the original application use the set of substituted components, define the Java system property Factory-Substitution-List and set its value to the name of the project whose *.jpx file contains the substitution list. The value should be just the project name without any *.jpr or *.jpx extension.

For example, consider a simple example that customizes the Product entity object and the ProductView view object described in Parent Classes and Interfaces for Extended Components. To perform the customization, assume you create new project named ExtendedProject that:

- Defines a library for the JAR file containing the base components
- Imports the package containing Product and ProductView
- Creates new extended components in a distinct package name called CustomizeProduct and CustomizeProductView
- Defines a component substitution list to use the extended components.

When creating the extended components, assume that you:

- Added an extra view attribute named SomeExtraAttribute to the ProductViewEx view object.
- Added a new validation rule to the CustomizedProduct entity object to enforce that the product name cannot be the letter "Q".
- Overrode the getChecksum() method in the CustomizedProduct.java class to return "**I am the CustomizedProduct Class**".

If you define the Factory-Substitution-List Java system property set to the value ExtendsAndSubstitutes, then when you run the exact same test client class described in Parent Classes and Interfaces for Extended Components the output of the sample will change to reflect the use of the substituted components.

Part III

Using the ADF Model Layer

This part describes tasks that developers can perform in the ADF Model layer of the Fusion web application.

Part III contains the following chapters:

18
Using ADF Model in a Fusion Web Application

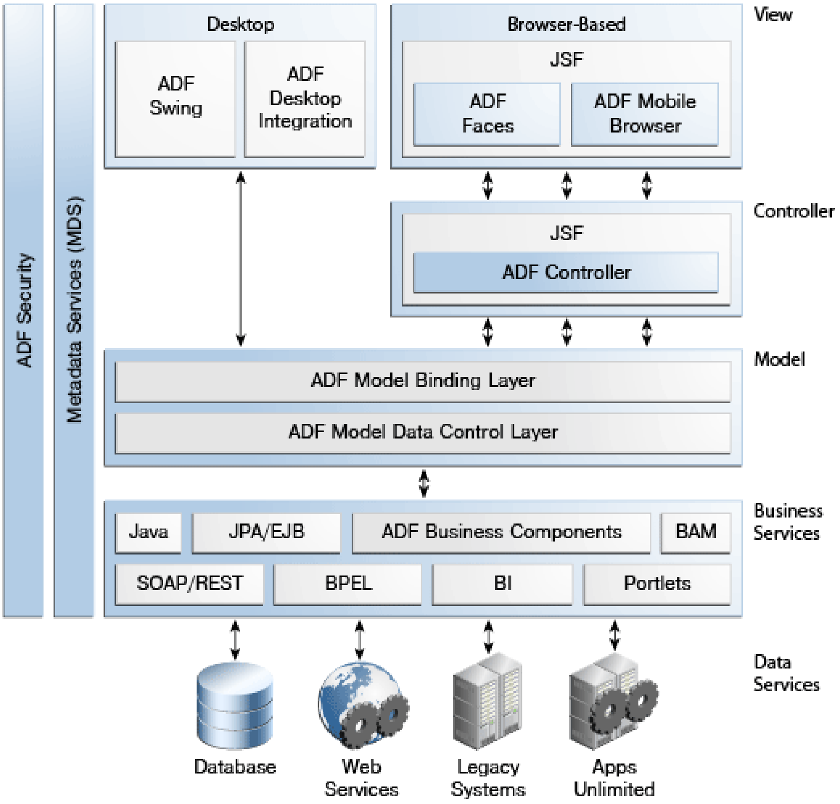
This chapter describes JDeveloper tools that you use to create databound pages from data modeled with ADF Business Components, using ADF data controls in a Fusion web application. Specifically, it describes how an ADF application module data model and ADF Business Components client interface methods appear at design time for drag and drop data binding. It also describes how they are accessible at runtime by the ADF Model data binding layer using ADF data controls.
This chapter includes the following sections:

About Using ADF Model in a Fusion Web Application

ADF Model is build on model-view-controller design where all the components are separated. This supports the development of Fusion Web application in a more structured way.
ADF Model implements concepts that enable decoupling the user interface technology from the business service implementation: **data controls** and declarative **ADF bindings**.

ADF Model builds upon the MVC (model-view-controller) design pattern, in which the code for the application's data model, visual interface, and application flow are all cleanly separated. As illustrated in Figure 18-1, ADF Model provides a layer of abstraction over the business services that typically serve as an application's model layer. For more information on how ADF Model fits in with the MVC architecture, see "Abstraction of the Application's Model Layer" in *Understanding Oracle Application Development Framework*.

Figure 18-1 ADF Model within the ADF Architecture



About ADF Data Controls

ADF Data Controls contains the functionality of the application module and created automatically when you create an application module.
ADF data controls abstract the implementation technology of a business service by using standard metadata interfaces to describe the service's operations and data collections, including information about the properties, methods, and types involved. In an application that uses **ADF Business Components**, a data control is automatically created when you create an **application module**, and it contains all the functionality of the application module. Developers can then use the representation of the data control displayed in JDeveloper's Data Controls panel to create UI components that are automatically bound to the application module. At runtime, the ADF Model layer reads the information describing the data controls and bindings from the appropriate XML files and then implements the two-way connection between the user interface and the business service.

When you create an ADF Business Component application module and add **view objects** to the data model for that application module, a data control is created for you automatically.

You can also create data controls based on different types of business services, such as EJB session beans, plain Java objects, and web services. For information on working with these kinds of data controls, see Using ADF Data Controls in *Developing Applications with Oracle ADF Data Controls*.

> **Note:**
>
> Using the ADF Model layer to perform business service access ensures that the view and the business service stay in sync. For example, while you could call a method on an application module by class-casting the data control reference to the **application module instance** and then calling the method directly, doing so would bypass the model layer and it would then become unaware of any changes.

About JSF Data Binding

You can use simple expression language to reference an object in JSF, on which the ADF bindings are based, which makes it easy to build a dynamic data-driven user interface..
ADF declarative bindings are based on JSF data binding. In JSF, you use a simple **expression language** (called EL) to bind to the information you want to present and/or modify. Example expressions look like `#{userInfoBean.principalName}` to reference a particular user's name, or `#{userInfoBean.principalName eq 'SKING'}` to evaluate whether a user's name is SKING or not. At runtime, a generic expression evaluator returns the `String` and `boolean` value of these respective expressions, automating access to the individual objects and their properties without requiring code.

At runtime, the value of certain JSF UI components is determined by the `value` attribute. While a component can have static text as its value, typically the `value` attribute will contain a binding that is an EL expression that the runtime infrastructure evaluates to determine what data to display. For example, an `outputText` component that displays the name of the currently logged-in user might have its `value` attribute set to the expression `#{userInfoBean.principalName}`. Since any attribute of a component can be assigned a value using an EL expression, it's easy to build dynamic, data-driven user interfaces. For example, you could hide a component when a user is not logged in by using a boolean-valued expression like `#{userInfoBean.prinicpalName !=null}` in the UI component's `rendered` attribute. If there is no principal name in the current instantiation of the `userInfoBean`, the `rendered` attribute evaluates to `false` and the component disappears from the page.

About ADF Data Binding

You can design a page by dragging and dropping ADF data controls as UI component on a page with little knowledge of code.. The code and objects that are needed to bind the component to data control are automatically created by JDeveloper.
ADF data binding extends JSF data binding by enabling you to bind to ADF data controls declaratively. In a typical JSF application, you would create objects like the `userInfoBean` object as a managed bean. The JSF runtime manages instantiating these beans on demand when any EL expression references them for the first time. However, in an application that uses ADF Model, instead of binding the UI component attributes to properties or methods on managed beans, JDeveloper automatically binds the UI component attributes to ADF Model, which uses XML configuration files that drive generic data binding features.

ADF declarative bindings abstract the details of accessing data from data collections in a data control and of invoking its operations. There are three basic kinds of declarative binding objects:

- Executable bindings: Included in executable bindings are iterator bindings, which simplify the building of user interfaces that allow scrolling and paging through collections of data and drilling-down from summary to detail information. Executable bindings also include bindings that allow searching and nesting a series of pages within another page, as well as bindings that cause operations to occur immediately.
- Value bindings: Used by UI components that display data. Value bindings range from the most basic variety that work with a simple text field to more sophisticated list and tree bindings that support the additional needs of list, table, and tree UI controls.
- Action bindings: Used by UI components like hyperlinks or buttons to invoke built-in or custom operations on data collections or a data control without writing code.

The group of bindings supporting the UI components on a page are described in a page-specific XML file called the **page definition file**. The ADF Model layer uses this file at runtime to instantiate the page's bindings. These bindings are held in a request-scoped map called the **binding container**, accessible during each page request using the EL expression `#{bindings}`. This expression always evaluates to the binding container for the current page.

> **Tip:**
>
> The current binding container is also available from `AdfContext` for programmatic access.

You can design a databound user interface by dragging an item from the Data Controls panel and dropping it on a page as a specific UI component. When you use data controls to create a UI component, JDeveloper automatically creates the various code and objects needed to bind the component to the data control you selected.

Additional Functionality for ADF Model Layer

ADF Model provides a different layer of configuration with different functionalities, which are helpful in creating a Fusion Web Application with ease.
You may find it helpful to understand other **Oracle ADF** features before you configure or use the ADF Model layer. Additionally, you may want to read about what you can do with your model layer configurations. Following are links to other functionality that may be of interest.

- ADF model and data binding rely on the business layer. In most cases, you will be working with representations of your view objects. For more information, see Defining SQL Queries Using View Objects. You may also want to be familiar with advanced functionality discussed in Tuning View Object Performance.
- You can use the ADF Model layer to create your view pages. After reading this chapter for the basic information on how data binding in the view layer works, you should refer to the chapters in Creating a Databound Web User Interface for information about using data binding to create specific view functionality.
- For information about how ADF Model works with the page lifecycle, see Understanding the Fusion Page Lifecycle .

- For a complete list of configuration parameters that affect the model layer, see DataBindings.cpx and pageNamePageDef.xml.
- For a complete list of binding properties, see Oracle ADF Binding Properties.

Exposing Application Modules with ADF Data Controls

In ADF, the application module components directly implement the interfaces to optimize the interaction between business components and data in the application by binding the objects for data collections. Because of the optimized interaction, the bindings work directly with the application module instances.
When you create an application module, a data control is created for the objects that you have added to the application module's data model.

> **Note:**
>
> You can also create data controls for other kinds of business services. See Using ADF Data Controls in *Developing Applications with Oracle ADF Data Controls*.

Importantly, the application module component directly implements interfaces that binding objects expected for data collections, built-in operations, and service methods. This optimized interaction allows the bindings to work *directly* with the application module instances in its data model in the following ways:

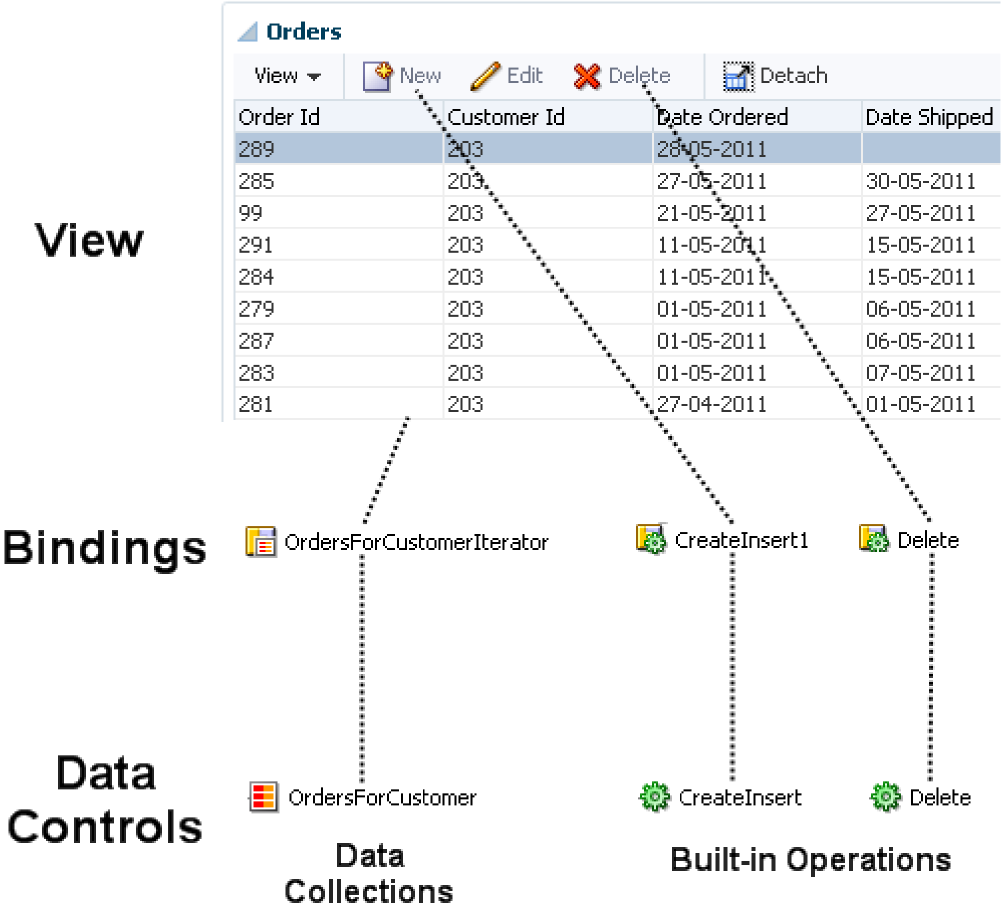
- Iterator bindings directly bind to the default **row set iterator** of the default row set of any view object instance. The row set iterator manages the current object and current range information.

  > **Tip:**
  >
  > You can also use the iterator binding to bind to a secondary named row set that you have created. To bind to a secondary row set iterator, you need to use the `RSIName`. For information about the difference between the default row set and secondary row sets and how to create them, see Working with Multiple Row Sets and Row Set Iterators.

- Action bindings directly bind to either:
  - Custom methods on the data control client interface
  - Built-in operations of the application module and view objects

Figure 18-2 illustrates examples of iterator and action bindings connecting UI components with **application module data control** objects.

Figure 18-2 Bindings Connect UI Components to Data Controls

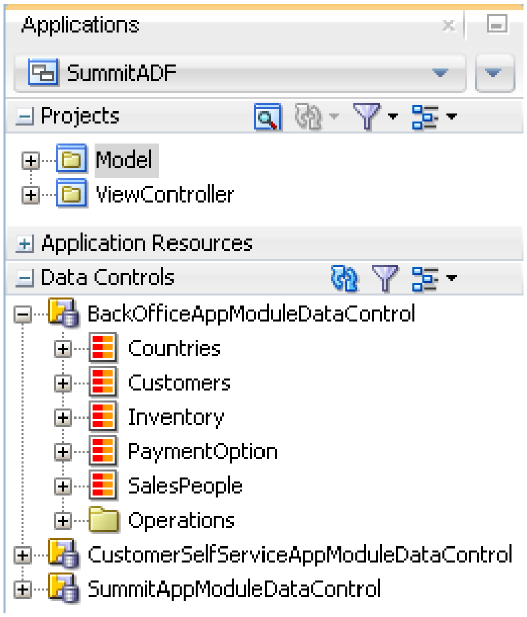


Order Id	Customer Id	Date Ordered	Date Shipped
289	203	28-05-2011	
285	203	27-05-2011	30-05-2011
99	203	21-05-2011	27-05-2011
291	203	11-05-2011	15-05-2011
284	203	11-05-2011	15-05-2011
279	203	01-05-2011	06-05-2011
287	203	01-05-2011	06-05-2011
283	203	01-05-2011	07-05-2011
281	203	27-04-2011	01-05-2011

How an Application Module Data Control Appears in the Data Controls Panel

You use the Data Controls panel to create databound UI components by dragging and dropping icons from the panel onto the visual editor for a page. Figure 18-3 shows the Data Controls panel displaying the data controls for the Summit sample application for Oracle ADF.

Figure 18-3 Data Controls Panel in JDeveloper



The Data Controls panel lists all the data controls that have been created for the application's business services and exposes all the collections (row sets of data objects), methods, and built-in operations that are available for binding to UI components.

Note:

If you've configured JDeveloper to expose them, any **view link accessor** returns are also displayed. For more information, see Working with Multiple Tables in a Master-Detail Hierarchy. To view the accessor methods:

1. From the JDeveloper main menu, choose **Tools > Preferences**.
2. Select the **Data Controls Panel** node.
3. Select **Show Underlying Accessor Nodes** to activate the checkbox.

In an application that uses ADF Business Components to define the business services, each data control on the Data Controls panel represents a specific application module, and exposes the view object instances in that application's data model. The hierarchy of objects in the data control is defined by the **view links** between view objects that have specifically been added to the **application module data model**. For information about creating view objects and view links, see Defining SQL Queries Using View Objects. For information about adding view links to the data model, see How to Enable Active Master-Detail Coordination in the Data Model.

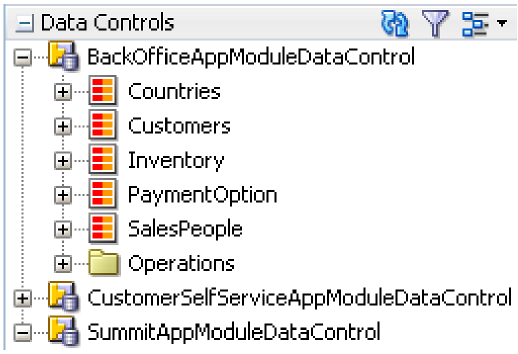
> **Tip:**
>
> You can open the overview editor for a view object by right-clicking the associated data control object and choosing **Edit Definition**.

For example, `BackOfficeAppModuleDataControl` represents the `BackOfficeAppModule` application module, which implements the part of the business service layer of the Summit ADF sample application that is available to the company's employees. The application module's data model contains numerous view object instances, including several master-detail hierarchies. The view layer of the Summit ADF sample application consists of JSF pages whose UI components are bound to data from the view object instances in `BackOfficeAppModule`'s data model, and to built-in operations and service methods on its client interface.

How the Data Model and Service Methods Appear in the Data Controls Panel

Each view object instance appears as a named data collection whose name matches the view object instance name. Figure 18-4 illustrates how the Data Controls panel displays the view object instances in `BackOfficeAppModule`'s data model. The Data Controls panel reflects the master-detail hierarchies in your application module data model by displaying detail data collections nested under their master data collection.

Figure 18-4 How the Data Model Appears in the Data Controls Panel


Data Controls
BackOfficeAppModuleDataControl
Countries
Customers
Inventory
PaymentOption
SalesPeople
Operations
CustomerSelfServiceAppModuleDataControl
SummitAppModuleDataControl

The Data Controls panel also displays any custom method on the application module's client interface as a named data control custom operation whose name matches the method name. If a method accepts arguments, they appear in a Parameters node as operation parameters nested inside the operation's node.

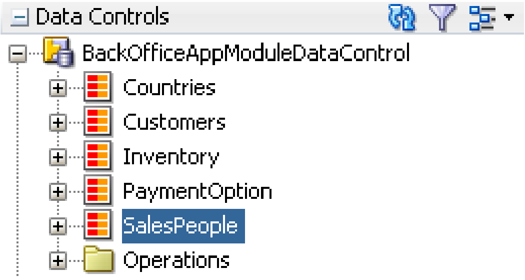
How Transaction Control Operations Appear in the Data Controls Panel

The Operations folder node of the application module data control exposes two data control built-in operations named `Commit` and `Rollback`, as shown in Figure 18-5. At runtime, when these operations are invoked by the data binding layer, they delegate to the `commit()` and `rollback()` methods of the `Transaction` object associated with the current application module instance.

Note:

In an application module with many view object instances and custom methods, you may need to scroll the Data Controls panel display to find the Operations node that is the direct child node of the data control. This node is the one that contains these built-in operations.

Figure 18-5 How Transaction Control Operations Appear in the Data Controls panel

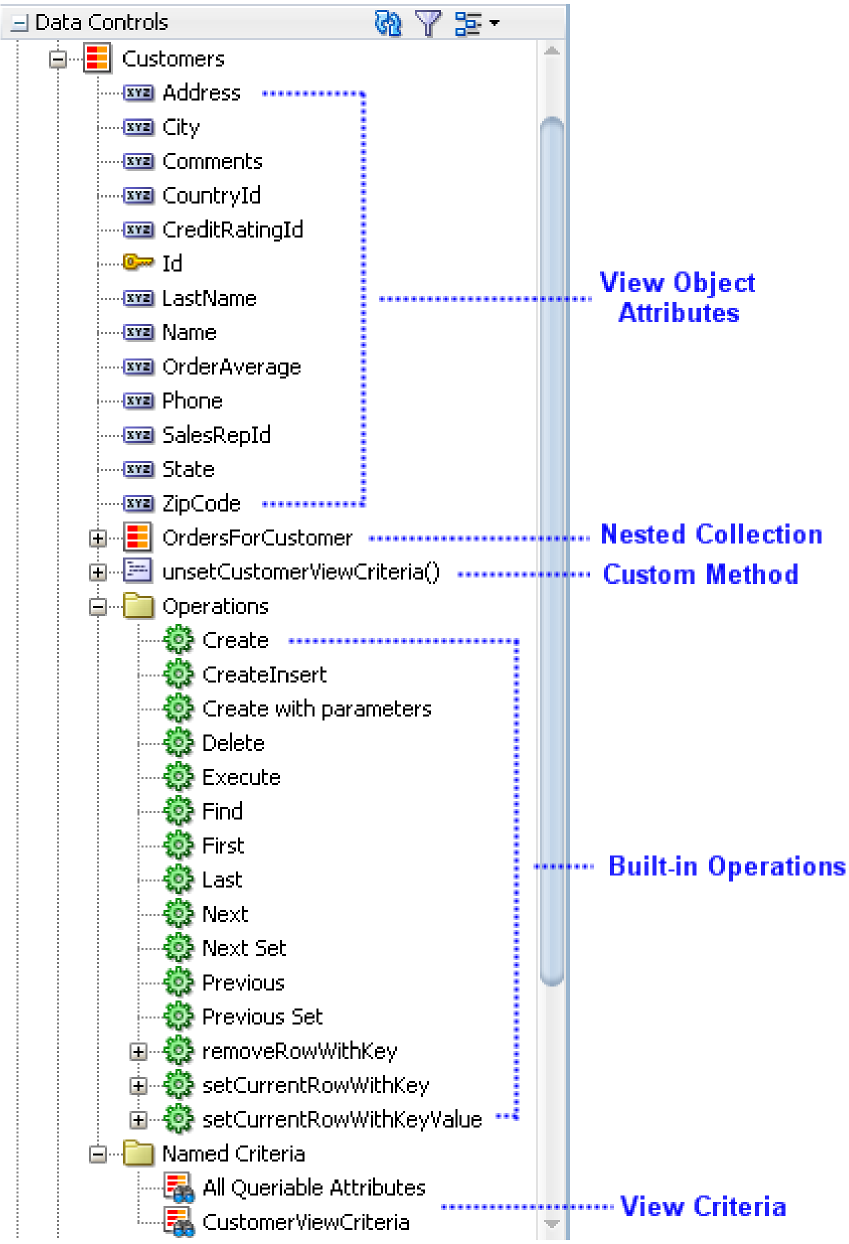


How View Objects Appear in the Data Controls Panel

The view object attributes are displayed as immediate child nodes of the corresponding data collection, as are any custom methods you've created. Figure 18-6 shows how each view object instance in the application module's data model appears in the Data Controls panel. If you have selected any custom methods to appear on the view object's client interface, they appear as custom methods immediately following the view object attributes at the same level. If the method accepts arguments, these appear in a nested Parameters node as operation parameters.

By default, implicit **view criteria** are created for each attribute that is able to be queried on a view object. They are represented by the All Queriable Attributes node under the Named Criteria node, as shown in Figure 18-6. If any named view criteria were created for the view object, they appear under the Named Criteria node. The conjunction used in the query, along with the criteria items and if applicable, any nested criteria, are shown as children. These items are used to create quick search forms, as detailed in Creating ADF Databound Search Forms .

Figure 18-6 How View Objects Appear in the Data Controls Panel



View Object Attributes

Nested Collection

Custom Method

Built-in Operations

View Criteria

As shown in Figure 18-6, the Operations node under the data collection displays all its available built-in operations. If an operation accepts one or more parameters, then those parameters appear in a nested Parameters node. At runtime, when one of these data collection operations is invoked by name by the data binding layer, the application module data control delegates the call to an appropriate method on the `ViewObject` interface to handle the built-in functionality. The built-in operations fall into three categories: operations that affect the current row, operations that refresh the data collection, and all other operations.

Operations that affect the current row:

- `Create`: Creates a new row that becomes the current row, but does not insert it.

- `CreateInsert`: Creates a new row that becomes the current row, and inserts the new blank row into the data source.
- `Create with Parameters`: Creates a new row taking parameter values. The passed parameters can supply the create-time value for the following:
  - A discriminator for a polymorphic view object
  - A composing parent's foreign key attribute needed for the creation of a polymorphic view object
  - A composed child view object row when it is not created in the context of a parent row

  For more information about polymorphic view objects, see Defining Polymorphic View Objects.
- `Delete`: Deletes the current row.
- `First`: Sets the current row to be the first row in the row set.
- `Last`: Sets the current row to be the last row in the row set.
- `Next`: Sets the row to be the next row in the row set.
- `Next Set`: Navigates forward one full set of rows.
- `Previous`: Sets the current row to be the previous row in the row set.
- `Previous Set`: Navigates backward one full set of rows.
- `setCurrentRowWithKey`: Tries to finds a row using the serialized string representation of row key passed as a parameter. If found, that row becomes the current row.
- `setCurrentRowWithKeyValue`: Tries to finds a row using the primary key attribute value passed as a parameter. If found, that row becomes the current row.

  For more information on using `setCurrentRowWithKey` and `setCurrentRowWithKeyValue`, see What You May Need to Know About Setting the Current Row in a Table.

Operations that refresh the data collection:

- `Execute`: Refreshes the data collection by executing or reexecuting the view object's query, leaving any bind parameters at their current values.
- `ExecuteWithParams`: Refreshes the data collection by first assigning new values to the named bind variables passed as parameters, then executing or reexecuting the view object's query.

> **Note:**
>
> The `executeWithParams` operation appears only for view objects that have defined one or more named bind variables at design time.

All other operations:

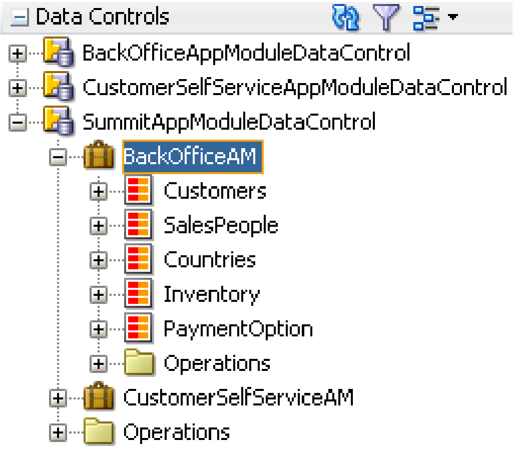
- `removeRowWithKey`: Tries to find a row using the serialized string representation of row key passed as a parameter. If found, the row is removed.

- Find: Toggles "Find Mode" on and off for the data collection.

How Nested Application Modules Appear in the Data Controls Panel

If you build composite application modules by including nested instances of other application modules, the Data Controls panel reflects this component assembly in the tree hierarchy. For example, the Summit ADF sample application contains the SummitAppModule application module, which contains nested instances of the BackOfficeAppModule and CustomerSelfServiceAppModule application modules. The nested instances are called BackOfficeAM and CustomerSelfServiceAM, respectively. As shown in Figure 18-7, there are top-level data controls for all three application modules, and the SummitAppModuleDataControl data control has subnodes for the BackOfficeAM and CustomerSelfServiceAM instances.

Figure 18-7 How Nested Application Modules Appear in the Data Controls Panel

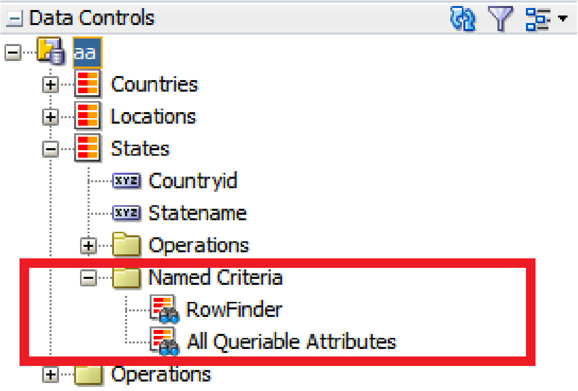


You need to be careful to perform your drag-and-drop data binding from the data control that corresponds to your intended usage. When you drop a data collection from the *top-level* BackOfficeAppModuleDataControl data control node in the panel, at runtime your page will use an instance of the BackOfficeAppModule application module acquired from a pool of BackOfficeAppModule components. When you drop a data collection from the BackOfficeAM *nested* instance of SummitAppModuleDataControl, at runtime your page will use an instance of the SummitAppModule application module acquired from a pool of SummitAppModule components. Since different types of application module data controls will have distinct transactions and database connections, inadvertently mixing and matching data collections from both a nested application module and a top-level data control will lead to unexpected runtime behavior.

How a Row Finder Appears in the Data Controls Panel

By default, any row finders that you define on a view object will appear in the Data Controls panel. At runtime, the row finder in the named criteria gets displayed as a search panel and displays only the attributes present in the named criteria and does not display any operators. The row-finder mapped attributes of a view object that the application exposes as a collection in the Data Controls panel are available in the row finder lookup operations. Figure 18-8 shows the Row Finder under Named Criteria in the Data Controls panel.

Figure 18-8 Row Finder in Data Controls Panel



When the application programmatically invokes the row finder on the view object instance, or when the end user supplies a value for a row-finder mapped attribute in the web service payload, the row finder locates the rows that match the values supplied to bind variables of the view criteria. The row finder does not alter the row set when locating matching rows.

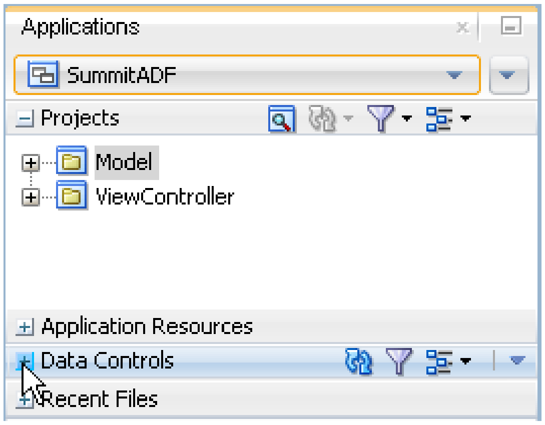
How to Open the Data Controls Panel

The Data Controls panel is a panel within the Applications window, located at the top left of JDeveloper. To view its contents, click the panel header to expand the panel. If you do not see the panel header, then the Applications window may not be displaying.

To open the Applications window and Data Controls panel:

1. From the main menu, choose **Window > Applications**.
2. To open the Data Controls accordion panel, click the expand icon in the Data Controls header, as shown in Figure 18-9.

Figure 18-9 Data Controls Panel in the Applications window



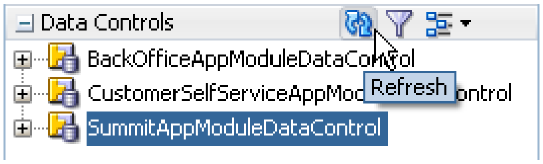
How to Refresh the Data Controls Panel

Any time changes are made to the application module or underlying services, you need to manually refresh the data control in order to view the changes.

To refresh the Data Controls panel:

Click the **Refresh** icon in the header of the Data Controls panel, as shown in Figure 18-10.

Figure 18-10 Refresh Icon on Data Controls Panel


Data Controls
BackOfficeAppModuleDataControl
CustomerSelfServiceAppMod
Refresh
ontrol
SummitAppModuleDataControl

When you click **Refresh**, the Data Controls panel looks for all available data controls, and therefore will now reflect any structural changes made to the data control.

Packaging a Data Control for Use in Another Project

You can package up data controls so that they can be used in another project. For example, one development group might be tasked with creating the services and data controls, while another development group might be tasked with creating the UI. The first group would create the services and data controls, and then package them up as an Oracle ADF Library and send it to the second group. The second group can then add the data controls to their project using the Resources window. For more information, see Reusing Application Components .

Using the Data Controls Panel

In ADF, all the components are listed in the Data Controls Panel, which helps you design a page easily. It lists the UI components, service methods, transaction control operations and also the nested application modules. All the code to bind the objects are automatically created during drag and drop operations.
You can design a databound user interface by dragging an item from the Data Controls panel and dropping it on a page as a specific UI component. When you use data controls to create a UI component, JDeveloper automatically creates the various code and objects needed to bind the component to the data control you selected.

In the Data Controls panel, each data control object is represented by a specific icon. Table 18-1 describes what each icon represents, where it appears in the Data Controls panel hierarchy, and what components it can be used to create.

Table 18-1 Data Controls Panel Icons and Object Hierarchy

Icon	Name	Description	Used to Create...
🗄	Data Control	Represents a data control. You cannot use the data control itself to create UI components, but you can use any of the child objects listed under it. Depending on how your business services were defined, there may be more than one data control. Usually, there is one data control for each application module. However, you may have additional data controls that were created for other types of business services (for example, for web services). For information about creating data controls for other business services, see Using ADF Data Controls in *Developing Applications with Oracle ADF Data Controls*.	Serves as a container for the other object and is not used to create anything.
▤	Collection	Represents a named data collection. A **data collection** represents a set of data objects (also known as a **row set**) in the data model. Each object in a data collection represents a specific structured data item (also known as a **row**) in the data model. Throughout this guide, *data collection* and *collection* are used interchangeably. For application modules, the data collection is the default row set contained in a view object instance. The name of the collection matches the view object instance name. A view link creates a **master-detail relationship** between two view objects. If you explicitly add an instance of a detail view object (resulting from a view link) to the application module data model, the collection contained in that detail view object appears as a child of the collection contained in the master view object. For information about adding detail view objects to the data model, see How to Enable Active Master-Detail Coordination in the Data Model. The children under a collection may be attributes of the collection, other collections that are related by a view link, custom methods that return a value from the collection, or built-in operations that can be performed on the collection. If you've configured JDeveloper to display view link accessor returns, then those are displayed as well.	Forms, tables, graphs, trees, range navigation components, and master-detail components. For information about using collections on a data control to create forms, see Creating a Basic Databound Page. For information about using collections to create tables, see Creating ADF Databound Tables . For information about using master-detail relationships to create UI components, see Displaying Master-Detail Data . For information about creating graphs, charts, and other visualization UI components, see Creating Databound Chart and Gauge Components.
XYZ	Attribute	Represents a discrete data element in an object (for example, an attribute in a row). Attributes appear as children under the collections or method returns to which they belong. For application module data controls, only attributes included in the corresponding view object are shown under a collection. If a view object joins one or more entity objects, that view object's collection will contain selected attributes from all of the underlying entity objects.	Label, text field, date, list of values, and selection list components. For information about using attributes to create fields on a page, see Creating Text Fields Using Data Control Attributes. For information about creating lists, see Creating Databound Selection Lists and Shuttles .

Table 18-1 (Cont.) Data Controls Panel Icons and Object Hierarchy

Icon	Name	Description	Used to Create...
▭	Structured Attribute	Represents a returned object that is neither a Java primitive type (represented as an attribute) nor a collection of any type. An example of a structured attribute would be a domain, which is a developer-created data type used to simplify application maintenance. For information about domains, see Creating Custom, Validated Data Types Using Domains.	Label, text field, date, list of values, and selection list components.
▤	Method	Represents an operation in the data control or one of its exposed structures that may accept parameters, perform some business logic and optionally return a single value, a structure, or a collection. In application module data controls, custom methods are defined in the application module itself and usually return either nothing or a single scalar value. For information about creating custom methods, see Implementing Business Services with Application Modules.	Command components For methods that accept parameters: command components and parameterized forms.
▤	Method Return	Represents an object that is returned by a custom method. The returned object can be a single value or a collection. If a custom method defined in an application module returns anything at all, it is usually a single scalar value. Application module methods do not need to return a set of data to the view layer, because displaying the latest changes to the data is handled by the view objects in the data model (For information, see What Happens at Runtime: How View Objects and Entity Objects Cooperate). However, custom methods in non-application module data controls (for example, a data control for a CSV file) can return collections to the view layer. A method return appears as a child under the method that returns it. The objects that appear as children under a method return can be attributes of the collection, other methods that perform actions related to the parent collection, or operations that can be performed on the parent collection.	The same components as for collections and attributes. When a single-value method return is dropped, the method is not invoked automatically by the framework. To invoke the method, you can drop the corresponding method as a button. If the form is part of a task flow, you can create a method activity to invoke the method. For information about executables, see Executable Binding Objects Defined in the Page Definition File.
⚙	Operation	Represents a built-in data control operation that performs actions on the parent object. Data control operations are located in an Operations node under collections or method returns, and also under the root data control node. The operations that are children of a particular collection or method return operate on those objects only, while operations under the data control node operate on all the objects in the data control. If an operation requires one or more parameters, they are listed in a Parameters node under the operation.	UI command components, such as buttons, links, and menus. For information, see Creating Command Components Using Data Control Operations, and Creating a Form to Edit an Existing Record.
🟩	Parameter	Represents a parameter value that is declared by the method or operation under which it appears. Parameters appear in the Parameters node under a method or operation.	Label, text, and selection list components.

Table 18-1 (Cont.) Data Controls Panel Icons and Object Hierarchy

Icon	Name	Description	Used to Create...
	Named criteria	Represents a query from which you can create a user search form. An All Queriable Attributes criteria is generated automatically for each accessor collection. This criteria can be used to create a search form where it is possible for the user to query based on any queriable attribute in the collection. You can create custom view criteria and add them to the Data Controls panel. See Working with Named View Criteria.	For information on creating search forms, seeCreating ADF Databound Search Forms .

How to Use the Data Controls Panel

JDeveloper provides you with a predefined set of UI components from which to choose for each data control item you can drop.

Before you begin:

It may be helpful to have an understanding of the different objects in the Data Controls panel. For more information, see Using the Data Controls Panel.

You will need to complete these tasks:

- Create an application module that contains instances of the view objects that you want in your data model, as described in Creating and Modifying an Application Module. Alternatively, if you are using another type of business service, create a data control for that business service as described in "Exposing Business Services with Data Controls" in *Developing Applications with Oracle ADF Data Controls*.
- Create a JSF page as described in Creating a Web Page.

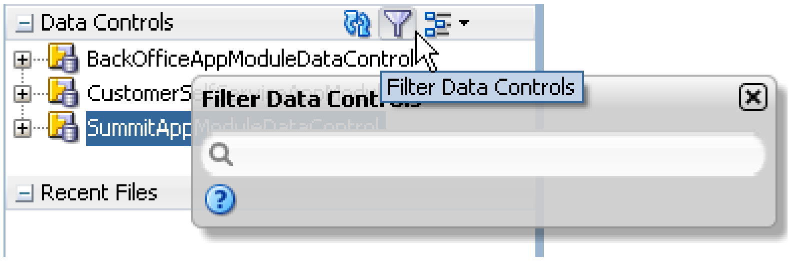
To use the Data Controls panel to create UI components:

1. Select an item in the Data Controls panel and drag it onto the visual editor for your page. For a definition of each item in the panel, see Table 18-1.

   If you need to drop an operation or method onto a method activity in a task flow, you can drag and drop it onto the activity in the task flow diagram.

   > **Tip:**
   >
   > You can use the Filter icon in the Data Controls Panel header to search for a specific item, as shown in Figure 18-11.

Figure 18-11 Filtering the Data Controls Panel

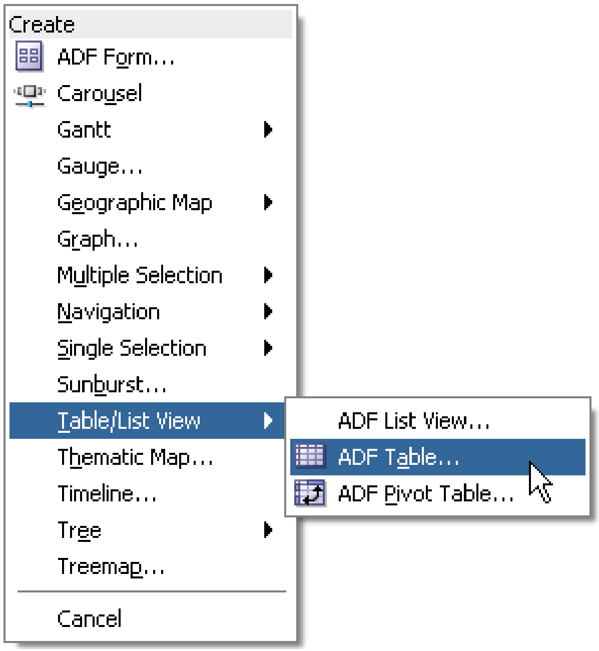


2. From the ensuing context menu, choose a UI component.

   When you drag an item from the Data Controls panel and drop it on a page, JDeveloper displays a context menu of all the default UI components available for the item you dropped. The components displayed are based on the libraries in your project.

   Figure 18-12 shows the context menu displayed when a data collection from the Data Controls panel is dropped on a page.

Figure 18-12 Data Controls Panel Context Menu

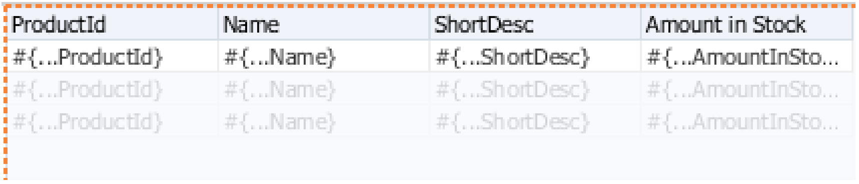


Depending on the component you select from the context menu, JDeveloper may display a dialog that enables you to define how you want the component to look. For example, if you select **Table/List View > ADF Table** from the context menu, the Create Table dialog launches. This dialog enables you to define which attributes you want to display in the table columns, what the column labels are, what types of UI components you want use for each column, and what functionality you want to include, such as row selection or column sorting. For more information about creating the various databound UI components, see the chapters in Creating a Databound Web User Interface .

The UI components selected by default are determined first by any UI control hints set on the corresponding business object. If no control hints have been set, then JDeveloper uses input components for standard forms and tables, and output components for read-only forms and tables. Components for lists are determined based on the type of list you chose when dropping the data control object.

Once you select a component, JDeveloper inserts the UI component on the page in the visual editor. For example, if you drag a collection from the Data Controls panel, choose **Table/List View > ADF Table** from the context menu, and select the read-only option, a read-only table appears in the visual editor, as shown in Figure 18-13.

Figure 18-13 Databound UI Component: ADF Read-Only Table



ProductId	Name	ShortDesc	Amount in Stock
#{...ProductId}	#{...Name}	#{...ShortDesc}	#{...AmountInSto...
#{...ProductId}	#{...Name}	#{...ShortDesc}	#{...AmountInSto...
#{...ProductId}	#{...Name}	#{...ShortDesc}	#{...AmountInSto...

By default, the UI components created when you use the Data Controls panel use ADF Faces components, are bound to attributes in the ADF data control, and may have one or more built-in features, including:

- Databound labels
- Tooltips
- Formatting
- Basic navigation buttons
- Validation, if validation rules are attached to a particular attribute. For more information, see Defining Validation and Business Rules Declaratively.

The default components are fully functional without any further modifications. However, you can modify them to suit your particular needs. Each component and its various features are discussed further in Creating a Databound Web User Interface .

> **Tip:**
>
> If you want to change the type of ADF databound component used on a page, the easiest method is to use either the visual editor or the Structure window to delete the component, and then drag and drop a new one from the Data Controls panel. When you use the visual editor or the Structure window to delete a databound component from a page, if the related binding objects in the page definition file are not referenced by any other component, JDeveloper automatically deletes those binding objects for you. Automatic deletion of binding objects will not happen if you use the source editor.

What Happens When You Use the Data Controls Panel

When an Oracle ADF web application is built using the JSF framework, it requires a few additional application object definitions to render and process a page containing ADF databound UI components. If you do not use the Data Controls panel, you will have to manually configure these various files yourself. However, when you use the Data Controls panel, JDeveloper does all of the following required steps:

- Creates a `DataBindings.cpx` file in the default package for the project (if one does not already exist), and adds an entry for the page.

  `DataBindings.cpx` files define the **binding context** for the application. The binding context is a container object that holds a list of available data controls and data binding objects. For more information, see What Happens at Runtime: How the Binding Context Works. Each `DataBindings.cpx` file maps individual pages to the binding definitions in the page definition file and registers the data controls used by those pages. For more information, see Working with the DataBindings.cpx File .

- Creates the `adfm.xml` file in the META-INF directory. This file creates a registry for the `DataBindings.cpx` file, which allows the application to locate it at runtime so that the binding context can be created.
- Registers the ADF binding filter in the `web.xml` file.

  The ADF binding filter preprocesses any HTTP requests that may require access to the binding context. For more information about the binding filter configuration, see Configuring the ADF Binding Filter.

- Adds any necessary libraries to the view project, such as the following:
  - ADF Faces Databinding Runtime
  - Oracle XML Parser v2
  - JDeveloper Runtime
  - ADF Model Runtime
  - BC4J Runtime
  - Oracle JDBC
  - Connection Manager
  - BC4J Oracle Domains
- Adds a page definition file (if one does not already exist for the page) to the page definition subpackage. The default subpackage is `view.pageDefs` in the `adfmsrc` directory.

> **Tip:**
>
> You can set the package configuration (such as name and location) in the ADF Model settings page of the Project Properties dialog (accessible by double-clicking the project node).

The page definition file (*pageName*`PageDef.xml`) defines the ADF binding container for each page in an application's view layer. The binding container provides runtime access to all the ADF binding objects for a page. In later chapters, you will see how the page definition files are used to define and edit the binding object definitions for specific UI components. For more information about the page definition file, see Working with Page Definition Files.

- Configures the page definition file, which includes adding definitions of the binding objects referenced by the page.
- Adds ADF Faces components to the JSF page.

  These prebuilt components include ADF data binding expression language (EL) expressions that reference the binding objects in the page definition file. For more information, see Creating ADF Data Binding EL Expressions.
- Adds all the libraries, files, and configuration elements required by ADF Faces components. For more information, see the "ADF Faces Configuration" appendix in *Developing Web User Interfaces with Oracle ADF Faces*.

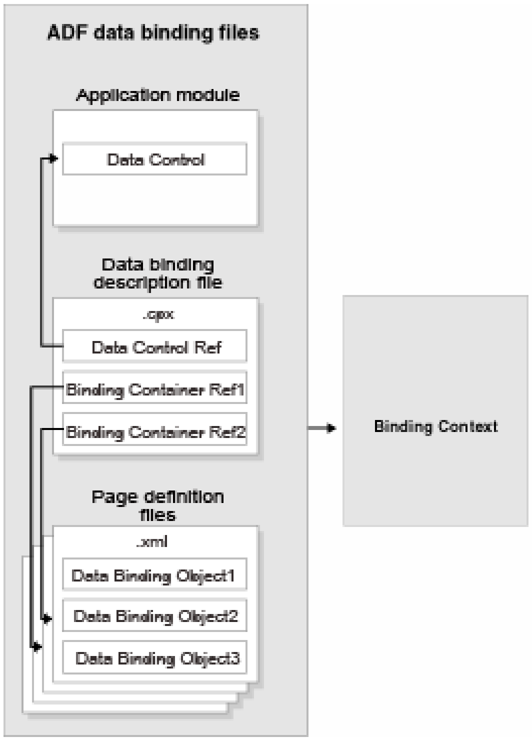
What Happens at Runtime: How the Binding Context Works

When a page contains ADF bindings, at runtime the interaction with the business services initiated from the client or controller is managed by the application through a single object known as the **binding context**. The binding context is a runtime map (named *data* and accessible through the EL expression `#{data}`) of all data controls and page definitions within the application.

The ADF lifecycle creates the Oracle ADF binding context from the application module, `DataBindings.cpx`, and page definition files, as shown in Figure 18-14. The union of all the `DataControls.dcx` files and any application modules in the workspace define the available data controls at design time, but the `DataBindings.cpx` file defines what data controls are available to the application at runtime. A `DataBindings.cpx` file lists all the data controls that are being used by pages in the application and maps the binding containers, which contain the binding objects defined in the page definition files, to web page URLs. The page definition files define the binding objects used by the application pages. There is one page definition file for each page.

The binding context does not contain real live instances of these objects. Instead, it is a map that contains references that become data control or binding container objects on demand. When the object (such as a page definition) is released from the application (for example when a task flow ends or when the binding container or data control is released at the end of the request), data controls and binding containers turn back into reference objects. For information about the ADF lifecycle, see Understanding the Fusion Page Lifecycle .

Figure 18-14   ADF File Binding Runtime Usage



> **Note:**
>
> Application module data controls also take advantage of **application module pooling**. The application module data control is a thin adapter over an application module pool that automatically acquires an available application module instance at the beginning of the request. During the current request, the application module data control holds a reference to the application module instance on behalf of the current user session. At the end of the request, the data control releases the instance back to the pool. For more information, see What Happens at Runtime: How the Application Uses Application Module Pooling and State Management.

Working with the DataBindings.cpx File

In ADF, you can find the metadata and the context for bindings in the DataBindings.cpx file. Depending on how an application is created, you can find one or multiple files. This file is available to the application at runtime.
The `DataBindings.cpx` files define the binding context for the entire application and provide the metadata from which the Oracle ADF binding objects are created at runtime. An application may have more than one `DataBindings.cpx` file if a component, for example a region, was created outside of the project and then imported. These files map individual pages to page definition files and declare which data controls are being used by the application. At runtime, only the data controls listed in the `DataBindings.cpx` files are available to the current application.

How JDeveloper Creates a DataBindings.cpx File

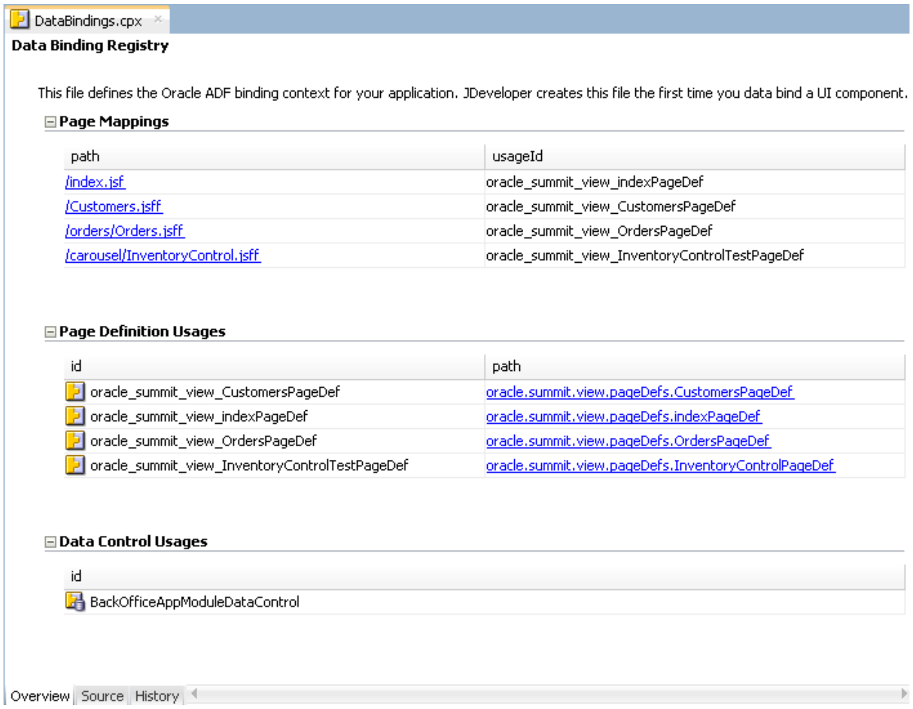
The first time you use the Data Controls panel to add a component to a page or an operation to an activity, JDeveloper automatically creates a `DataBindings.cpx` file in the default package of the view project. It resides in the `adfmsrc` directory for the project. Once the `DataBindings.cpx` file is created, JDeveloper adds an entry for the first page or task flow activity. Each subsequent time you use the Data Controls panel, JDeveloper adds an entry to the `DataBindings.cpx` for that page or activity, if one does not already exist.

> **Tip:**
>
> JDeveloper supports refactoring. That is, you can safely rename or move many of the objects referenced in the `DataBindings.cpx` file, and the references will be updated. For more information, see Refactoring a Fusion Web Application .

What Happens When JDeveloper Creates a DataBindings.cpx File

Once JDeveloper creates a `DataBindings.cpx` file, you can open it in the overview editor. Figure 18-15 shows the `DataBindings.cpx` file from the Summit ADF sample application, as viewed in the overview editor.

Figure 18-15 DataBindings.cpx File in the Overview Editor



Data Binding Registry

This file defines the Oracle ADF binding context for your application. JDeveloper creates this file the first time you data bind a UI component.

Page Mappings

path	usageId
/index.jsf	oracle_summit_view_indexPageDef
/Customers.jsff	oracle_summit_view_CustomersPageDef
/orders/Orders.jsff	oracle_summit_view_OrdersPageDef
/carousel/InventoryControl.jsff	oracle_summit_view_InventoryControlTestPageDef

Page Definition Usages

id	path
oracle_summit_view_CustomersPageDef	oracle.summit.view.pageDefs.CustomersPageDef
oracle_summit_view_indexPageDef	oracle.summit.view.pageDefs.indexPageDef
oracle_summit_view_OrdersPageDef	oracle.summit.view.pageDefs.OrdersPageDef
oracle_summit_view_InventoryControlTestPageDef	oracle.summit.view.pageDefs.InventoryControlPageDef

Data Control Usages

id
BackOfficeAppModuleDataControl

The following example shows the contents of the .cpx file in the Summit ADF sample application.

```
<?xml version="1.0" encoding="UTF-8" ?>
<Application xmlns="http://xmlns.oracle.com/adfm/application"
             version="11.1.1.56.60" id="DataBindings" SeparateXMLFiles="false"
             Package="oracle.summit.view" ClientType="Generic">
  <definitionFactories>
    <factory nameSpace="http://xmlns.oracle.com/adf/controller/binding"
             className="oracle.adf.controller.internal.binding.
                        TaskFlowBindingDefFactoryImpl"/>
    <dtfactory className="oracle.adf.controller.internal.dtrt.binding.
                          BindingDTObjectFactory"/>
    <factory nameSpace="http://xmlns.oracle.com/adfm/dvt"
             className="oracle.adfinternal.view.faces.dvt.model.binding.
                        FacesBindingFactory"/>
  </definitionFactories>
  <pageMap>
    <page path="/index.jsf" usageId="oracle_summit_view_indexPageDef"/>
    <page path="/Customers.jsff" usageId="oracle_summit_view_CustomersPageDef"/>
    <page path="/orders/Orders.jsff"
          usageId="oracle_summit_view_OrdersPageDef"/>
    <page path="/carousel/InventoryControl.jsff" usageId="oracle_summit_view_

InventoryControlTestPageDef"/>
  </pageMap>
  <pageDefinitionUsages>
    <page id="oracle_summit_view_CustomersPageDef"
          path="oracle.summit.view.pageDefs.CustomersPageDef"/>
    <page id="oracle_summit_view_indexPageDef"
          path="oracle.summit.view.pageDefs.indexPageDef"/>
    <page id="oracle_summit_view_OrdersPageDef"
          path="oracle.summit.view.pageDefs.OrdersPageDef"/>
    <page id="oracle_summit_view_InventoryControlTestPageDef"
          path="oracle.summit.view.pageDefs.InventoryControlPageDef"/>
  </pageDefinitionUsages>
  <dataControlUsages>
    <BC4JDataControl id="BackOfficeAppModuleDataControl"
                     Package="oracle.summit.model.services"
                     FactoryClass="oracle.adf.model.bc4j.DataControlFactoryImpl"
                     SupportsTransactions="true" SupportsFindMode="true"
                     SupportsRangesize="true" SupportsResetState="true"
                     SupportsSortCollection="true"
                     Configuration="BackOfficeAppModuleLocal"
                     syncMode="Immediate"
                     xmlns="http://xmlns.oracle.com/adfm/datacontrol"/>
  </dataControlUsages>
</Application>
```

The **Page Mappings** section of the editor maps each JSF page or task flow activity to its corresponding page definition file using an ID. The **Page Definition Usages** section maps the page definition ID to the absolute path for page definition file in the application. The **Data Control Usages** section identifies the data controls being used by the binding objects defined in the page definition files. These mappings allow the binding container to be initialized when the page is invoked.

You can use the overview editor to change the ID name for page definition files or data controls by double-clicking the current ID name and editing inline. Doing so will update all references in the application. Note, however, that JDeveloper updates only the ID name and not the file name. Be sure that you do not change a data control name to a reserved word. For more information, see How to Edit an Existing Application Module.

You can also click an element in the Structure window and then use the Properties window to change property values. For more information about the elements and attributes in the `DataBindings.cpx` file, see DataBindings.cpx.

Configuring the ADF Binding Filter

In ADF, when you add a databound object, JDeveloper automatically configures a filter which is a ADF binding filter that processes any HTTP request. This filter specifies the name of the binding objects and link filters to servlets in a web application.
The ADF binding filter is a servlet filter that is an instance of the `oracle.adf.model.servlet.ADFBindingFilter` class. ADF web applications use the ADF binding filter to preprocess any HTTP requests that may require access to the binding context. To do this, the ADF binding filter must be aware of all `DataBindings.cpx` files that exist for an application.

How JDeveloper Configures the ADF Binding Filter

The first time you add a databound component to a page using the Data Controls panel, JDeveloper automatically configures the filter for you in the application's `web.xml` file.

What Happens When JDeveloper Configures an ADF Binding Filter

To configure the binding filter, JDeveloper adds the following elements to the `web.xml` file:

- An ADF binding filter class: Specifies the name of the binding filter object, which implements the `javax.servlet.Filter` interface.

  The ADF binding filter is defined in the `web.xml` file, as shown in the following example. The `filter-name` element must contain the value `adfBindings`, and the `filter-class` element must contain the fully qualified name of the binding filter class, which is `oracle.adf.model.servlet.ADFBindingFilter`.

  ```
  <filter>
     <filter-name>adfBindings</filter-name>
     <filter-class>oracle.adf.model.servlet.ADFBindingFilter</filter-class>
  </filter>
  ```

- Filter mappings: Link filters to static resources or servlets in the web application.

  At runtime, when a mapped resource is requested, a filter is invoked. Filter mappings are defined in the `web.xml` file, as shown in the following example. The `filter-name` element must contain the value `adfBindings`.

  ```
  <filter-mapping>
    <filter-name>adfBindings</filter-name>
    <servlet-name>Faces Servlet</servlet-name>
    <dispatcher>FORWARD</dispatcher>
    <dispatcher>REQUEST</dispatcher>
  </filter-mapping>
  ```

> **Tip:**
>
> If you have multiple filters defined in the `web.xml` file, be sure to list them in the order in which you want them to run. At runtime, the filters are executed in the sequence in which they appear in the `web.xml` file.
>
> The `adfBindings` filter should appear after the Trinidad filter and before any filters that depend on the ADF context to be initialized.

What Happens at Runtime: How the ADF Binding Filter Works

At runtime, the ADF binding filter performs the following functions:

- Overrides the character encoding when the filter is initialized with the name specified as a filter parameter in the `web.xml` file. The parameter name of the filter `init-param` element is `encoding`.
- Instantiates the `ADFContext` object, which is the execution context for a Fusion web application and contains context information about ADF, including the security context and the environment class that contains the request and response object.
- Initializes the binding context for a user's HTTP session. To do this, it first loads the bindings as defined in the `DataBindings.cpx` file in the current project's `adfmsrc` directory. If the application contains `DataBindings.cpx` files that were imported from another project, those files are present in the application's class path. The filter additively loads any auxiliary `.cpx` files found in the class path of the application.
- Serializes incoming HTTP requests from the same browser (for example, from frame sets) to prevent multithreading problems.
- Notifies data control instances that they are about to receive a request, allowing them to do any necessary per-request setup.
- Notifies data control instances after the response has been sent to the client, allowing them to do any necessary per-request cleanup.

Working with Page Definition Files

In ADF, you add UI components on a page from the Data Controls panel. Then, JDeveloper creates a page definition file and adds definition for each binding object. The page definition file shows in the overview editor, which allows you to view and configure parameters, bindings and so on.
Page definition files define the binding objects that populate the data in UI components at runtime. For every page that has ADF bindings, there must be a corresponding page definition file that defines the binding objects used by that page. Page definition files provide design time access to all the ADF bindings. At runtime, the binding objects defined by a page definition file are instantiated in a binding container, which is the runtime instance of the page definition file.

Note:

When multiple windows are open to the same page, ADF Controller assigns each window its own `DataControlFrame`. This ensures that each window has its own binding container.

How JDeveloper Creates a Page Definition File

The first time you use the Data Controls panel to add a component to a page, JDeveloper automatically creates a page definition file for that page and adds definitions for each binding object referenced by the component. For each subsequent databound component you add to the page, JDeveloper automatically adds the necessary binding object definitions to the page definition file.

By default, the page definition files are located in the `view.PageDefs` package in the Application Sources node of the view project. If the corresponding JSF page is saved to a directory other than the default (`public_html`), or to a subdirectory of the default, then the page definition will also be saved to a package of the same name. For example, if you save your JSF file to the `public_html\myDirectory` directory, the page definition will be saved to the `myDirectory` package. You can change the location of the page definition files using the ADF Model Settings page of the Project Properties dialog.

JDeveloper names the page definition files using the following convention:

`pageName``PageDef.xml`

where *`pageName`* is the name of the JSF page or fragment. For example, if the JSF page is named `index.jsf`, the default page definition file name is `indexPageDef.xml`. If you organize your pages into subdirectories, JDeveloper prefixes the directory name to the page definition file name using the following convention:

`directoryName_pageName``PageDef.xml`

Tip:

Page definitions for task flows follow the same naming convention.

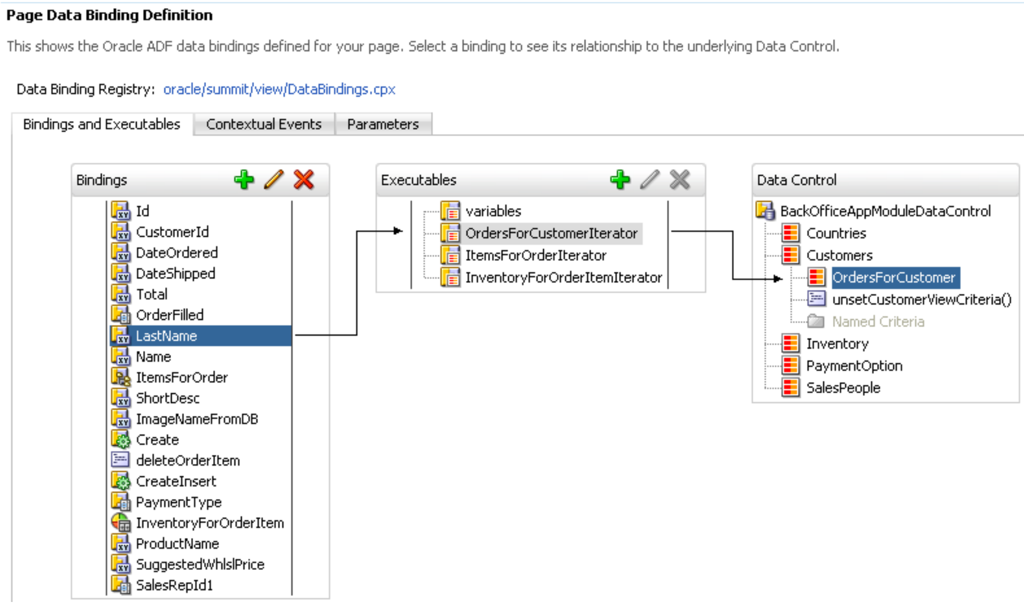
To open a page definition file, you can right-click directly on the page or activity in the visual editor, and choose **Go to Page Definition**, or for a JSF page, you can click the **Bindings** tab of the editor and click the Page Definition File link.

> **Tip:**
>
> While JDeveloper automatically creates a page definition for a JSF page when you create components using the Data Controls panel, or for a task flow when you drop an item onto an activity, it does not delete the page definition when you delete the associated JSF page or task flow activity. (This is to allow bindings to remain when they are needed without a JSF page, for example when using ADF Desktop Integration features.) If you no longer want the page definition, you need to delete the page definition and all references to it manually. Note however, that as long as a corresponding page or activity is never called, the page definition will never be used to create a binding context. It is therefore not imperative to remove any unused page definition files from your application.

What Happens When JDeveloper Creates a Page Definition File

When JDeveloper creates a page definition file, it is displayed in the overview editor. Figure 18-16 shows the page definition file in the overview editor that was created for the `Orders.jsff` page fragment in the Summit ADF sample application.

Figure 18-16 Page Definition File in the Overview Editor



Page Data Binding Definition

This shows the Oracle ADF data bindings defined for your page. Select a binding to see its relationship to the underlying Data Control.

Data Binding Registry: oracle/summit/view/DataBindings.cpx

The overview editor contains the following tabs, which allow you to view and configure bindings, contextual events, and parameters for a page:

- Bindings and Executables: The Bindings and Executables tab of the page definition overview editor shows three different types of objects: bindings, executables, and the associated data controls. (The data controls do not display unless you select a binding or executable.) For example, in Figure 18-16, you can see that the binding for the `LastName` attribute uses the `OrdersForCustomerIterator` iterator to get its value. The iterator accesses the `OrdersForCustomer` collection on the `BackOfficeAppModuleDataControl` data control. See Executable Binding Objects Defined in the Page Definition File.

By default, the model binding objects are named after the data control object that was used to create them. If a data control object is used more than once on a page, JDeveloper adds a number to the default binding object names to keep them unique. In Creating ADF Data Binding EL Expressions, you will see how the ADF data binding EL expressions reference the binding object names.

Table 18-2 shows the icons for each of the binding objects, as displayed in the overview editor (note that while parameter objects are shown in the **Parameter** section of the editor, they are also considered binding objects).

Table 18-2 Binding Object Icons

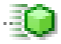
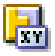
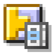
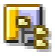
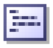
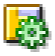
Binding Object Type	Icon	Description
Parameter		Represents a parameter binding object.
Bindings		Represents an attribute value binding object.
		Represents a list value binding object.
		Represents a tree value binding object.
		Represents a method action binding object
Bindings/ Executables		Represents an action binding object.

Table 18-2 (Cont.) Binding Object Icons

Binding Object Type	Icon	Description
Executables		Represents an iterator binding object.
		Represents a task flow executable binding object.
		Represents a search region binding object, which is used when named criteria objects are added to a page.

- Contextual Events: You can create contextual events that artifacts in an application can subscribe to. For example, you can use contextual events in a customer registration page to display the appropriate informational topic. One region in the page might contain a customer registration task flow, and the other an informational topic task flow. A contextual event can be passed from the customer registration region to the informational topic region so that the informational topic task flow can display the correct information topic. At design time, the event name, producer region, consumer region, consumer handler, and other information is stored in the event map section of the page definition file. See Using Contextual Events.
- Parameters: Parameter binding objects declare the parameters that the page evaluates at the beginning of a request. For more information about the ADF lifecycle, see Understanding the Fusion Page Lifecycle . Such parameters can also be passed through task flows, as described in Using Parameters in Task Flows.

  You can define the value of a parameter in the page definition file using static values, or EL expressions that assign a static value. The following example shows a parameter binding object that uses an EL expression to assign its value.

  ```
  <parameters>
  <parameter name="productId" value="${payLoad}"/>
  </parameters>
  ```

> **Tip:**
>
> The EL expression for the parameter values uses the dollar sign ($) because these expressions need to be resolved eagerly, so that the result is returned immediately, as the page is rendered. Most EL expressions in a JSF application use the hash sign (#), which defers the expression evaluation so that the model is prepared before the values are accessed.

When you click an item in the overview editor (or the associated node in the Structure window), you can use the Properties window to view and edit the attribute values for the item, or you can edit the XML source directly by clicking the **Source** tab. The following example shows XML code excerpts for the page definition file shown in Figure 18-16.

```
<?xml version="1.0" encoding="UTF-8" ?>
<pageDefinition xmlns="http://xmlns.oracle.com/adfm/uimodel"
                version="11.1.1.59.23" id="OrdersPageDef"
                Package="oracle.summit.view.pageDefs">
  <parameters/>
  <executables>
    <variableIterator id="variables"/>
    <iterator Binds="OrdersForCustomer" RangeSize="25"
              DataControl="BackOfficeAppModuleDataControl"
              id="OrdersForCustomerIterator" ChangeEventPolicy="ppr"/>
    <iterator Binds="ItemsForOrder" RangeSize="25"
              DataControl="BackOfficeAppModuleDataControl"
              id="ItemsForOrderIterator" ChangeEventPolicy="ppr"/>
    <iterator Binds="InventoryForOrderItem" RangeSize="-1"
              DataControl="BackOfficeAppModuleDataControl"
              id="InventoryForOrderItemIterator" ChangeEventPolicy="ppr"/>
  </executables>
  <bindings>
    <attributeValues IterBinding="OrdersForCustomerIterator" id="Id">
      <AttrNames>
        <Item Value="Id"/>
      </AttrNames>
    </attributeValues>
.
.
.
    <list IterBinding="OrdersForCustomerIterator" StaticList="false"
          Uses="LOV_OrderFilled" id="OrderFilled" DTSupportsMRU="false"/>
    <attributeValues IterBinding="OrdersForCustomerIterator" id="LastName">
      <AttrNames>
        <Item Value="LastName"/>
      </AttrNames>
    </attributeValues>
    <attributeValues IterBinding="OrdersForCustomerIterator" id="Name">
      <AttrNames>
        <Item Value="Name"/>
      </AttrNames>
    </attributeValues>
    <tree IterBinding="ItemsForOrderIterator" id="ItemsForOrder">
      <nodeDefinition DefName="oracle.summit.model.views.ItemVO"
                      Name="ItemsForOrder0">
        <AttrNames>
          <Item Value="ProductId"/>
          <Item Value="Name"/>
          <Item Value="Price"/>
          <Item Value="Quantity"/>
          <Item Value="ItemTotal"/>
        </AttrNames>
      </nodeDefinition>
    </tree>
.
.
.
  </bindings>
</pageDefinition>
```

In later chapters, you will see how the page definition file is used to define and edit the bindings for specific UI components. For a description of all the possible elements and attributes in the page definition file, see pageNamePageDef.xml.

Control Binding Objects Defined in the Page Definition File

There are three types of binding objects used to bind UI components to objects on the data control:

- Value: Displays data in UI components by referencing an iterator binding. Each discrete UI component on a page that will display data from the data control is bound to a value binding object. Types of value binding objects include:
  - Attribute Value: Binds text fields to a specific attribute in an object (also referred to as an **attribute binding** object.)
  - List: Binds the list items to all values of an attribute in a data collection.
  - Tree: Binds an entire table to a data collection and can also bind the root node of a tree to a data collection.
  - Button (boolean): Binds a checkbox to a boolean value for an attribute.
  - Graph: Binds a graph directly to the source data.
- Method Action: Binds command components, such as buttons or links, to custom methods on the data control. A method action binding object encapsulates the details about how to invoke a method and what parameters (if any) the method expects.
- Action: Binds command components, such as buttons or links, to built-in data control operations (such as, Commit or Rollback) or to built-in collection-level operations (such as, Create, Delete, Next, or Previous).

The following example shows an excerpt from a sample `bindings` element. Among other things, the element defines a tree binding for a table, several attribute bindings for text fields, an action binding for the `CreateInsert` built-in operation, and a list binding for a `selectOneChoice` component.

```
<bindings>
.
.
.
  <tree IterBinding="ItemsForOrderIterator" id="ItemsForOrder">
    <nodeDefinition DefName="oracle.summit.model.views.ItemVO"
                    Name="ItemsForOrder0">
      <AttrNames>
        <Item Value="ProductId"/>
        <Item Value="Name"/>
        <Item Value="Price"/>
        <Item Value="Quantity"/>
        <Item Value="ItemTotal"/>
      </AttrNames>
    </nodeDefinition>
  </tree>
  <attributeValues IterBinding="ItemsForOrderIterator" id="ShortDesc">
    <AttrNames>
      <Item Value="ShortDesc"/>
    </AttrNames>
  </attributeValues>
  <attributeValues IterBinding="ItemsForOrderIterator" id="ImageNameFromDB"
                   ChangeEventPolicy="ppr">
    <AttrNames>
      <Item Value="ImageNameFromDB"/>
    </AttrNames>
  </attributeValues>
.
.
.
  <action IterBinding="ItemsForOrderIterator" id="CreateInsert"
          RequiresUpdateModel="true" Action="createInsertRow"/>
  <list IterBinding="OrdersForCustomerIterator" StaticList="false"
        Uses="LOV_PaymentTypeId" id="PaymentType" DTSupportsMRU="true"/>
</bindings>
```

In the example, the tree element defines the bindings for the ItemsForOrder table. The AttrNames element is used to define the columns for which values will be displayed. The IterBinding attribute references the iterator binding that manages the data to be displayed in the table. For more information, see Executable Binding Objects Defined in the Page Definition File and Iterator and Value Bindings for Tables.

The attributeValues elements define the value bindings for text fields on the page. In the example, the ShortDesc attribute binding will display the value of ShortDesc, which is defined in the AttrNames element. The IterBinding attribute references the iterator binding that manages the data to be displayed in the text field. For more information, see What Happens When You Create a Text Field .

The binding object defined in the action element encapsulates the information needed to invoke the built-in CreateInsert operation on the ItemsForOrder collection. The value of true in the RequiresUpdateModel attribute specifies that the model layer needs to be updated before the operation is executed. For more information, see What Happens When You Create Command Components Using Operations.

If this operation also raised a contextual event, an event definition would also appear. If the page contained bindings that consumed an event, the event mapping would appear. For more information, see Using Contextual Events.

The PaymentType element defines the list binding used to display the list of payment type codes by using the LOV_PaymentTypeId LOV. For more information about creating lists using LOVs on view objects, see Creating Databound Selection Lists and Shuttles .

Executable Binding Objects Defined in the Page Definition File

There are the following types of executable binding objects:

- Iterator: Binds to an iterator that iterates over view object collections. There is one iterator binding for each collection used on the page. All of the value bindings on the page must refer to an iterator binding in order for the component values to be populated with data at runtime.

When you drop a collection or an attribute of a collection on the page, an iterator binding is automatically added as an executable. Iterator binding objects bind to an underlying ADF `RowSetIterator` object, which manages the current object and current range information. The iterator binding exposes the current object and range state to the other binding objects used by the page.

By default, iterator bindings are configured so that any submitted data changes are cached until the data is committed back to the data source. When a data change is submitted, any components on the page whose bindings are associated with the iterator are refreshed to show the changed data. For more information, see What You May Need to Know About Partial Page Rendering and Iterator Bindings.

The iterator **range** represents the current set of objects to be displayed on the page. The maximum number of objects in the current range is defined in the `rangeSize` attribute of the iterator. For example, if a collection in the data control contains products and the iterator range size is 25, the first 25 products in the collection are displayed on the page. If the user scrolls down, the next set of 25 is displayed, and so on. If the user scrolls up, the previous set of 25 is displayed. If your view object uses range paging, then you can configure the iterator binding to return a set of ranges at one time. For more information, see Using Range Paging to Efficiently Scroll Through Large Result Sets.

> **Note:**
>
> If you have two pages each with an iterator binding bound to the iterator on the same view object (which you will if you drop the same collection, for example, on two different pages), then you should ensure that the `rangeSize` attribute is the same for both pages' iterator bindings. If not, the page with a smaller range size may cause the iterator to reexecute, causing unexpected results on the other page.

- Method Iterator: Binds to an iterator that iterates over the collections returned by custom methods in the data control.

  A method iterator binding is always related to a method action binding object. The method action binding encapsulates the details about how to invoke the method and what parameters (if any) the method is expecting. The method action binding is itself bound to the method iterator, which provides the data.

  You will see method iterator executable binding objects only if you drop a method return collection or an attribute of a method return collection from a custom method on the data control. If you are using only application module data controls, you will see only iterator binding objects.

- Variable Iterator: Binds to an iterator that exposes all the variables in the binding container to the other bindings. While there is an iterator binding for each collection, there is only one variable iterator binding for all variables used on the page. (The variable iterator is like an iterator pointing to a collection that contains only one data object whose attributes are the binding container variables.)

  Page variables are local to the binding container and exist only while the binding container object exists. When you use a data control method (or an operation) that requires a parameter that is to be collected from the page, JDeveloper automatically defines a variable for the parameter in the page definition file. Attribute bindings can reference the page variables.

A variable iterator can contain one of two types of variables: `variable` and `variableUsage`. A `variable` type variable is a simple value holder, while a `variableUsage` type variable is a value holder that is related to a view object's named bind parameter. Defining a variable as a `variableUsage` type allows it to inherit the default value and UI control hints from the view object named bind variable to which it is bound.

- Page: Binds to the template's page definition file (if a template is used). For more information about how this works with templates, see Using Page Templates.

> **Note:**
>
> You can also use the `page` element to bind to another page definition file. However, at runtime, only the current incoming page's (or if the rendered page is different from the incoming, the rendered page's) binding container is automatically prepared by the framework during the current request. Therefore, to successfully access a bound value in another page from the current page, you must programmatically prepare that page's binding container in the current request (for example, using a backing bean). Otherwise, the bound values in that page may not be available or valid in the current request.

- Search Region: Binds named criteria to the iterator, so that the search can be executed.
- Task Flow: Instantiates the binding container for a region's task flow.
- Multi Task Flow: Instantiates the binding container for a region's task flow from an array of task flows. This is useful when you have a page that contains an unknown number of regions, such as a `panelTabbed` component where each tab is a region, and users can add and delete tabs at runtime. For more information, see Configuring a Page To Render an Unknown Number of Regions.

At runtime, executable bindings are refreshed based on the value of their `Refresh` attribute. Refreshing an iterator binding reconnects it with its underlying `RowSetIterator` object. Before refreshing any bindings, the ADF runtime evaluates any `Refresh` and `RefreshCondition` attributes specified in the executables. The `Refresh` attribute specifies the ADF lifecycle phase within which the executable should be invoked. The `RefreshCondition` attribute specifies the conditions under which the executable should be invoked. You can specify the `RefreshCondition` value using a boolean EL expression. If you leave the `RefreshCondition` attribute blank, it evaluates to `true`.

By default, the `Refresh` value is set to `deferred`. This means the binding will not be executed unless its value is accessed (for example by an EL expression on a JSF page). Once called, it will not reexecute unless any parameter values for the binding have changed, or if the binding itself has changed.

For more information about how bindings are refreshed and how to set the `Refresh` and `RefreshCondition` attributes, see About the JSF and ADF Page Lifecycles.

The following example illustrates executable binding objects from the Summit ADF sample application.

```
<executables>
  <variableIterator id="variables"/>
  <iterator Binds="OrdersForCustomer" RangeSize="25"
            DataControl="BackOfficeAppModuleDataControl"
            id="OrdersForCustomerIterator" ChangeEventPolicy="ppr"/>
  <iterator Binds="ItemsForOrder" RangeSize="25"
            DataControl="BackOfficeAppModuleDataControl"
            id="ItemsForOrderIterator" ChangeEventPolicy="ppr"/>
  <iterator Binds="InventoryForOrderItem" RangeSize="-1"
            DataControl="BackOfficeAppModuleDataControl"
            id="InventoryForOrderItemIterator" ChangeEventPolicy="ppr"/>
</executables>
```

In the example, the iterator binding named `OrdersForCustomerIterator` was created by dropping the `OrdersForCustomer` collection on the page as an ADF form. The iterator binding named `ItemsForOrderIterator` was created by dropping the `ItemsForOrder` collection as a table. The `InventoryForOrderItemIterator` iterator binding was created by dropping the `InventoryForOrderItem` collection as a graph.

The `OrdersForCustomer` collection has a master-detail relationship with the `ItemsForOrder` collection, so the two iterators are automatically synchronized. For more information, see What Happens at Runtime: ADF Iterator for Master-Detail Tables and Forms.

The `Binds` attribute of the `iterator` element defines the collection the iterator will iterate over. The `RangeSize` attribute defines the number of objects the iterator is to display on the page at one time. A `RangeSize` value of `-1` causes the iterator to display *all* the objects from the collection.

> **Tip:**
>
> Normally, an iterator binding's default range size is 25. However, when an iterator binding is created from the Edit List Binding dialog, the range size defaults to `-1` so that all choices display in the list, not just the first 25.

> **Performance Tip:**
>
> When you want to reduce the number of roundtrips the iterator requires to fetch the data objects from the view object in the ADF Business Components layer, you can set the `rangeSize` attribute to `-1`, and the objects will be fetched in a single round trip to the server, rather than in multiple trips as the user navigates through the objects.

Creating ADF Data Binding EL Expressions

In ADF, when you create a component, JDeveloper creates data binding expressions for every component. You can either edit these expressions or create your own expression to bind the component by using the Expression Builder.
To display data from the data model, web page UI components are bound to binding objects using JSF Expression Language (EL) expressions. These EL expressions reference a specific binding object in a binding container. At runtime, the JSF runtime

evaluates an EL expression and pulls the value from the binding object to populate the component with data when the page is displayed. If the user updates data in the UI component, the JSF runtime pushes the value back into the corresponding binding object based on the same EL expression.

> **Tip:**
>
> There may be cases when you need to use EL expressions within managed beans. For information on working with EL expressions within managed beans, see the "Creating EL Expressions" section in *Developing Web User Interfaces with Oracle ADF Faces*.

How to Create an ADF Data Binding EL Expression

When you use the Data Controls panel to create a component, the ADF data binding expressions are created for you. The expressions are added to every component attribute that will either display data from or reference properties of a binding object. Each prebuilt expression references the appropriate binding objects defined in the page definition file. You can edit these binding expressions or create your own, as long as you adhere to the basic ADF binding expression syntax. ADF data binding expressions can be added to any component attribute that you want to populate with data from a binding object.

In JSF pages, a typical ADF data binding EL expression uses the following syntax to reference any of the different types of binding objects in the binding container:

```
#{bindings.BindingObject.propertyName}
```

where:

- `bindings` is a variable that identifies that the binding object being referenced by the expression is located in the binding container of the current page. All ADF data binding EL expressions must start with the `bindings` variable.
- *`BindingObject`* is the ID, or for attributes the name, of the binding object as it is defined in the page definition file. The binding `objectID` or name is unique to that page definition file. An EL expression can reference any binding object in the page definition file, including parameters, executables, or value bindings.
- *`propertyName`* is a variable that determines the default display characteristics of each databound UI component and sets properties for the binding object at runtime. There are different binding properties for each type of binding object. For more information about binding properties, see What You May Need to Know About ADF Binding Properties.

For example, in the following expression that might appear on a JSF page:

```
#{bindings.ProductName.inputValue}
```

the `bindings` variable references a bound value in the current page's binding container. The binding object being referenced is `ProductName`, which is an attribute binding object. The binding property is `inputValue`, which returns the value of the first `ProductName` attribute.

Tip:

While the binding expressions in the page definition file can use either a dollar sign ($) or hash sign (#) prefix, the EL expressions in JSF pages can only use the hash sign (#) prefix.

As stated previously, when you use the Data Controls panel to create UI components, these expressions are built for you. However, you can also manually create them if you need to. The JDeveloper Expression Builder is a dialog that helps you build EL expressions by providing lists of binding objects defined in the page definition files, as well as other valid objects to which a UI component may be bound. It is particularly useful when creating or editing ADF databound expressions because it provides a hierarchical list of ADF binding objects and their most commonly used properties. For information about binding properties, see What You May Need to Know About ADF Binding Properties.

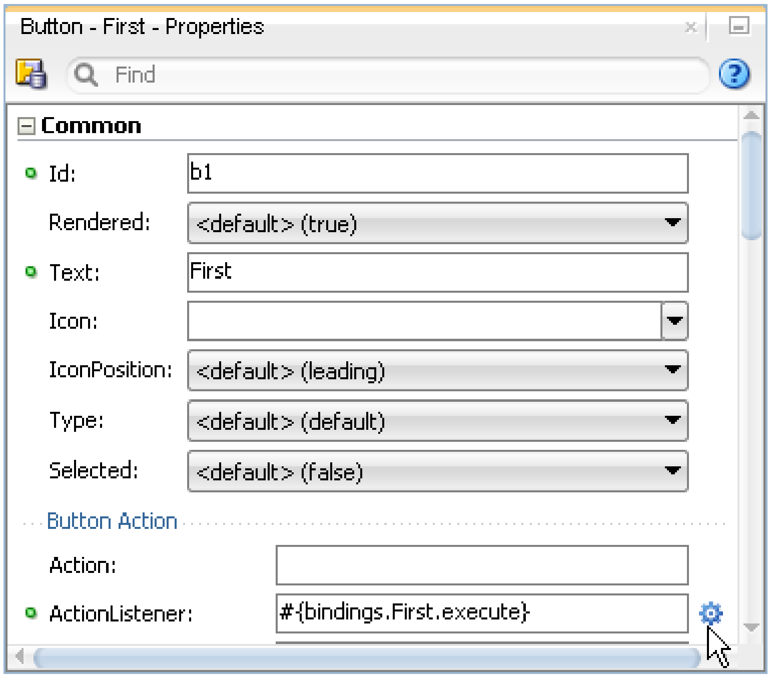
Opening the Expression Builder from the Properties Window

You can select an item in the visual editor, and then create EL expressions for specific attributes using the Properties window.

To open the Expression Builder from the Properties window:

1. Select a UI component in the Structure window or the visual editor.
2. In the Properties window, hover the mouse over the property and click the icon that appears to the right of the property, as shown in Figure 18-17.

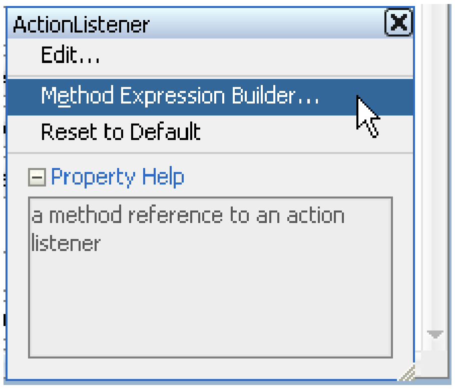
Figure 18-17 Icon for Property Menu



Id: b1
Rendered: <default> (true)
Text: First
Icon:
IconPosition: <default> (leading)
Type: <default> (default)
Selected: <default> (false)
Action:
ActionListener: #{bindings.First.execute}

3. In the popup menu, choose **Expression Builder** (or **Method Expression Builder**) as shown in Figure 18-18.

Figure 18-18 Property Menu



a method reference to an action listener

Using the Expression Builder

Once the Expression Builder is open, you can use it to create EL expressions.

Before you begin:

It may be helpful to have an understanding of how to create EL expressions when using the ADF Model layer. SeeCreating ADF Data Binding EL Expressions.

To use the Expression Builder:

1. Open the Expression Builder dialog as shown in Opening the Expression Builder from the Properties Window.
2. Use the Expression Builder to edit or create ADF binding expressions using the following features:
   - Use the **Variables** tree to select items that you want to include in the binding expression. The tree contains a hierarchical representation of the binding objects. Each icon in the tree represents various types of binding objects that you can use in an expression (see Table 18-3 for a description of each icon).

     To narrow down the tree, you can either use the dropdown filter or enter search criteria in the search field. Double-click an item in the tree to move it to the **Expression** box.
   - Use the operator buttons to add logical or mathematical operators to the expression.

> **Tip:**
>
> You can also type the expression directly in the **Expression** box.

Table 18-3 Icons Under the ADF Bindings Node of the Expression Builder

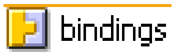
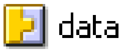
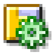
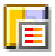
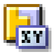
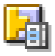
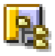
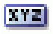
Icon	Description
	Represents the `bindings` container variable, which references the binding container of the current page. Opening the `bindings` node exposes all the binding objects for the current page.
	Represents the `data` binding variable, which references the entire binding context (created from all the `.cpx` files in the application). Opening the `data` node exposes all the page definition files in the application.
	Represents an action binding object. Opening a node that uses this icon exposes a list of valid action binding properties.
	Represents an iterator binding object. Opening a node that uses this icon exposes a list of valid iterator binding properties.
	Represents an attribute binding object. Opening a node that uses this icon exposes a list of valid attribute binding properties.
	Represents a list binding object. Opening a node that uses this icon exposes a list of valid list binding properties.
	Represents a table or tree binding object. Opening a node that uses this icon exposes a list of valid table and tree binding properties.
	Represents an ADF binding object property. For more information about ADF properties, see What You May Need to Know About ADF Binding Properties.

Table 18-3 (Cont.) Icons Under the ADF Bindings Node of the Expression Builder

Icon	Description
🟩	Represents a parameter binding object.
●	Represents a bean class.
☰	Represents a method.

What You May Need to Know About ADF Binding Properties

When you create a databound component using the Expression Builder, the EL expression might reference specific ADF binding properties. At runtime, these binding properties can define such things as the default display characteristics of a databound UI component or specific parameters for iterator bindings. The ADF binding properties are defined by Oracle APIs. For a full list of the available properties for each binding type, see Oracle ADF Binding Properties.

Values assigned to certain properties are defined in the page definition file. For example, iterator bindings have a property called `RangeSize`, which specifies the number of rows the iterator should display at one time. The value assigned to `RangeSize` is specified in the page definition file, as shown in the following example.

```
<iterator Binds="ItemsForOrder" RangeSize="25"
              DataControl="BackOfficeAppModuleDataControl"
              id="ItemsForOrderIterator" ChangeEventPolicy="ppr"/>
```

Using Simple UI First Development

In ADF, though you can use the Data Controls panel to design the page and bind objects, when your page uses declarative components, first add the UI components and then add the bindings, which can be reused as ADF Faces components.
While the Data Controls panel enables you to design and create bound components in a single drag-and-drop action, in some cases, it may be preferable to create the basic UI components first and add the bindings later. For example, if your page will use declarative components, you will first need to drop the declarative component, and then bind it to the correct ADF control. Declarative components are reusable, composite UI components that are made up of other ADF Faces components. Once imported into a project, declarative components can be dropped onto a page from the

Components window, similar to standard ADF Faces components. While the entire declarative component cannot use ADF data binding, you can use ADF data binding on the individual components that make up the declarative component, once the declarative component is dropped on the page. For more information about declarative components, see the "Using Declarative Components" section of *Developing Web User Interfaces with Oracle ADF Faces*.

> **Note:**
>
> If you know the UI components on your page will eventually use ADF data binding, but you need to develop the pages before the data controls are ready, then you should consider using placeholder data controls, rather than manually binding the components. Using placeholder data controls will provide the same declarative development experience as using developed data controls. For more information, see Designing a Page Using Placeholder Data Controls .

When designing web pages, keep in mind that ADF bindings can be added only to certain ADF Faces tags or their equivalent JSF HTML tags. Table 18-4 lists the ADF Faces and JSF tags to which you can later add ADF bindings.

> **Tip:**
>
> To enable the use of JSF Reference Implementation UI component tags with ADF bindings, you must choose the **Include JSF HTML Widgets for JSF Databinding** option in the **ADF View Settings** of the project properties. However, using ADF Faces tags, especially with ADF bindings, provides greater functionality than does using the reference implementation JSF tags.

Table 18-4   Tags That Can Be Used for ADF Bindings

ADF Faces Tags Used in ADF Bindings	Equivalent JSF HTML Tags
Text Fields	
`af:inputText`	`h:inputText`
`af:outputText`	`h:outputText`
`af:outputLabel`	`h:outputLabel`
`af:inputDate`	n/a
Tables	
`af:table`	`h:dataTable`
Actions	
`af:button`	`h:commandButton`
`af:link`	`h:commandLink`
`af:commandMenuItem`	n/a
Selection Lists	
`af:inputListOfValues`	n/a

Table 18-4   (Cont.) Tags That Can Be Used for ADF Bindings

ADF Faces Tags Used in ADF Bindings	Equivalent JSF HTML Tags
`af:selectOneChoice`	`h:selectOneMenu`
`af:selectOneListbox`	`h:selectOneListbox`
`af:selecOneRadio`	`h:selectOneRadio`
`af:selectBooleanCheckbox`	h:selectBooleanCheckbox
Queries	
`af:query`	n/a
`af:quickQuery`	n/a
Trees	
`af:tree`	n/a
`af:treeTable`	n/a

Before adding binding to the UI components, ensure that you follow these guidelines:

- When creating the JSF page using the Create JSF JSP wizard, choose the **Do not Automatically Expose UI Components in a Managed Bean** option on the **Managed Bean** tab.

  This option turns off JDeveloper's auto-binding feature, which automatically associates every UI component in the page to a corresponding property in the backing bean for eventual programmatic manipulation. If you use the auto-binding feature, you will have to remove the managed bean bindings later, after you have added the ADF bindings. The managed bean UI component property bindings do not affect the ADF bindings, but their presence may be confusing in the JSF code. For information about managed beans, see Using a Managed Bean in a Fusion Web Application.

- Add the ADF Faces tag libraries.

  While you can add ADF bindings to JSF components, the ADF Faces components provide greater functionality, especially when combined with ADF bindings.

How to Apply ADF Model Data Binding to Existing UI Components

You apply ADF model binding to components using the Structure window.

Before you begin:

It may be helpful to have an understanding of UI first development. For more information, see Using Simple UI First Development.

You will need to complete these tasks:

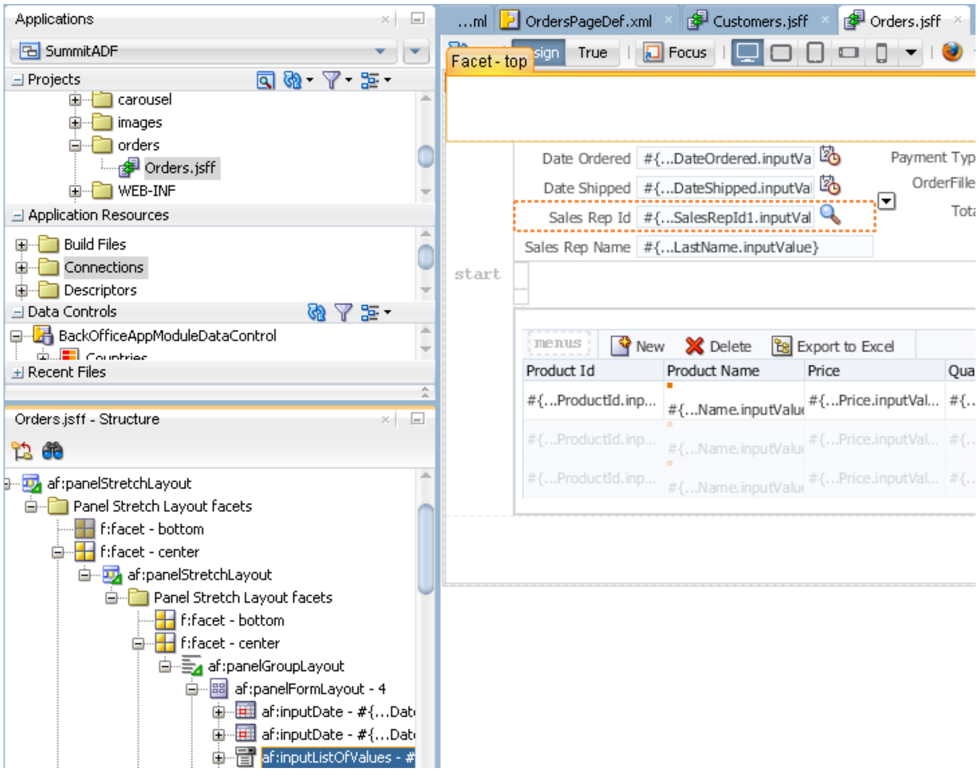
- Create an application module that contains instances of the view objects that you want in your data model, as described in Creating and Modifying an Application Module. Alternatively, if you are using another type of business service, create a data control for that business service as described in "Exposing Business Services with Data Controls" in *Developing Applications with Oracle ADF Data Controls*.
- Create a JSF page as described in Creating a Web Page, and add any components to the page.

To apply ADF Model data binding:

1. In the Design page of the visual editor, select the UI component to which you want to add ADF bindings.

   The component must be one of the tags listed in Table 18-4. When you select a component in the visual editor, JDeveloper simultaneously selects that component tag in the Structure window, as shown in Figure 18-19.

Figure 18-19 The Structure Window in JDeveloper



2. In the Structure window, right-click the UI component, and from the context menu, choose **Bind to ADF Control**.

> **Note:**
>
> Your project must already contain data controls for the **Bind to ADF Control** menu option to appear. If yours does not, you should consider using placeholder data controls, as described in Designing a Page Using Placeholder Data Controls .

3. In the Bind to ADF Control dialog, select the data control to which you want the UI component bound. JDeveloper will notify you if you choose a control that is not compatible with the selected UI component.

What Happens When You Apply ADF Model Data Binding to UI Components

When you use the Data Controls panel all of the required ADF objects are automatically created for you, as described in What Happens When You Use the Data Controls Panel.

19 Using Validation in the ADF Model Layer

This chapter describes how to use validation rules in the ADF model layer of your Oracle ADF application.

This chapter contains the following sections:

About ADF Model Layer Validation

In ADF, if you use data controls in an application that are not based on ADF Business Components, then you must use the validation rules to ensure that the data entered by user is validated.

In the model layer, ADF Model validation rules can be set for a binding's attribute on a particular page. When a user edits or enters data in a field and submits the form, the bound data is validated against any set rules and conditions. If validation fails, the application displays an error message.

Note that you don't need to add additional ADF Model validation if you have already set validation rules in the business domain layer of your entity objects. In a Fusion web application based on **ADF Business Components**, you won't need to use ADF Model validation unless you use data controls other than your **application module data control**.

ADF Model Layer Validation Use Cases and Examples

If your application uses data controls that are not based on ADF Business Components, you can use ADF Model layer validation to ensure the quality of user-entered data.

Best Practice:

Use business layer validation whenever possible. However, the following are examples of when you might need to use model layer validation:

- When your business layer does not support declarative validation.
- When you want to validate before an expensive roundtrip begins (for example, for a binding to a web service).
- When you need to validate a form created using parameters (for example, using the `executeWithParams` operation).

You can also use client-side validation for cases where you want to validate before a roundtrip to the server. For more information, see the "Adding Validation" section in *Developing Web User Interfaces with Oracle ADF Faces*.

Many of the declarative validation features available for ADF Business Components objects are also available at the model layer, should your application warrant the use of model-layer validation in addition to business-layer validation.

Additional Functionality for ADF Model Layer Validation

You may find it helpful to understand other ADF validation features before you use model layer validation. Following are links to other functionality that may be of interest.

- When you use the ADF Business Components application module data control, you do not need to use model-layer validation. Consider defining all or most of your validation rules in the centralized, reusable, and easier to maintain entity objects of your business layer. For more information, see Defining Validation and Business Rules Declaratively.
- Many of the declarative validation features available at the page level are also available at the bean level, which you would implement on the data control structure file. This can be very useful, because validation rules on the data control structure file apply to all usages of the data control. For information about implementing validation rules on the data control structure file, see the "Adding Business Logic to Data Controls" chapter in the *Developing Applications with Oracle ADF Data Controls*.

Defining Validation Rules in the ADF Model Layer

ADF provides a list of validation rules that you can use to validate binding attributes on the page definition file. You can use the Validation Rules Editor dialog to add, edit, or delete a validation rule for an attribute.
You can configure ADF Model validation for a binding's attributes on the page definition file. Validation rules in the model layer are executed for a binding's attribute on a particular page when the containing form is submitted.

You can optionally set the `skipValidation` property to `true` to bypass the ADF Model validation. You can set `skipValidation` to `skipDataControls` to validate the bound objects without validating the transaction. For example, set `skipValidation` to `skipDataControls` if you have a table action that opens a popup window to accept data entries and you want to allow the view layer to validate those entries before the commit on the table. The `skipValidation` property can be found in the Properties window after you have selected the root node of the page definition file in the Structure window.

How to Add ADF Model Layer Validation

You set ADF Model validation on the page definition file. You define the validation rule, and set an error message to display when the rule is broken.

Table 19-1 describes the ADF Model validation rules that you can configure for a binding's attributes.

Table 19-1 ADF Model Validation Rules

Validator Rule Name	Description
Compare	Compares the attribute's value with a literal value
List	Validates whether or not the value is in a list of values
Range	Validates whether or not the value is within a range of values
Length	Validates the value's character or byte size against a size and operand (such as greater than or equal to)
Regular Expression	Validates the data using Java regular expression syntax
Required	Validates whether or not a value exists for the attribute

Before you begin:

It may be helpful to have an understanding of the use of validation rules in the model layer. For more information, see Defining Validation Rules in the ADF Model Layer.

You may also find it helpful to understand additional functionality that can be added using other validation features. For more information, see Additional Functionality for ADF Model Layer Validation.

You will need to complete this task:

- Create a component on the page. The component must have binding attributes.

To create an ADF Model validation rule:

1. In the Applications window, double-click the page definition that contains the binding for which you want to create a rule.
2. In the Structure window, select the attribute, list, or table binding.
3. In the Properties window, click the **More** dropdown menu and choose **Edit Validation Rule**.
4. In the Validation Rules Editor dialog, expand the binding node, select the attribute name, and click **New**.
5. In the Add Validation Rule dialog, select a validation rule and configure the rule as needed.
6. Click the **Failure Handling** tab and configure the message to display when the rule is broken.

What Happens at Runtime: Model Validation Rules

When a user submits data, as long as the submitted value is a non-null value or a string value of at least one character, then all validators on a component are called one at a time. Because the `f:validator` tag on the component is bound to the `validator` property on the binding, any validation routines set on the model are accessed and executed.

The process then continues to the next component. If all validations are successful, the Update Model Values phase starts and a local value is used to update the model. If any validation fails, the current page is redisplayed along with an error message.

What You May Need to Know About Default Error Handling

When exceptions occur, the default behavior is to display only leaf-level errors in ADF Faces error dialogs. The default error handling mechanism for ADF Controller retrieves the list of exceptions from the page binding container and determines how to display each error. If the exception is not a `JboException`, a Faces error message is displayed with the exception message.

If the exception is a `JboException`, the error handling mechanism drills down recursively into nested exceptions, and displays a single Faces error message for each unique exception. However, if the exception has `JboWarning.HIDE_DETAIL_EXCEPTIONS_HINT`, the error handling mechanism will not drill down into nested exceptions.

Additionally, the error handling mechanism retrieves a list of binding errors for each exception and displays a Faces error message for the component associated with the binding that reported the error.

Customizing Error Handling

In ADF, you can use the ErrorHandlerClass property to set the error handler you want to use to report error or informational messages. You need not write code for error messages, but you can use the Properties windows of DataBindings.cpx file where you can set the error handler.
You can report errors using a custom error handler that extends the default `DCErrorHandlerImpl` class. You are not required to write any code to register your custom exception handler class. Instead, you select the root node of the `DataBindings.cpx` file in the Structure window, and then use the Properties window to set the `ErrorHandlerClass` property to the fully qualified name of the error handler you want it to use.

Your custom error handler can contain the following overridable methods:

- `reportException()`: Called to report any exception that occurs. It can be overridden to analyze reported exceptions.
- `getDisplayMessage()`: Returns the message that will be reported to JSF for each error that occurs. Returning `null` is the way your custom error handler signals that a given exception should not be reported to the client.
- `getDetailedDisplayMessage()`: Returns the detail portion of the message as a `String` object or HTML that will be reported to JSF for each error that occurs.

Returning `null` is the way your custom error handler signals that a given exception should not be reported to the client.

- `skipException()`: Returns a boolean depending on whether you want to display each item from the nested exception in the final error list displayed to the user. This method override lets you implement logic to check for specifics exception types and, based on the business scenario, determine whether to display it in the list.

The following example shows a custom error handler that extends the `DCErrorHandlerImpl` class and shows the override for the `skipException()` method that is needed to skip exceptions that should not appear in the list displayed to the user.

```
package view.controller.fwkext;

import java.sql.SQLIntegrityConstraintViolationException;

import java.util.ArrayList;
import java.util.List;

import oracle.adf.model.binding.DCBindingContainer;
import oracle.adf.model.binding.DCErrorHandlerImpl;

import oracle.jbo.CSMessageBundle;
import oracle.jbo.DMLConstraintException;
import oracle.jbo.JboException;

public class CustomErrorHandler extends DCErrorHandlerImpl {

   List<ExceptionMapper> exceptionMapperList = new ArrayList<ExceptionMapper>();

   public CustomErrorHandler() {
     super();
     exceptionMapperList.add(new DisableJboExceptionCodesMapper());
   }

   public void reportException(DCBindingContainer bc, Exception ex) {
     for (ExceptionMapper mapper : exceptionMapperList) {
       if (mapper.canMapException(ex)) {
         ex = mapper.mapException(ex);
       }
     }
     super.reportException(bc, ex);
   }

   /**
    * If an exception is a RowValException or a TxnValException and they
    * have nested exceptions, then do not display it. This example shows
    * an implementation that skips the SQLIntegrityConstraintViolationException
    * from displaying in the error final list displayed to the user.
    */
   @Override
   protected boolean skipException(Exception ex) {

      if (ex instanceof DMLConstraintException) {
            return false;
        } else if (ex instanceof SQLIntegrityConstraintViolationException) {
            return true;
        }
        return super.skipException(ex);
    }

}
```

Your error handler class must have a default constructor that matches the name of the custom class. For example, if you create a custom handler class `MyErrorHandler`, the exception error handler would invoke the default constructor, as shown in the following example. Following this convention allows dynamic invocation by the class name alone.

```
ErrorHandlerClass="viewcontroller.MyErrorHandler"
 public MyErrorHandler()
 {
   super();
 }
```

How to Customize the Detail Portion of a Message

If you plan to customize and use the detail portion of a message, you can create a custom error handler and implement the `getDetailedDisplayMessage` method to retrieve and process that message. The finalized message will be passed to the view layer to be integrated with other messages.

To customize the detail portion of a message:

1. Create a custom error handler class that extends the default `DCErrorHandlerImpl` class.
2. In that class, override the `getDetailedDisplayMessage` method that returns a `DCErrorMessage` object.

   The following example shows an implementation of the `getDetailedDisplayMessage` method in the custom error handler class.

   ```
   public final class MyErrorMessageHandler extends DCErrorHandlerImpl {
       public MyErrorMessageHandler (){
           super();
       }
       public DCErrorMessage getDetailedDisplayMessage(BindingContext ctx,
                                                       RegionBinding ctr,
                                                       Exception ex) {
           ...
           return new MyDCErrorMesssage(ctr, ex);
       }
   }
   ```

3. Create a custom class that implements the `DCErrorMessage` interface. The class must implement the `getHTMLText` method and the `getText` method.

   You will add code to the `getHTMLText` method to perform the actual processing, but you must also implement `getText` to satisfy the interface requirements.

4. In the `getHTMLText` implementation, add code to create and process the error message.

   `getHTMLText` of `getDetailedDisplayMessage` should return the finalized version of the error message as an HTML fragment that will be inserted into the HTML code of the page. For this reason, you should perform all necessary preprocessing on the text message before the message is returned by `getDetailedDisplayMessage`. For instance, you may want to retrieve the localized version of the message or change the right-to-left ordering of the message before it is returned.

The following example shows an implementation of this interface.

```
public final class MyDCErrorMesssage implements DCErrorMessage {
    RegionBinding m_regionBinding;
    Exception m_ex;
       public MyDCErrorMesssage(RegionBinding ctr, Exception ex) {
        super();
        this.m_regionBinding = ctr;
        this.m_ex = ex;
    }
    public String getText() {
        ...
        return "Message String";
    }
    public String getHTMLText() {
        ...
        /* Add code to process the message, including localization */
        /* and right-to-left directional requirements. */
        /* Return the message as the finalized HTML fragment.*/
        return "<html><b>error</b> message details</html>";
    }
}
```

To format the message using HTML tags, you must enclose the message within <html></html> tags, as shown in the example. Note that only the following HTML tags are allowed in error messages:

- <span>
- <b>
- <a>
- <i>
- <em>
- <br>
- <hr>
- <li>
- <ol>
- <ul>
- <p>
- <tt>
- <big>
- <small>
- <pre>

How to Display an Informational Message from a Custom Error Handler

You can use an error handler to display an informative message in response to a warning in an application using custom validation in the ADF Model layer. For applications that use model layer validation, you need to use a custom error handler at the application CPX level, for all errors and warnings.

Before you begin:

It may be helpful to have an understanding of how custom validation in the model layer works. For more information, see Customizing Error Handling.

You may also find it helpful to understand additional functionality that can be added using other validation features. For more information, see Additional Functionality for ADF Model Layer Validation.

You must also launch JDeveloper, and open the application to which you want to add the custom error handler.

To display an informational message

1. In the data model project, create a class that extends `JboWarning`. For example:

```
public class InformationalMessage extends JboWarning
{
    public static final String INFO_PROPERTY = "$InformationalMessage$INFO";
    public InformationalMessage(String message)
    {
        super(message, "", new Object[]{INFO_PROPERTY});
    }
}
```

2. In the application module implementation class, add a method and expose it as a client method. For example:

```
public void addInformationMessageTest()
  {
      addWarning
        (new InformationalMessage("Testing. This is an information
message"));
  }
```

3. In the view controller project, create a customer error handler class, as described in Customizing Error Handling.

4. Add the customer error handler you created to the data bindings file (`DataBindings.cpx`). For example:

```
<Application xmlns="http://xmlns.oracle.com/adfm/application"
             id="DataBindings" SeparateXMLFiles="false"
             Package="my.view" ClientType="Generic"
             ErrorHandlerClass="my.view.CustomErrorHandler">
  <pageMap>
    <page path="/Test.jspx" usageId="my_view_TestPageDef"/>
  </pageMap>
```

5. To test the method, you can drop the client application module method onto a `.jspx` page as a command button to invoke the information message.

   When you run the `.jspx` file and press the button, the informational message is displayed.

How to Write an Error Handler to Deal with Multiple Threads

Oracle ADF constructs an instance of the custom error handler for each `BindingContext` object that is created. Because Oracle ADF serializes simultaneous web requests from the same logical end-user session, multiple threads generally will not use the same error handler at the same time. However, to guarantee a thread-safe custom error handler, use the `setProperty()` API on `JboException`. This method stores in the exception objects themselves any hints you might need later during the phase when exceptions are translated to JSF `FacesMessage` objects for display.

20

Designing a Page Using Placeholder Data Controls

This chapter describes how to create and use placeholder data controls in an Oracle ADF application. It shows you how to create placeholder data types, including master-detail relationships. It also describes how to create and import sample data.

This chapter includes the following sections:

About Placeholder Data Controls

ADF provides placeholder data controls that you can use to support the declarative process and this does not require coding. When the real data controls are ready, you can rebind the UI components, which makes it easier when you design a complex page.

Application development is typically divided into two separate processes: technical implementation and user interface design. More often than not, they are undertaken by separate teams with very different skill sets. The two teams can work together either in a **data-first** approach or a **UI-first** approach, or with some overlap between the two. With either approach, the teams usually work together iteratively, refining the application with each cycle.

In a data-first approach, the model, or data control is built first. Then the designer creates the layout and page flow by dragging and dropping the data controls onto pages as UI components. The model data is automatically bound to the components. This approach requires the data model to be available before the designer can proceed.

In a UI-first approach, the designer creates the layout using components from the Components window. When the data controls do become available, UI components are then bound to them. With this approach, you should be able to see most of the layout and page flows to make a development evaluation. However, until the data controls are available and bound to components, the application may not fully convey the intent of its design. For instance, an application that has a **master-detail relationship** is best reviewed when there is actual data that dynamically drives that relationship.

Placeholder data controls are easy-to-create, yet fully functional, stand-in data controls that can efficiently speed up the design-development process. UI designers can use placeholder data controls to create page layouts and page flows without the need to have real data controls available. These placeholder controls can be loaded with sample data to realistically simulate application execution for design evaluations. When the real data controls are ready, the UI components can be easily rebound to complete the application.

Creating placeholder data controls is a purely declarative process and does not require coding. It does not require in-depth knowledge of the underlying model, data source technology, actual database schema, or any of the complex relationships in an actual production data control. Placeholder data controls do not require an existing data source or a database connection. You can define multiple data types with multiple attributes. You can also define nested master-detail hierarchies between data types. Placeholder data controls have the same built-in operations such as Execute, Next, and Create. An implicitly created named criteria item allows the user to create search forms as if **view objects** and **view criteria** were available.

Placeholder Data Controls Use Cases and Examples

For many complex applications, the UI design may actually drive the development of the model, or data source. In this UI-first scenario, having placeholder data controls with sample data is essential to properly model the behavior of the application. In some cases, even if production data controls are available, UI designers may opt to use placeholder data controls because of their flexibility and ease of use.

Placeholder data controls can be used in many situations. In addition to being used for design review and development, they can be used to develop realistic runtime mock-ups for usability studies, or for proof-of-concept requirements. They can be used to create demos when the data model is not yet ready.

Additional Functionality for Placeholder Data Controls

You may find it helpful to understand some data access features before you start working with placeholder data controls. Following are links to other functionality that may be of interest.

- After your initial design with placeholder data controls, when the final data controls are available, you can simply rebind the components. For more information about rebinding components, see Creating a Basic Databound Page and Creating ADF Databound Tables .
- A placeholder data type attribute can be configured to be a **list of values (LOV)**. For more information about LOVs, see Working with List of Values (LOV) in View Object Attributes.
- You can create master-detail relationships between placeholder data types, similar to the master-detail data collections in a standard data control. For more information on master-detail forms and tables, see Displaying Master-Detail Data .
- When you define placeholder data types in a master-detail hierarchy, JDeveloper creates **view links** that define that relationship. For more information about view links, see Working with Multiple Tables in a Master-Detail Hierarchy.
- Placeholder data controls can be used for the development of search forms. For more information about query search forms, see Creating ADF Databound Search Forms .
- You might want to package placeholder data controls into reusable components as ADF Library JARs. For more information about reusable components and the ADF Library, see Reusing Application Components .

Creating Placeholder Data Controls

ADF provides the menu to create Placeholder Data Controls for a project, which appears as a node in Data Controls panel with built-in operations node. You can then customize the Placeholder Data Controls, add them to your page, then rebind them to real data controls.
You add placeholder data controls to a project using the New Gallery. After the placeholder data control has been created, it appears as a node in the Data Controls panel. It has a different icon than do standard data controls. Instead of an Operations node, the placeholder data control has a Built-in Operations node. Although the Built-in Operations node contains Commit and Rollback operations, these operations do not perform commits or rollbacks because there is not an actual data source for the data.

When a data control is initially created, it does not have any data types associated with it. You will need to manually create the data types as described in section Creating Placeholder Data Types.

How to Create a Placeholder Data Control

Placeholder data controls are defined at the project level in JDeveloper. You must already have created a project before you can create placeholder data controls.

Before you begin:

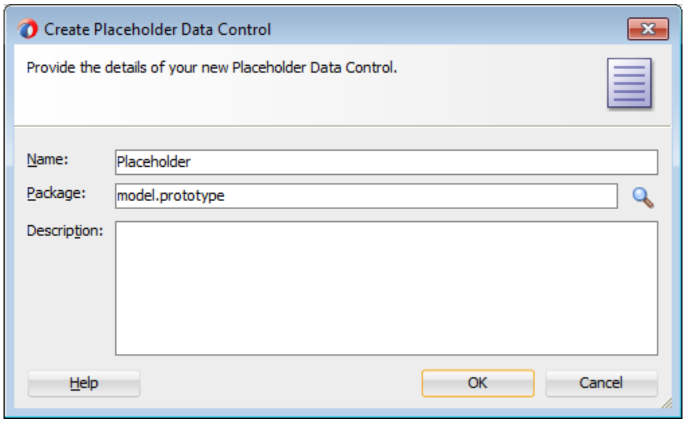
It may be helpful to have an understanding of the options you have for creating placeholder data controls. For more information, see Creating Placeholder Data Controls.

You may also find it helpful to understand additional functionality that can be added after using placeholder data controls. For more information, see Additional Functionality for Placeholder Data Controls.

To create a placeholder data control:

1. In the Applications window, right-click the project to which you want to add a placeholder data control and choose **New > From Gallery**.
2. In the New Gallery, expand **Business Tier**, select **Data Controls** and then **Placeholder Data Control**, and click **OK**.
3. In the Placeholder Data Control dialog, as shown in Figure 20-1, enter:
   - **Name**: The name of the placeholder data control.
   - **Package**: The package name that will be used to reference the placeholder data control.
   - **Description**: Optional description of the placeholder data control.

Figure 20-1 New Placeholder Data Control



Create Placeholder Data Control

Provide the details of your new Placeholder Data Control.

Name: Placeholder

Package: model.prototype

Description:

4. Click **OK**.

What Happens When You Create a Placeholder Data Control

When you create a placeholder data control, the package you selected to contain the data control appears under the project node in the Applications window. A data control XML file `*PlaceholderDataControl*.xml` appears under the package, where `*PlaceholderDataControl*` is the name of the placeholder data control. The following example shows a sample file called `Placeholder01.xml`, which was created when the `Placeholder01` data control was created.

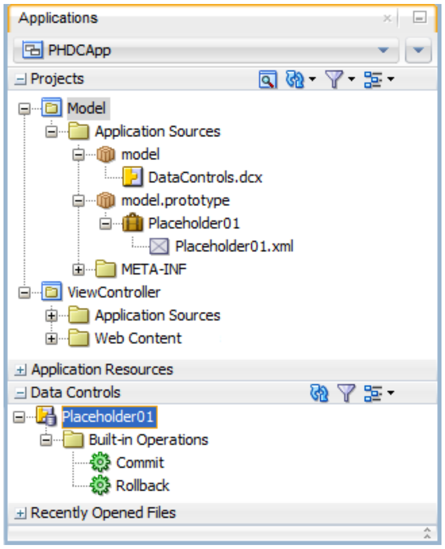
```
<?xml version='1.0' encoding='windows-1252' ?>
<AppModule
   xmlns="http://xmlns.oracle.com/placeholder"
   Name="Placeholder01" >
</AppModule>
```

JDeveloper also creates a `DataControls.dcx` file if it has not yet been defined, and adds entries for the placeholder data control, as shown in the following example.

```
<?xml version="1.0" encoding="UTF-8" ?>
<DataControlConfigs xmlns="http://xmlns.oracle.com/adfm/configuration"
                    version="12.1.2" id="DataControls"
                    Package="model">
  <PlaceholderDataControl SupportsTransactions="true" SupportsFindMode="true"
                          SupportsResetState="true" SupportsRangesize="true"
                          SupportsSortCollection="true"
                          FactoryClass=
                            "oracle.adf.model.placeholder.DataControlFactoryImpl"
                          id="Placeholder01"
                          xmlns="http://xmlns.oracle.com/adfm/datacontrol"
                          Definition="model.prototype.Placeholder01"
                          Package="model.prototype"/>
</DataControlConfigs>
```

In the Data Controls panel, the placeholder data control appears alongside other data controls in the root tree. A placeholder data control that does not yet have data types defined will have only the Commit and Rollback built-in operations available, as shown in Figure 20-2.

Figure 20-2 Applications Window and Data Controls Panel



Creating Placeholder Data Types

The placeholder data type is the data collection for the placeholder data control. ADF provides options to create the placeholder data type, add attributes from an exiting data type or from a CSV file, add sample data, and configure the placeholder data type.
A standard data control obtains its data collections and attributes from its underlying data source in the model or business service layer. For example, an **application module data control** obtains its data collections from the view objects and associated database tables.

For a placeholder data control, instead of data collections, it has placeholder data types. A **placeholder data type** is analogous to a data collection. It can be dropped onto a page to create complex components such as forms, tables, and trees. It also has a set of attributes that can be dropped onto pages as individual components such as input text, output text, and select choice. Some attributes may be defined as LOVs.

When you first create a placeholder data control, it is devoid of any data types because there are no underlying database tables for the placeholder data control to reference. You must declaratively create one or more placeholder data types. For each data type, you specify attribute names, types, default UI components, and other options. You can create multiple data types for a data control, similar to the multiple data collections in an application module.

After you have created a placeholder data type, it appears as a child node of the placeholder data control. It also has a Built-in operations node with the standard set of operations. It has a Named Criteria node that contains an All Queriable Attributes item that is analogous to the named view criteria of a view object in a standard data control. You can drag and drop the All Queriable Attributes item onto a page to create a query or quick query search form. In a standard data control, you can create multiple view criteria on a view object. Because there is no real view object in a placeholder data type, only one All Queriable Attributes item is available. For more information about query search forms, see Creating ADF Databound Search Forms .

You can create master-detail relationships between placeholder data types, similar to the master-detail data collections in a standard data control. You can drop master-detail data types onto pages to create master-detail forms and tables. For more information on master-detail forms and tables, see Displaying Master-Detail Data .

JDeveloper allows you to reuse placeholder data types created for other placeholder data controls in the same project. When you are creating a data type, you can select an option to load existing data types from another placeholder data control. If you select the **Import From: Existing Data Type** option, the attributes from the imported data type will be added to the list of attributes.

You can also select an option to import the sample data associated with the imported data type when the attributes are added.

> **Note:**
>
> Although you do not need sample data until you run the application, you should add sample data to provide a consistent design time rendering of the components.

How to Create a Placeholder Data Type

After you have created a placeholder data control, you can proceed to create data types. You define a name for the data type, and define each of its individual attributes. For each attribute, you then define its type, format, default UI component, and whether or not it should be an LOV.

In order to simplify the process of creating placeholder data types, you can select from a list of four of the most common types: `String`, `Boolean`, `Date`, and `Number`. Because placeholder attributes are typed, you can create column labels and include UI control hints in the design.

If you have a sample data file in comma-separated value (CSV) format, you can create all the attributes and load sample data using the sample data file import function. You do not need to create the attributes. JDeveloper will create them for you from the format of the CSV file, which can optionally contain a list of column headings as the first row. The attributes default to type `String`. You can manually reset each attribute to another type as required. For instructions to import sample data, see How to Import Attributes and Sample Data to a Placeholder Data Type.

Before you begin:

It may be helpful to have an understanding of the options you have for creating placeholder data types. For more information, see Creating Placeholder Data Types.

You may also find it helpful to understand additional functionality that can be added after using placeholder data controls. For more information, see Additional Functionality for Placeholder Data Controls.

Additionally, you must first create the placeholder data control, as described in How to Create a Placeholder Data Control.

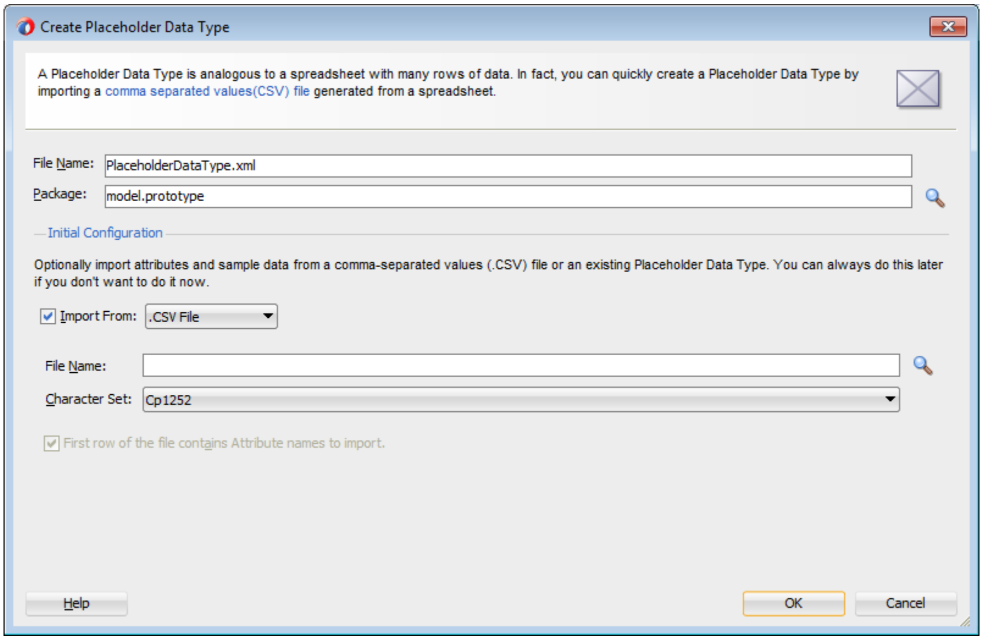
To create a placeholder data type:

1. In the Applications window, double-click the placeholder data control to which you want to add a placeholder data type.
2. In the overview editor, in the **Data Types** section, select the placeholder data control and click the **New Placeholder Data Type** icon.

   > **Note:**
   >
   > You can alternatively open the Create Placeholder Data Type dialog by right-clicking the placeholder data control in the Data Controls panel, and choosing **New Placeholder Data Type**.

   Figure 20-3 shows the Create Placeholder Data Type dialog.

   Figure 20-3 Create Placeholder Data Type Dialog



Create Placeholder Data Type

A Placeholder Data Type is analogous to a spreadsheet with many rows of data. In fact, you can quickly create a Placeholder Data Type by importing a comma separated values(CSV) file generated from a spreadsheet.

File Name: PlaceholderDataType.xml

Package: model.prototype

Initial Configuration

Optionally import attributes and sample data from a comma-separated values (.CSV) file or an existing Placeholder Data Type. You can always do this later if you don't want to do it now.

☑ Import From: .CSV File

File Name:

Character Set: Cp1252

☑ First row of the file contains Attribute names to import.

3. In the Create Placeholder Data Type dialog, enter a file name and package for the placeholder data type.
4. If you have an existing placeholder data type or a `.csv` file, from which you want to import attributes and data, select **Import From** and select either **.CSV File** or **Existing Data Type**, and then specify the options for the import.

   To import from a `.csv` file:

   a. In the **File Name** field, specify the file from which you want to import, or click the **Browse** icon to navigate to and select the file.
   b. From the **Character Set** dropdown list, select the character set that the file uses.
   c. If your file has a header row with the names of the columns, select **First row of the file contains Attribute names to import**.

   To import from an existing placeholder data type:

   a. Expand the tree to locate and select an existing placeholder data type.
   b. If the existing data type contains data that you want to import, select **Also Import Sample Data**.
5. Click **OK**.

   The placeholder data type is created and opened in the overview editor.
6. In the overview editor, click the **Attributes** navigation tab to view, add, and edit attributes for the placeholder data type.
7. Click the **Data** navigation tab to view, add, and edit sample data for the placeholder data type.

   You need sample data for runtime and for a consistent design time.

How to Add Attributes and Sample Data to a Placeholder Data Type

If you intend to run an application using the placeholder data control, you will need to add sample data to be displayed during execution. You can add sample data to the placeholder data type attributes manually or by importing the data from another placeholder data type or CSV file. Although having sample data is necessary only at runtime, you should add sample data for a consistent design time rendering of the components.

If you did not import attributes when you created your placeholder data type, you can add them using the Attributes page of the overview editor. Also, you can use the Data page of the overview editor to add sample data to be displayed during execution. Both the Attributes page and the Data page have an Import button that opens the Import Placeholder Data Type dialog, which allows you to import attributes and sample data.

Before you begin to add sample data to a placeholder data type, you should have already created a placeholder data control and a placeholder data type. If you are entering the data manually, you should have the data ready. If you are loading the data from a CSV file, you need to have the location of the file. If you are importing from another placeholder data type, you must have already created the other data type.

Manually Adding Attributes to a Placeholder Data Type

Using the Attributes page of the overview editor, you can add and configure attributes to a placeholder data type.

Before you begin:

It may be helpful to have an understanding of the options you have for creating placeholder data types. For more information, see Creating Placeholder Data Types.

You may also find it helpful to understand additional functionality that can be added after using placeholder data controls. For more information, see Additional Functionality for Placeholder Data Controls.

Additionally, you must first create the placeholder data control, as described in How to Create a Placeholder Data Control, and create the placeholder data type, as described in How to Create a Placeholder Data Type.

To manually add a placeholder data type attribute:

1. In the Applications window, double-click the placeholder data type to which you want to add attributes.

2. In the overview editor, click the **Attributes** navigation tab.
3. On the Attributes page, click the **Add** icon.
4. In the Insert View Attributes dialog, enter a name for the attribute, and click **OK**.
5. After adding attributes, select an attribute and use the fields beside the attributes list to configure the selected attribute.
   - **Name**: Enter a name for the attribute.
   - **Data Type**: Select a type for the attribute from the dropdown list. The supported types are `String`, `Boolean`, `Date`, and `Number`.
   - **Default Component**: Select a default component for the attribute from the dropdown list. For an LOV, select **Combo Box List of Values**. Then click the **Edit** icon to configure the LOV. For more information, see How to Configure a Placeholder Data Type Attribute to Be a List of Values.
   - **Default Value**: Enter the initial value for the attribute.
   - **Label**: Enter a label for the attribute. The label will be used when the component is displayed.
   - **Tooltip**: Enter text to be displayed as a tooltip for the attribute.
   - **Format Type**: This field is enabled only when the type is `Date` or `Number`. Select a format type from the dropdown list.
   - **Format**: This field is enabled only when a format mask has been defined for that format type.
   - **Display Width**: Enter the character width to be used by the control that displays this attribute.
   - **Searchable**: Select this checkbox to make the attribute searchable.

   Click the **Add** icon to add more attributes.

Adding Sample Data Manually

You can use the Sample Data page of the Edit Placeholder Data Type dialog to manually enter sample data for your placeholder data control.

Before you begin:

It may be helpful to have an understanding of the options you have for creating placeholder data types. For more information, see Creating Placeholder Data Types.

You may also find it helpful to understand additional functionality that can be added after using placeholder data controls. For more information, see Additional Functionality for Placeholder Data Controls.

Before you add sample data to a placeholder data type, you need to perform the following tasks:

- Create a placeholder data type, as described in How to Create a Placeholder Data Type.
- Have your list of sample data ready.

To add sample data to a placeholder data type manually:

1. In the Data Controls panel, right-click the placeholder data type to which you want to add sample data, and choose **Open**.

2. In the overview editor, click the **Data** navigation tab.
3. In the Data page, click the **Add** icon to add a row to the data table, and enter a value for each attribute. Repeat as necessary until you have entered adequate sample data.

How to Import Attributes and Sample Data to a Placeholder Data Type

As an alternative to manually entering attributes and sample data for a placeholder data type, you can import them in a single operation. Using the Attributes page of the overview editor, you can import attributes from an existing placeholder data type or from a `.csv` file.

A `.csv` (comma-separated values) file is a plain text file containing rows of attribute values, delimited by commas. A `.csv` file can optionally have a header row that defines the names of each attribute.

In the overview editor for a placeholder data type, both the Attributes page and the Data page have an **Import** button that opens the Import Placeholder Data Type dialog, which allows you to import attributes and sample data.

Before you begin:

It may be helpful to have an understanding of the options you have for creating placeholder data types. For more information, see Creating Placeholder Data Types.

You may also find it helpful to understand additional functionality that can be added after using placeholder data controls. For more information, see Additional Functionality for Placeholder Data Controls.

Additionally, you must first create the placeholder data control, as described in How to Create a Placeholder Data Control, and create the placeholder data type, as described in How to Create a Placeholder Data Type.

To import attributes into a placeholder data type:

1. In the Applications window, double-click the placeholder data type to which you want to add attributes.
2. In the overview editor, click the **Attributes** navigation tab.
3. If you are importing attributes as well as sample data, you must delete any existing attributes in the placeholder data type.

   If you do not remove the existing attributes, JDeveloper imports only the first columns of data for the number of defined attributes. For example, if you have two attributes defined in your data type and four columns of sample data in your CSV file, JDeveloper will import only the first two columns of data.
4. On the Attributes page, click **Import**.
5. In the Import Placeholder Data Type dialog, specify where you want to import attributes from.

   To use a `.csv` file:

   a. Select **Import from .CSV File**.
   b. In the **File Name** field, enter the full path and name of the file, or click the **Browse** icon to navigate to and select it.

c. From the **Character Set** dropdown list, select the character set represented in the specified `.csv` file.

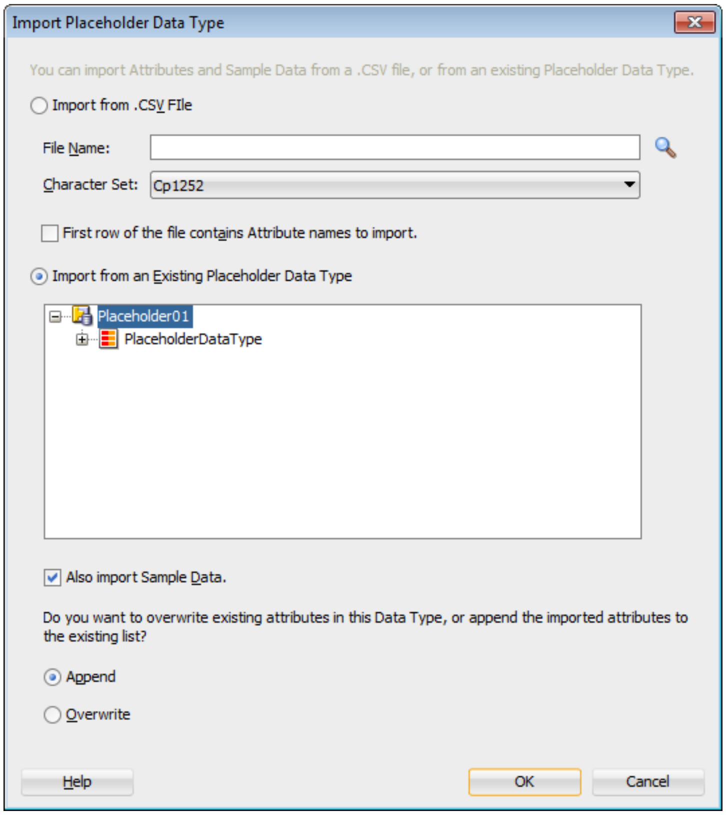
d. If the `.csv` file has a header row that contains the names of the attributes, select **First row of the file contains Attribute names to import**.

To use an existing placeholder data type:

a. Select **Import from an Existing Placeholder Data Type**.

b. Expand the available placeholder data controls displayed to see the placeholder data types they contain, and select the one you want to copy.

Figure 20-4 Import Placeholder Data Type Dialog



Import Placeholder Data Type

You can import Attributes and Sample Data from a .CSV file, or from an existing Placeholder Data Type.

○ Import from .CSV FIle

File Name:

Character Set: Cp1252

□ First row of the file contains Attribute names to import.

● Import from an Existing Placeholder Data Type

Placeholder01
PlaceholderDataType

☑ Also import Sample Data.

Do you want to overwrite existing attributes in this Data Type, or append the imported attributes to the existing list?

● Append

○ Overwrite

6. Select the **Also Import Sample Data** checkbox to load sample data from the file or existing placeholder data type.

7. Specify whether you would like to append or overwrite the attributes already defined for the placeholder data type.

   - Select **Append** to add the imported attributes to the current list of attributes in the placeholder data type
   - Select **Overwrite** to replace the current attributes with the imported attributes.

8. Click **OK**.

9. After importing attributes, you can optionally go to the Attributes page and reconfigure the attributes.

What Happens When You Add Sample Data

Placeholder sample data, whether added manually added or imported, are stored in message bundle files. The following example shows a sample message bundle with two rows of five attributes.

```
model.prototype.SupplierLocationsDataType.SL_0_0=101
model.prototype.SupplierLocationsDataType.SL_0_1=so
model.prototype.SupplierLocationsDataType.SL_0_2=Houston
model.prototype.SupplierLocationsDataType.SL_0_3=TX
model.prototype.SupplierLocationsDataType.SL_0_4=USA
model.prototype.SupplierLocationsDataType.SL_1_0=101
model.prototype.SupplierLocationsDataType.SL_1_1=dw
model.prototype.SupplierLocationsDataType.SL_1_2=Atlanta
model.prototype.SupplierLocationsDataType.SL_1_3=GA
model.prototype.SupplierLocationsDataType.SL_1_4=USA
```

What Happens When You Create a Placeholder Data Type

When you create a placeholder data type, JDeveloper creates a *PlaceholderDataType*.xml file, where *PlaceholderDataType* is the name of the placeholder data type you had specified.

The *PlaceholderDataType*.xml file has the same format as a view object XML file. It includes the name of the view object and the name and values of each placeholder attribute that was defined.

The following example shows a *PlaceholderDataType*.xml for a Supplier data type. Two attributes are declaratively defined: Supplier_Id and Supplier_Name.

```
<?xml version='1.0' encoding='windows-1252' ?>
<ViewObject
  xmlns="http://xmlns.oracle.com/placeholder"
  Name="SuppliersDataType"
  InheritPersonalization="merge"
  BindingStyle="OracleName"
  CustomQuery="true">
  <ViewAttribute
    Name="Supplier_Id"
    Type="oracle.jbo.domain.Number"/>
  <ViewAttribute
    Name="Supplier_Name"
    Type="java.lang.String"/>
</ViewObject>
```

Since a data type is similar to a data collection and is based on a view object, each data type will have a corresponding *PlaceholderDataType*.xml file.

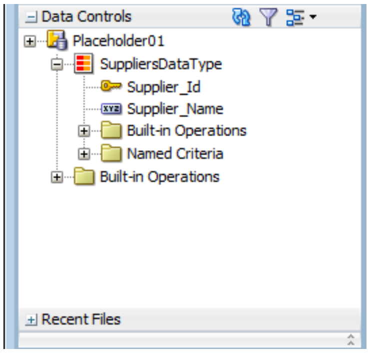
JDeveloper also adds entries for each placeholder data type to the *PlaceholderDataControl*.xml file. For example, after the Suppliers data type is created, the data control XML file includes a new ViewUsage entry for the Suppliers data type, as shown in the following example.

```
<?xml version="1.0" encoding="windows-1252" ?>
<AppModule
  xmlns="http://xmlns.oracle.com/placeholder"
  Name="Placeholder01"
  InheritPersonalization="merge">
  <ViewUsage
    Name="SuppliersDataType"
    ViewObjectName="model.prototype.SuppliersDataType"/>
</AppModule>
```

In the Data Controls panel, a placeholder data type node appears under the placeholder data control. Expanding the node reveals the presence of each of the attributes, the Built-in Operations node, and the Named Criteria node.

Figure 20-5 shows a placeholder data control as it appears in the Data Controls panel.

Figure 20-5 Data Controls Panel Showing Placeholder Data Control



How to Configure a Placeholder Data Type Attribute to Be a List of Values

A placeholder data type attribute can be configured to be a list of values (LOV). An LOV-formatted attribute binds to UI components that display dropdown lists or list picker dialogs. For more information about LOVs, see Working with List of Values (LOV) in View Object Attributes.

When you are configuring a placeholder data type attribute, you can select an option to bring up a dialog to configure that attribute to be an LOV.

If you have only one data source, you can only create a fixed LOV. To create a dynamic LOV, there must be more than one placeholder data type available to be the source.

Configuring an Attribute to Be a Fixed LOV

Before you begin, you should determine which attribute you want to be a fixed LOV and which values should be in the fixed list.

Before you begin:

It may be helpful to have an understanding of the options you have for creating placeholder data types. For more information, see Creating Placeholder Data Types.

You may also find it helpful to understand additional functionality that can be added after using placeholder data controls. For more information, see Additional Functionality for Placeholder Data Controls.

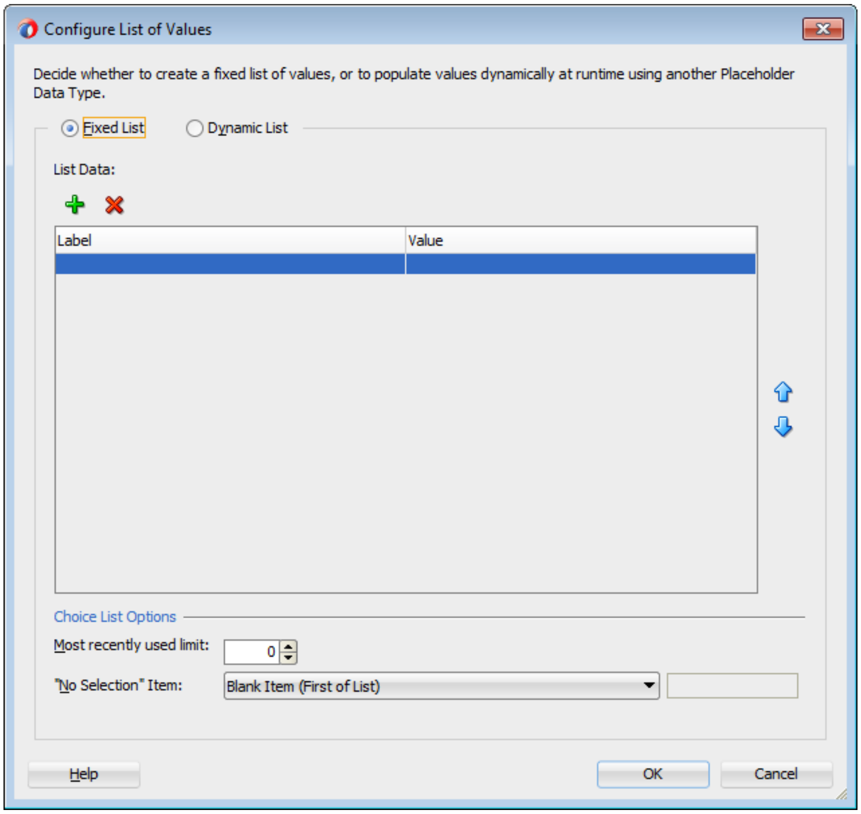
Additionally, you must have already created the placeholder data type for your placeholder data control.

To configure an attribute to be a fixed LOV:

1. In the Data Controls panel, right-click the placeholder data type and choose **Open**.
2. In the overview editor, click the **Attributes** navigation tab.
3. In the Attributes page, select the attribute you want to configure as an LOV.
4. Click the **Edit LOV Binding** icon next to the **Default Component** pulldown list.

   The Configure List of Values dialog appears, as shown in Figure 20-6.

Figure 20-6 Configure List of Values Dialog for a Fixed LOV



Configure List of Values

Decide whether to create a fixed list of values, or to populate values dynamically at runtime using another Placeholder Data Type.

● Fixed List ○ Dynamic List

List Data:

Label	Value

Choice List Options

Most recently used limit: 0

"No Selection" Item: Blank Item (First of List)

5. In the dialog, select **Fixed List**.
6. Click the **Add** icon to add an entry to the list of values.
7. For each entry, enter a label and a value.

   When the user selects an item from the list of values, the value entry will be entered into the input field.
8. Specify the maximum number of the most recently used items that will be displayed in the dropdown list.
9. From the **No Selection Item** dropdown list, select an option for how you want the "no selection" item to be displayed.

   For instance, selecting **Blank Item (First of List)** will display the "no selection" item as a blank at the beginning of the list.

10. Click **OK**.

Configuring an Attribute to Be a Dynamic LOV

Using a placeholder data type to serve as the source, you can configure an attribute to be a dynamic LOV.

Before you begin:

It may be helpful to have an understanding of the options you have for creating placeholder data types. For more information, see Creating Placeholder Data Types.

You may also find it helpful to understand additional functionality that can be added after using placeholder data controls. For more information, see Additional Functionality for Placeholder Data Controls.

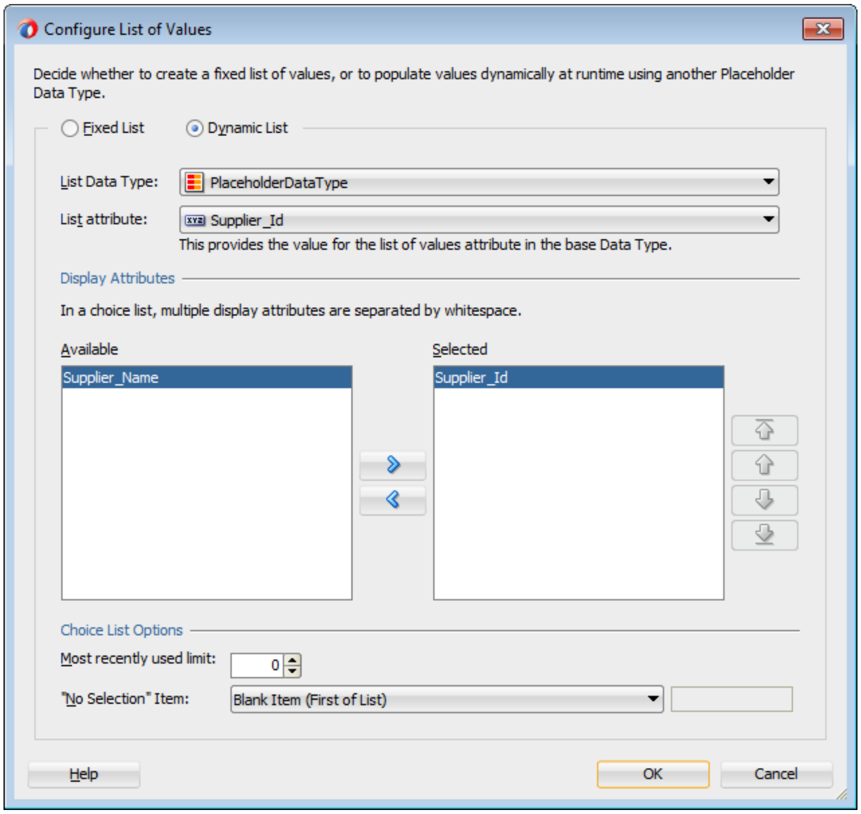
Also, you should have already created another placeholder data type to serve as the source of the dynamic LOV.

To configure an attribute to be a dynamic LOV:

1. In the Data Controls panel, right-click the placeholder data type and choose **Open**.
2. In the overview editor, click the **Attributes** navigation tab.
3. In the Attributes page, select the attribute you want to configure as an LOV.
4. Click the **Edit LOV Binding** icon next to the **Default Component** pulldown list.

   The Configure List of Values dialog appears, as shown in Figure 20-7.

Figure 20-7 Configure List of Values Dialog for a Dynamic LOV



Decide whether to create a fixed list of values, or to populate values dynamically at runtime using another Placeholder Data Type.

○ Fixed List ● Dynamic List

List Data Type: PlaceholderDataType

List attribute: Supplier_Id

This provides the value for the list of values attribute in the base Data Type.

Display Attributes

In a choice list, multiple display attributes are separated by whitespace.

Available: Supplier_Name

Selected: Supplier_Id

Choice List Options

Most recently used limit: 0

"No Selection" Item: Blank Item (First of List)

5. In the Configure List of Values dialog, select **Dynamic List**.
6. Select the list data type with the source attribute. You must have a source placeholder data type for this selection to be available.
7. Select the list attribute.
8. Shuttle the desired display attributes from the **Available** list to the **Selected** list.
9. Select the maximum number of the most recently used items that will be displayed in the dropdown list.
10. From the **No Selection Item** dropdown list, select an option for how you want the "no selection" item to be displayed. For example, selecting **Blank Item (First of List)** will display the "no selection" item as a blank at the beginning of the list.
11. Click **OK**.

How to Create Master-Detail Data Types

You create master-detail relationships between data types in the same way you create master-detail hierarchies between tables. In a standard data control, you can use view links to define source and target view objects that would become the master and the detail objects. For more information about master-detail relationships, see Displaying Master-Detail Data .

You first create a master data type and its attributes. Then you create a detail data type as a child of the master data type. You define the source attribute in the master data type that defines the relationship to the detail data type.

Before you begin:

It may be helpful to have an understanding of the options you have for creating placeholder data types. For more information, see Creating Placeholder Data Types.

You may also find it helpful to understand additional functionality that can be added after using placeholder data controls. For more information, see Additional Functionality for Placeholder Data Controls.

Before you create a placeholder master-detail hierarchy, you must perform the following tasks:

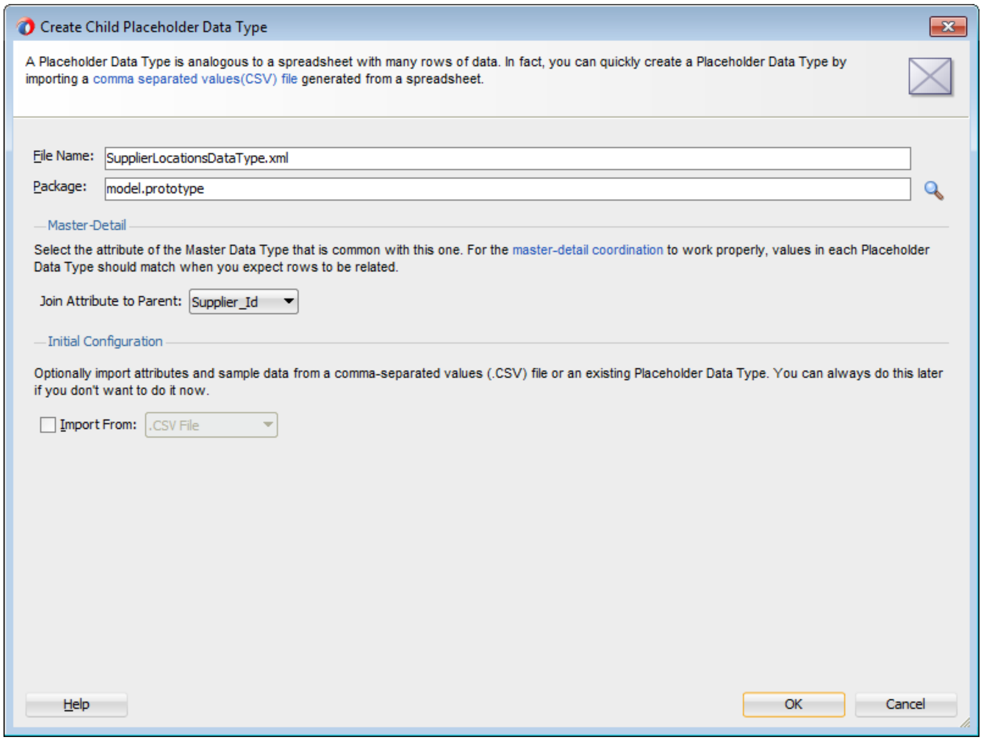
- Determine the data structure of the master data type and the data structure of the detail data type.
- Determine which attribute in the master will be the source for the detail data type.
- Create a placeholder data type to be the master, as described in How to Create a Placeholder Data Type.

To create master-detail hierarchical data types:

1. In the Data Controls panel, right-click the master placeholder data type and choose **New Child Placeholder Data Type**.

   Figure 20-8 shows the Create Placeholder Data Type dialog for entering detail data type attributes.

Figure 20-8 Create Child Placeholder Data Type Dialog



Create Child Placeholder Data Type

A Placeholder Data Type is analogous to a spreadsheet with many rows of data. In fact, you can quickly create a Placeholder Data Type by importing a comma separated values(CSV) file generated from a spreadsheet.

File Name: SupplierLocationsDataType.xml

Package: model.prototype

Master-Detail

Select the attribute of the Master Data Type that is common with this one. For the master-detail coordination to work properly, values in each Placeholder Data Type should match when you expect rows to be related.

Join Attribute to Parent: Supplier_Id

Initial Configuration

Optionally import attributes and sample data from a comma-separated values (.CSV) file or an existing Placeholder Data Type. You can always do this later if you don't want to do it now.

□ Import From: .CSV File

2. In the Create Placeholder Data Type dialog, enter a name and package for the detail data type.
3. From the **Join Attribute to Parent** dropdown list, select the attribute from the master data type that will be used to join to the new detail data type.
4. If you have an existing placeholder data type or a `.csv` file, from which you want to import attributes and data, select **Import From** and select either **.CSV File** or **Existing Data Type**, and then specify the options for the import.

   To import from a `.csv` file:

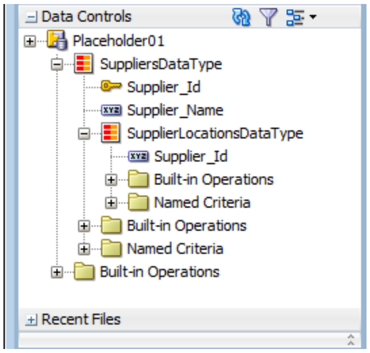
   a. In the **File Name** field, specify the file from which you want to import, or click the **Browse** icon to navigate to and select the file.
   b. From the **Character Set** dropdown list, select the character set that the file uses.
   c. If your file has a header row with the names of the columns, select **First row of the file contains Attribute names to import**.

   To import from an existing placeholder data type:

   a. Expand the tree to locate and select an existing placeholder data type.
   b. If the existing data type contains data that you want to import, select **Also Import Sample Data**.
5. Click **OK**.

The Data Controls panel displays the detail data type as a child of the master data type, as shown in Figure 20-9. If you did not import attributes, you can add them now, as described in How to Add Attributes and Sample Data to a Placeholder Data Type.

Figure 20-9   Master Detail Hierarchy in Placeholder Data Control



What Happens When You Create a Master-Detail Data Type

A master-detail relationship is implemented in the same way as is a standard master-detail relationship, using view object and view links. When you define placeholder data types in a master-detail hierarchy, JDeveloper creates a `DTLink.xml` file that contains metadata entries for view links that define that relationship. For more information about view links, see Working with Multiple Tables in a Master-Detail Hierarchy. For example, in the relationship between the master data type `SuppliersDataType` and the detail data type `SupplierLocationsDataType` associated with a key `Supplier_Id`, JDeveloper creates a `DTLink.xml` file in the form of a view link file to define that relationship, as shown in the following example.

```
<?xml version="1.0" encoding="windows-1252" ?>
<ViewLink
  xmlns="http://xmlns.oracle.com/placeholder"
  Name="DTLink"
  InheritPersonalization="merge">
  <ViewLinkDefEnd
    Name="SuppliersDataType"
    Cardinality="1"
    Source="true"
    Owner="model.prototype.SuppliersDataType">
    <AttrArray Name="Attributes">
      <Item Value="model.prototype.SuppliersDataType.Supplier_Id"/>
    </AttrArray>
  </ViewLinkDefEnd>
  <ViewLinkDefEnd
    Name="SupplierLocationsDataType"
    Cardinality="-1"
    Owner="model.prototype.SupplierLocationsDataType">
    <AttrArray Name="Attributes">
      <Item Value="model.prototype.SupplierLocationsDataType.Supplier_Id"/>
    </AttrArray>
  </ViewLinkDefEnd>
</ViewLink>
```

Using Placeholder Data Controls

You can use the placeholder data controls to create a layout, search form, or to bind components. After the real data controls are ready, you can rebind the components.

You can also package the placeholder data controls and add to ADF Library JARs and reuse them.
You use placeholder data controls in the same way you would use standard data controls. You can drag data types onto pages and use the context menus to drop the data types as forms, tables, trees, graphs, and other components. You can drop individual attributes onto pages as text, lists of values, single selections, and other components. You can use any of the built-in operations such as Create, Execute, and Next by dropping them as buttons, command links, and menu items.

You can work in several ways to take advantage of placeholder data controls:

- Build a page using the placeholder data controls and rebind to real data controls later.
- Build a page using components, bind them to placeholder data controls, and rebind to real data controls later.
- Build a page using some combination of components from the Components window, components from the placeholder data controls, and then bind or rebind to the real data controls later.

Limitations of Placeholder Data Controls

You can use placeholder data controls in your application development in many situations. For most UI design evaluations, placeholder data controls should be able to fully substitute for real data controls.

There are a few limitations:

- Because data types are not tied to an underlying data source, the Commit and Rollback operations do not perform real transactions, nor do they update the cache.
- Placeholder data controls can only be created declaratively. You cannot create custom methods like you can with a real application module or data control.
- Placeholder data controls cannot be used when there is a need either for custom data or for filtering or custom code to fetch data. Placeholder data controls will disable those operations.

Creating Layout

Use the drag-and-drop feature to create the page using the placeholder data controls, any available real data controls, and components from the Components window. If you intend to run a page or application that requires real data, enter sample data for your placeholder data types. If you have a large amount of sample data, you may be able to create CSV files from the data source and load them into the data type. You may also use spreadsheets and other tools to create CSV sample data files.

Creating a Search Form

In a standard data control, you can create view criteria on view objects to modify the query. These view criteria are also used for drag-and-drop creation of query and quick query search forms. The named view criteria items appear under the Named Criteria node for a data collection. For more information about query and quick query search forms, see Creating ADF Databound Search Forms .

For placeholder data controls, there is also a Named Criteria node under each data type node. An automatically created All Queriable Attributes item appears under this node and can be used to drag and drop onto pages to create the query or quick query search forms.

Binding Components

Instead of building the page using the data controls, for instance, if you are unsure of the shape of your data, you can lay out the page first using the Components window and later bind it to the data types, attributes, or operations of the placeholder data controls.

Rebinding Components

After the final data controls are available, you can simply rebind the components. You can select the component in the Structure window and use the context menu to open the relevant rebind dialog. You can also drag and drop the data control item onto the UI component to initiate a rebinding editor. The rebinding procedures are the same whether the component was originally bound to a placeholder data control or a standard data control.

For more information about rebinding components, see Creating a Basic Databound Page and Creating ADF Databound Tables .

Packaging Placeholder Data Controls to ADF Library JARs

A useful feature of placeholder data controls is that they allow parallel development and division of labor among developers and designers. You may be able to leverage that further by packaging placeholder data controls into reusable components as ADF Library JARs. ADF Libraries are JARs that have been packaged to contain all the necessary artifacts of an ADF component. For more information about reusable components and the ADF Library, see Reusing Application Components . You can create libraries of placeholder data controls and distribute them to multiple designers working on the same UI project. Because they are lightweight, you can even use them in place of available real data controls for the earlier phases of UI design.

21 Creating ADF REST Data Controls from ADF RESTful Web Services

This chapter describes how to create data controls for ADF RESTful web services that are based on ADF Business Components application modules.
This chapter includes the following sections:

- About Data Controls for RESTful Web Services Based on Application Modules
- Consuming Business Components as REST Resources with a REST Data Control
- Using REST Data Controls
- Configuring REST Data Controls Published by Application Modules

About Data Controls for RESTful Web Services Based on Application Modules

In ADF, using the Data Controls panel, you can create UI components to represent REST resources and methods through which you can consume the RESTful web services functionality.
ADF Business Components can be configured to publish RESTful web services. This enables you to develop applications that use such services published by other parts of the application or by a different application entirely. Therefore, when you use these services, you do not have to package the underlying business components with your application.

To simplify the consumption of RESTful web services that are based on ADF Business Components, you can use JDeveloper to create connections to these services and then generate data controls based on the resources encompassed by those connections. You can then build a client for the RESTful web service by using the Data Controls panel to create UI components to represent the REST resources and methods. You do not need to write any code to parse the web service or create a client web application that implements the service.

Though the web services exposed by application modules are based on REST principles, when you consume the services using data controls, you interact with them in much the same way that you would directly with an application module. The underlying implementation of the pages' data operations are based on REST methods, but you can build the pages with the same types of data control objects and built-in operations that you use when creating databound UI components based on application module data controls.

Though RESTful web services are stateless between requests, applications created with data controls based on RESTful web services published by application modules behave in a stateful manner. However, because the information about the state of the application is contained within the exchanged messages and not on the server, the application is stateless. User CRUD actions corresponding to HTTP methods (`GET`, `PATCH`, `POST`, and `DELETE`) are cached until the user invokes the `Commit` or `Rollback` operation, at which point the cache is cleared (and the data is committed to the data source or the changes are discarded).

> **Note:**
>
> This chapter concerns the creation of data controls that are based on RESTful web services that expose functionality developed with ADF Business Components. It is also possible to create data controls based on SOAP-based web services and RESTful web services that are not based on ADF Business Components. For more information, see "Exposing Web Services Using the ADF Model Layer" in *Developing Applications with Oracle ADF Data Controls*.

REST Web Service Data Control Use Cases and Examples

You can create a data control to consume RESTful web services that are exposed by ADF Business Components. You then use the data controls to create databound UI components on a web page to access the service's functionality.

Typically you consume such web services as a way to obtain functionality for an application without having to develop that functionality yourself. In addition, you do not need to package the base application modules directly in your application. These RESTful web services that you consume might be published by a partner, a service provider, or another development team, or they might be part of a cloud-based service.

Additional Functionality for REST Web Service Data Controls

You may find it helpful to understand other data access features before you start working with RESTful web services based on ADF Business Components. Following are links to other functionality that may be of interest.

- For further details on how ADF data controls and data binding work, see Using ADF Model in a Fusion Web Application.
- For information on designing databound user interfaces by dragging items from the Data Controls panel and dropping them on pages as UI components, see Creating a Basic Databound Page.
- For information about publishing RESTful web services based on ADF Business Components, see Creating RESTful Web Services with Application Modules.

Consuming Business Components as REST Resources with a REST Data Control

ADF provides a wizard to create an IDE connection for RESTful web services based on ADF Business Components, which is added to the Resource Window. You can then use this IDE connection in any application window to consume the RESTful web services.
The REST data control enables you to access and consume data streams from specified URIs defined by running RESTful web services that have been published

by ADF Business Components. There are two ways of creating the connections that consumes RESTful web services that are based on the ADF Business Components:

1. Create an IDE connection to the `describe` resource of the RESTful web service that has been published by the application module and then create a data control that is based on that connection.
2. Create a REST connection and the data controls together by using the Web Service Data Control wizard.

How to Create a REST Connection

You use the Create REST Connection dialog to create an IDE connection to a `describe` resource for a RESTful web service based on an ADF Business Components application module. You can access this wizard through the New Gallery or the Resources window. Any connection that you create with this wizard is added to the Resources window and can be used to create a data control in any application workspace.

Before you begin:

It may be helpful to have a general understanding of REST data controls. For more information, see Consuming Business Components as REST Resources with a REST Data Control.

You may also find it helpful to understand additional functionality that can be added using other RESTful web services features. For more information, see Additional Functionality for REST Web Service Data Controls.

> Make sure you have access to the RESTful web service that the data control will access. To use the describe feature to create a REST connection, the RESTful web service must be based on an ADF Business Components application module.

To create a REST connection:

1. From the main menu, choose **File > New > From Gallery**.
2. In the New Gallery, expand **General**, select **Connections** and then **REST Connection**, and click **OK**.
3. In the Create REST Connection dialog, enter a name for the connection.
4. Select the **IDE Connections** radio button.
5. In the **URL Endpoint** field, enter the URI for the REST resource's `describe` object.

> **Note:**
>
> This URI is composed of the host, port, resource path, and the `describe` resource. The resource path typically consists of the context root of the web service project plus the URL pattern specified in the servlet mapping in the REST project's `web.xml` file. Do include the version name and resource name, but do not include any URL parameters.
>
> For example, you can enter something like the following:
>
> ```
> http://service.example.com:7101/WebServiceProjectContextRoot/
> rest/VersionName/ResourceName/describe
> ```

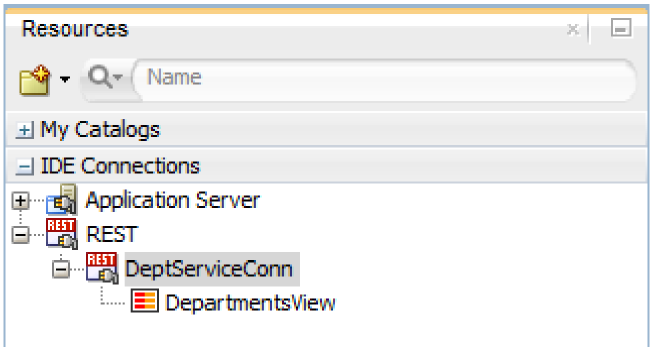
6. Select the level of authentication from the **Authentication Type** dropdown list.

   None is the default authentication type and disables authentication. Use **Digest** when security is desired. In this way, the password will be transmitted across the network as an MD5 digest of the user's password and cannot be determined by sniffing network traffic. **Basic** authentication is primarily only useful when service access over the network does not require high security.
7. If digest or basic authentication is selected, specify the user name and password required to access the web site.
8. After you have entered the name and endpoint, you can click **Test Connection** to verify the REST connection is valid.
9. Click **OK**.

What Happens When You Create a REST Connection

The REST connection is created as an IDE connection in JDeveloper and is available for use in any application that you are developing in the IDE. You can view the connection in the IDE Connections panel of the Resources window as shown in Figure 21-1.

Figure 21-1 Resources Window with a REST Connection



How to Create a Data Control From ADF REST Resources

You can create a REST data control by dragging a resource listed under a REST connection from the Resources window to the Data Controls panel. Unlike with the creation of REST data controls based on other types of RESTful web services, you do not need to go through a complex wizard to configure how the data control exposes individual resources.

Before you begin:

It may be helpful to have a general understanding of REST data controls. For more information, see Consuming Business Components as REST Resources with a REST Data Control.

You may also find it helpful to understand additional functionality that can be added using other RESTful web services features. For more information, see Additional Functionality for REST Web Service Data Controls.

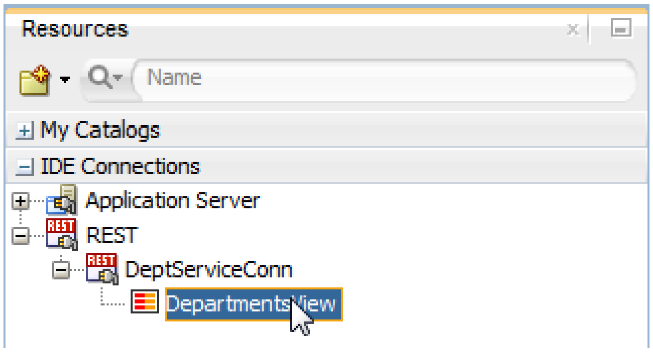
You need to complete these tasks:

- Create an application workspace and a project in that workspace. Depending on how you decide to organize your projects, you can use an existing application workspace and project or create new ones. For information on creating an application workspace, see "Creating Applications and Projects" in *Developing Applications with Oracle JDeveloper*.
- Create a REST connection as described in How to Create a REST Connection.

To create a REST data control:

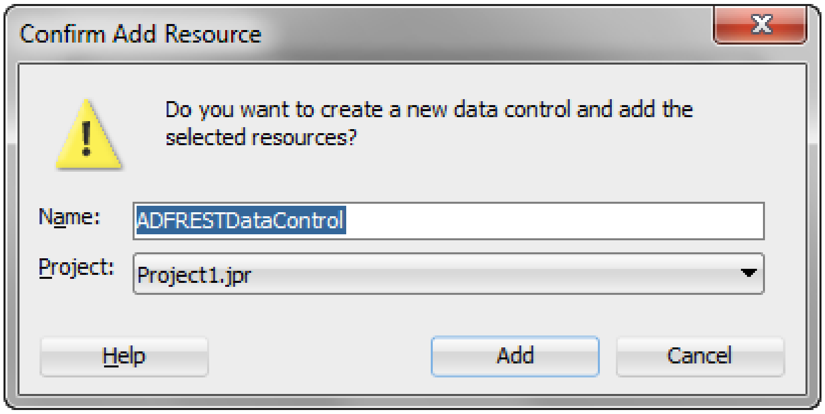
1. From the main menu, choose **Window > Resources**.
2. In the Applications window, select the node for the project where you want to create the data control.
3. In the Resources window, expand **IDE Connections** and expand the REST connection.
4. Select the resource on which you want to base the data control, as shown in Figure 21-2, and drag it to the Data Controls panel.

Figure 21-2 Resource Selected from RESTful web service Connection



5. In the Confirm Add Resource dialog as shown in Figure 21-3, select the project where you want to generate the data control and click Add.

Figure 21-3 Confirm Add Resource Dialog



Confirm Add Resource

Do you want to create a new data control and add the selected resources?

Name: ADFRESTDataControl

Project: Project1.jpr

6. Repeat steps 4 and 5 for each resource that you want to include in the data control.

How to Create an ADF Data Control Using the Web Service Data Control Wizard

You can use the Web Service Data Control wizard to create a REST connection and the **data controls** together. The REST connection you create will consume the RESTful web services that are based on the ADF Business Components. Creating a data control in this way does not require any manual configuration, because the necessary metadata is provided in the REST service's describe endpoint.

Before you begin:

It may be helpful to have an understanding of how web service data controls are used in ADF applications. For more information, see About Web Service Data Controls in ADF Applications.

It may be helpful to have an understanding of RESTful web services based on ADF Business Components. For more information, see Creating RESTful Web Services with Application Modules.

You may also find it helpful to understand additional functionality that can be added using other web services features. For more information, see Additional Functionality for Web Service Data Controls in ADF Applications.

You will need to complete these tasks:

- Create the ADF REST resource and deploy it as a RESTful web service, as described in Creating RESTful Web Services with Application Modules.

To create a REST connection and data control for a RESTful web service using the Data Control Wizard:

1. In the Applications window, right-click the project in which you want to create a web service data control and choose **New > From Gallery**.
2. In the New Gallery, select **All Items**, scroll down, select **Web Service Data Control (SOAP/REST) (Data Controls)**, and then click **OK**.
3. In the Create Web Service Data Control wizard, on the Data Source page, specify a name for the data control and select the **REST** radio button.

You can either create Describe-Based ADF Data Control or Generic Data Control with manually described features.

4. Select **Describe-Based ADF Data Control**.
5. Click the **Create a new REST connection** icon to open the Create REST Connection dialog.
6. Enter a name for the connection and a describe URI, which is the RESTful web service that is based on the ADF Business Components. Do not include any resources or parameters in the URL.

   This URI is composed of the host, port, resource path, and the describe resource. The resource path typically consists of the context root of the web service project plus the URL pattern specified in the servlet mapping in the REST project's `web.xml` file. Do include the version name and resource name, but do not include any URL parameters. For example, you can enter something like the following:

   ```
   http://service.example.com:7101/WebServiceProjectContextRoot/rest/
   VersionName/ResourceName/describe
   ```

7. Select the level of authentication from the **Authentication Type** dropdown list. **None** is the default authentication type and disables authentication. Use **Digest** when security is desired. In this way, the password will be transmitted across the network as an MD5 digest of the user's password and cannot be determined by sniffing network traffic. **Basic** authentication is primarily only useful when service access over the network does not require high security.

   > **Note:**
   >
   > In the Create REST Connection dialog, you can click **Test Connection** to verify that you can connect to the URI. However, the URI's server may be configured to not accept requests on the base URI, meaning that the test will fail. Regardless of that fact, you can click **OK** to create the connection. If you have such a base URI and would like to make sure that you can connect to the service, you can temporarily add a resource to the URL, test the connection, and then remove the resource before clicking **OK**.

8. Click **Next**.
9. On the OWSM Policies page, you can optionally set policies for the web service client as required, and click **Next**
10. Based on the web service connection that you specified, the Select Resources page displays the list of resources. You can either select all the resources to add to the data control, which is the default selection, or select a specific set of resources to add to the data control.
11. Click **Next**, and click **Finish**.

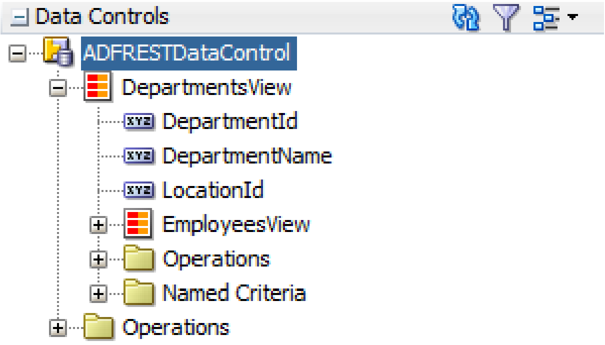
What Happens When You Create a Data Control Based on a REST Connection

When you create a data control based on a resource encompassed by a REST connection, JDeveloper creates a data control definition file (`DataControls.dcx`), opens the file in the overview editor, and displays the data control's hierarchy in the Data Controls panel.

The `DataControls.dcx` overview editor is populated with collection nodes for each resource. The Data Controls panel displays those same collection nodes in addition to subnodes for attributes and operations for those collections.

For example, Figure 21-4 shows a data control with collection object called `DepartmentsView` that corresponds with a resource exposed by the RESTful web service. Its subnodes include attributes of that collection and a nested collection, which can be used to create a list-of-values (LOV) component. In addition, it exposes a combination of data control methods and built-in data control operations that are mapped to standard REST methods.

Figure 21-4 Data Controls Panel with REST Data Control Displayed



For operations that take a parameter of a complex data type, a structured attribute node also appears.

For more information on the objects displayed in the Data Controls panel, see Using REST Data Controls.

Using REST Data Controls

Similar to other data controls, you can use the Data Controls panel for the REST data controls. You can drag an drop the components from the Data Control panel, add attributes, and configure them.
As with other kinds of data controls, you can design a databound user interface by dragging an item from the Data Controls panel and dropping it on a page as a specific UI component. For information on general use of the Data Controls panel, see How to Use the Data Controls Panel.

In the Data Controls panel, each data control object is represented by an icon. Table 21-1 describes what each icon represents, where it appears in the Data Controls panel hierarchy, and what components it can be used to create.

> **Note:**
>
> You can also create data controls based on REST web services that are not based on ADF Business Components. However, such data controls have some differences in the exposed methods and resources. For example, the `Commit` and `Rollback` operations are not exposed for REST data controls that are not based on ADF Business Components. For information on creating and using REST data controls not based on business components, see Consuming Web Services Using the ADF Model Layer in *Developing Applications with Oracle ADF Data Controls*.

Table 21-1   Data Controls Panel Icons and Object Hierarchy for REST Data Control

Icon	Name	Description	Used to Create...
	Data Control	Represents a data control. You cannot use the data control itself to create UI components, but you can use the child objects listed under the data control. There may be more than one data control, each representing a logical grouping of data functions.	Serves as a container for the other objects. Not used to create anything.

Table 21-1 (Cont.) Data Controls Panel Icons and Object Hierarchy for REST Data Control

Icon	Name	Description	Used to Create...
▤	Method	Represents any custom non-HTTP methods that are exposed as part of the web service. **Note:** When you specify a HTTP payload method, the Web Services Data Control generates a schema and displays the specified operations (GET, POST, PATCH, or DELETE) in the Data Controls panel. These payload operations are represented as Method. If a Method accepts arguments, those arguments appear in a Parameters node as parameters, nested inside the method's node. **Note:** While designing a page, you can use these payload operation methods on a form as a command button or link. When you drop an operation on a form, the operation takes the corresponding input parameters that is already dropped as input. You can then bind the form to this button.	Command components. For methods that accept parameters: command components and parameterized forms. See Using Command Components to Invoke Functionality in the View Layer.

Table 21-1 (Cont.) Data Controls Panel Icons and Object Hierarchy for REST Data Control

Icon	Name	Description	Used to Create...
	Method Return	Represents an object that is returned by a web service method. The returned object can be a single value or a collection. A method return appears as a child under the method that returns it. The objects that appear as children under a method return can be attributes of the collection, other methods that perform actions related to the parent collection, and operations that can be performed on the parent collection. When a single-value method return is dropped, the method is not invoked automatically by the framework. You should either drop the corresponding method as a button to invoke the method, or if working with task flows you can create a method activity for it. For more information about executables, see Executable Binding Objects Defined in the Page Definition File.	The same components as for collections and attributes and for query forms. For more information on query forms, see Creating ADF Databound Search Forms .
	Collection	Represents a data collection returned by an operation on the URL service. Collections appear as children under method returns, other collections, or structured attributes. The children under a collection may be attributes, other collections, custom methods, and built-in operations that can be performed on the collection.	Forms, tables, graphs, trees, range navigation components, and master-detail components. See Creating a Basic Databound Page, Creating ADF Databound Tables , Displaying Master-Detail Data , and Creating Databound Chart and Gauge Components..
	Structured Attribute	Represents a returned object that is a complex type but not a collection. For example, a structured attribute might represent a single user assigned to the current service request. **Note:** If you specify a HTTP payload method, the data control generates a schema for specified operations (GET, POST, PATCH, or DELETE) and displays the input parameters for each operation separately in the Data Controls panel. While designing a page, you can use these input parameters for the operations and can drop on a page as an editable form.	Label, text field, date, list of values, and selection list components For more information, see Creating Text Fields Using Data Control Attributes and Creating Databound Selection Lists and Shuttles ..

Table 21-1 (Cont.) Data Controls Panel Icons and Object Hierarchy for REST Data Control

Icon	Name	Description	Used to Create...
XYZ	Attribute	Represents a discrete data element in an object (for example, an attribute in a row). Attributes appear as children under the collections or method returns to which they belong.	Label, text field, and selection list components. For more information, see Creating Text Fields Using Data Control Attributes.
⚙	Operation	Represents a built-in data control operation that performs actions on the parent object. Data control operations are located in an **Operations** node under collections and under the data control's root node. The following navigation operations are supported: `First`, `Last`, `Next`, and `Previous`. `Execute` is supported for refreshing queries. Because REST data controls based on business components are transactional, any data operations that a user performs are batched together and not committed to the data source until the user issues the `Commit` operation. Likewise, the `Rollback` operation is available for the user to undo any uncommitted changes.	UI command components, such as buttons, links, and menus. See Creating Command Components Using Data Control Operations and Creating an Input Form.
🟩	Parameter	Represents a parameter value that is declared by the method under which it appears. Parameters appear in a node under a method or operation.	Label, text, and selection list components.

> **Note:**
>
> In order to use the Data Controls panel for a service based on a REST connection, the service needs to be running and accessible through the connection. Otherwise the Data Controls panel will not have the information necessary to display the structure of the service and enable you to create UI components based on those elements.

Configuring REST Data Controls Published by Application Modules

In ADF, you can use the overview editor for the DataControls.dcx file to configure the REST data controls that are based on RESTful web services. This file stores the metadata and configuration details for the data control that is specific to a resource. After you create a REST data control based on a RESTful web service that is published by an ADF Business Components application module, you can use declarative metadata to configure some aspects the default appearance and behavior of the UI components created from the data control. The things that you can configure this way include:

- Attribute default value

- UI hints for labels, tooltips, number formats, and so on
- List of Value (LOV) objects
- Validation rules

The mechanism for setting this metadata is similar to what is provided for configuring entity objects and view objects.

How to Configure a REST Data Control

You can make a data control configurable by using the overview editor for the `DataControls.dcx` file to create data control structure files that correspond to objects encompassed by the data control. You can then edit the individual data control structure files.

Before you begin:

It may be helpful to have a general understanding of data control configuration. For more information, see Configuring REST Data Controls Published by Application Modules.

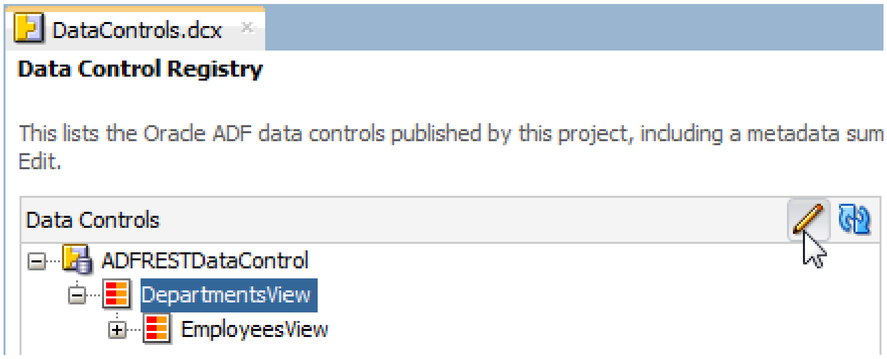
You will need to complete this task:

Create a REST data control, as described in How to Create a Data Control From ADF REST Resources.

To configure a REST data control:

1. In the Applications window, double-click **DataControls.dcx**.
2. In the overview editor, select the object that you would like to configure and click the **Edit** icon to generate a data control structure file, as shown in Figure 21-5.

Figure 21-5 Overview Editor for the DataControls.dcx File



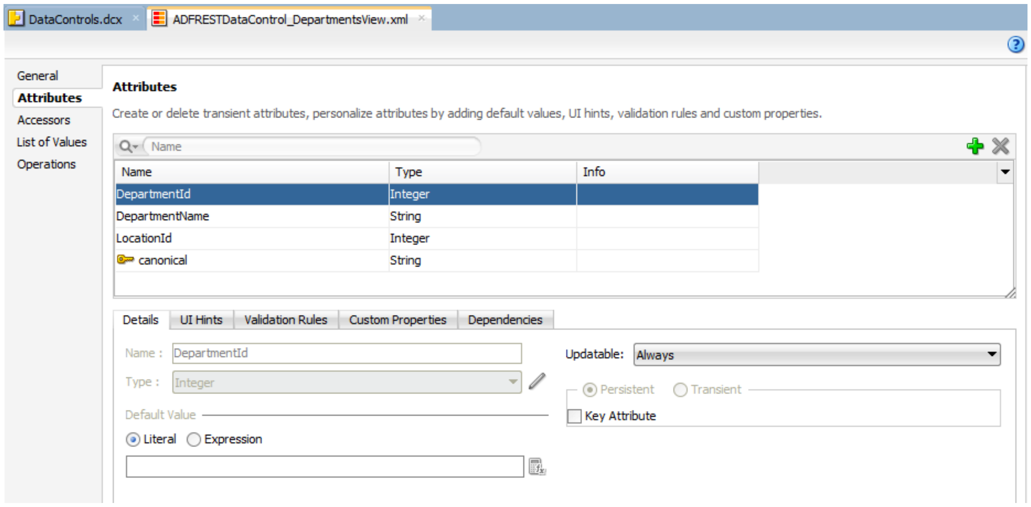
> **Note:**
>
> To edit a REST data control, it must be based on a REST connection. If you created the REST data control from a URL connection in a previous version of JDeveloper, the connection was converted to a REST connection when you migrated the application.

3. In the overview editor of the data control structure file, make the desired modifications.

What Happens When You Edit a Data Control

When you edit a data control, JDeveloper creates a data control structure file that contains metadata for the affected collection and opens that file in the overview editor, as shown in Figure 21-6. This file stores configuration data for the data control that is specific to that resource, such as any UI hints or validators that you have specified for the data object.

Figure 21-6 Overview Editor for a Data Control Structure File



DataControls.dcx | ADFRESTDataControl_DepartmentsView.xml

General
Attributes
Accessors
List of Values
Operations

Attributes

Create or delete transient attributes, personalize attributes by adding default values, UI hints, validation rules and custom properties.

Name	Type	Info
DepartmentId	Integer	
DepartmentName	String	
LocationId	Integer	
canonical	String	

Details | UI Hints | Validation Rules | Custom Properties | Dependencies

Name : DepartmentId

Type : Integer

Updatable: Always

● Persistent ○ Transient

□ Key Attribute

Default Value

● Literal ○ Expression

> **Note:**
>
> REST data controls based on services published by application modules also inherit some of the UI hints that the developer of those application modules may have set on the base entity objects and view objects. If you set a UI hint on the REST data control, that definition takes precedence over any conflicting ones set at the entity object or view object level.

For more information on UI hints, validation rules, and LOVs, see the following sections:

- Defining UI Hints for View Objects.
- Using the Built-in Declarative Validation Rules.
- Working with List of Values (LOV) in View Object Attributes.

What You May Need to Know About Primary Keys in REST Data Controls

Because a unique REST resource is identified by a specific resource URI rather than a data attribute, the key for a REST data control based on a service published by an application module is represented by a generated attribute called `canonical`. You can see this indicated by the key icon on the Attributes page of the overview editor for the data control structure file, as shown in Figure 21-6.

This is important to note if you want to use the `setCurrentRowWithKey` and `setCurrentRowWithKeyValue` operations. These operations require a parameter (`rowKey`) that identifies the row you want to set as current. So, rather than using the primary key of the entity object as you would when configuring an application module data control, you use the generated `canonical` attribute.

22
Consuming ADF RESTful Web Services

This chapter describes the supported HTTP methods, HTTP headers, request URL parameters, media types, and other concepts of the ADF REST runtime and the use cases that it supports for making REST API calls on resources exposed by ADF RESTful web services created for an ADF Business Components Model project. The information described in this chapter is useful to the ADF Business Components developer who wants to create RESTful web services for consumption by web service clients through a REST API.

This chapter includes the following sections:

About the ADF REST Framework

ADF provides a REST framework that supports creating and interacting with resources and ADF RESTful web services based on ADF Business Components. ADF REST framework supports the exchange of resource information by client and server during runtime.

The ADF REST framework is an ADF Model framework that allows Fusion web application developers to expose a Web API based on the REST architectural style. The framework itself does not constitute a Web API, but supports creating and interacting with resources and RESTful web services based on ADF Business Components business objects. A consequence of being RESTful, is a REST API that allows client application developers to interact with exposed business objects.

In the Fusion web application, ADF REST resources acted on by RESTful web services are backed by view object instances exposed in the data model of the ADF Business Components Model project. ADF Business Components developers working in the ADF Model project can decide on the set of attributes to expose from backing view objects, the actions to make CRUD operations available, and the view link relationships to preserve for the resulting resource.

The design-time choices made by ADF Business Components developers are captured in REST resource definition XML files. Web service client developers may interact with these resource definitions through the REST API supported by the ADF REST runtime framework. As a result, the consuming service client may invoke CRUD operations to interact with the REST resources and ADF business objects. The data is shaped by the resource's backing view object instance, with the parent-child relationships intact.

ADF REST Resource Use Cases and Examples

Any ADF Business Components developer working in the ADF Model project can publish ADF REST resources to contribute to the Fusion web application.

Specific runtime features of ADF REST framework that support the exchange of resource information by client and server include:

- Interacting with the REST resource is supported by separate JSON-based media type structures for the resource, action execution, and the results of action execution.
- Passing a custom header to specify the version of the REST API framework that will be used to process requests. The version header allows you to override the default version declaration defined by the web application. By specifying the REST API framework version, the web application developer may opt into framework enhancements made by Oracle.
- Interacting with the REST resource using standard HTTP request methods, including GET, POST, PATCH, and DELETE.
- Executing an HTTP method using an `X-HTTP-Method-Override` header using a POST request method when client restrictions prevent the use of standard HTTP methods to interact with the REST resource.
- Merging payload content with the REST resource using a POST method in combination with a header `Upsert-Mode` set to true. This action implements create if the record does not exist or update if the record exists.
- Locating a REST resource item (such as a specific employee) or resource collection (such as an collection of all employees) is supported by REST-compliant URI path names based on resource names defined in the ADF Business Components Model project.
- Obtaining a description of the REST resource, including the resource collection attributes and available actions, is supported by a specific media type and describe action.
- Obtaining a error responses in the form of a JSON payload with exception detail when ADF REST framework version 4 or later is enabled. Alternatively, with framework version 3 or earlier, error responses are in the form of a simple message string.

- Linking to a canonical REST resource is supported when a resource with a super set of updatable entities is defined. The canonical link supports alternate links for the backing view object.
- Filtering of the REST resource is supported by query string parameters on the URI specified by the client to access the specific resource.
- Encoding formats are supported on the REST resource to enable compression and decompression.
- Content streaming of BLOB and CLOB attributes exposed by the REST resource is supported by the ADF REST framework.
- Performing data consistency checks while invoking the RESTful web service is supported by the ADF REST framework. This capability uses version history in the database to enable clients to manage HTTP payloads according to updates in the resource itself.
- Passing a custom header to specify the ADF REST framework version (for example, version 2) with which the ADF REST runtime will use to process the request. The ADF REST framework version permits service clients to opt into framework enhancements made by Oracle in subsequent releases of Oracle JDeveloper.
- Authorizing users to access the REST resource is enabled by ADF Security permission grants.

Specific design time features of ADF REST framework that affect the availability of runtime functionality include:

- Registering REST resources with an application-specific resource version name (or number) to support versioning of REST resources. REST resource versions are named in the `adf-config.xml` file for the application.
- Declaring a default ADF REST framework version for individual REST resources to ensure the ADF REST runtime executes requests using functionality offered by a particular ADF REST framework version. Declaring a particular framework version allows service clients to opt into framework enhancements made by Oracle in subsequent releases of Oracle JDeveloper. The default ADF REST framework version is declared in the `adf-config.xml` file for the application. In JDeveloper starting with release 12.2.1.2.0, versions 1, 2, and 3 exist. In the release 12.2.1.4.0 and later, versions 1, 2, 3, 4, 5, 6, and 7 exist.
- Defining List of Value (LOV) attributes on the view object backing the resource to expose LOV accessor links in the URL resource path.
- Defining row finders to locate a resource using the row finder key value in the URL resource path.
- Defining view instance of the application data model using a super set of updatable attributes to expose canonical links in the URL resource path.
- Configuring change-indicator attributes on the entity object backing the resource to enable data consistency checking when making requests to access resources on the server.
- Configuring a custom content type on the BLOB or CLOB attribute of the view object backing the resource to support content streaming using a link exposed in the resource.
- Configuring security to authenticate users before allowing access to the resource.

Additional Functionality for RESTful Web Services

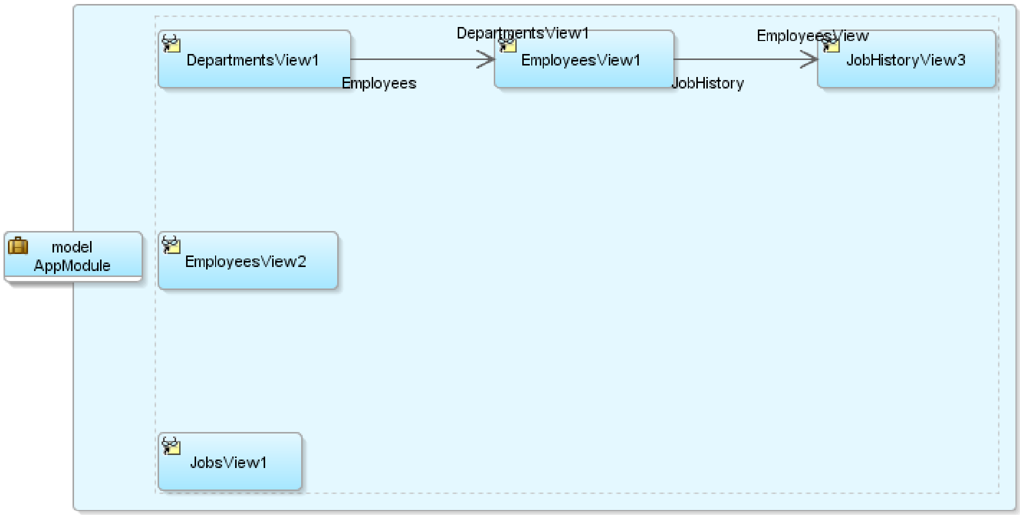
You may find it helpful to understand related Oracle ADF features before you start working with RESTful web services. Following are links to supporting functionality that may be of interest.

- For details about ADF Business Components and creating an ADF Business Components Model project, see Getting Started with ADF Business Components.
- For details about creating RESTful web services in the ADF Business Components project, including enabling LOV links, row finders, canonical resources, change indicators, custom content types for LOB attributes, and security, see Creating ADF RESTful Web Services with Application Modules.
- For details about using the ADF REST data control to design the user interface, see Creating ADF REST Data Controls from ADF RESTful Web Services.

About the Resource Samples in This Chapter

This chapter describes typical use cases for interacting with and manipulating ADF REST resources. All samples are based on DEPARTMENTS, EMPLOYEES, JOBHISTORY, and JOBS tables in the Oracle HR schema. Figure 22-1 depicts the view objects generated from these table in the ADF Business Components model project.

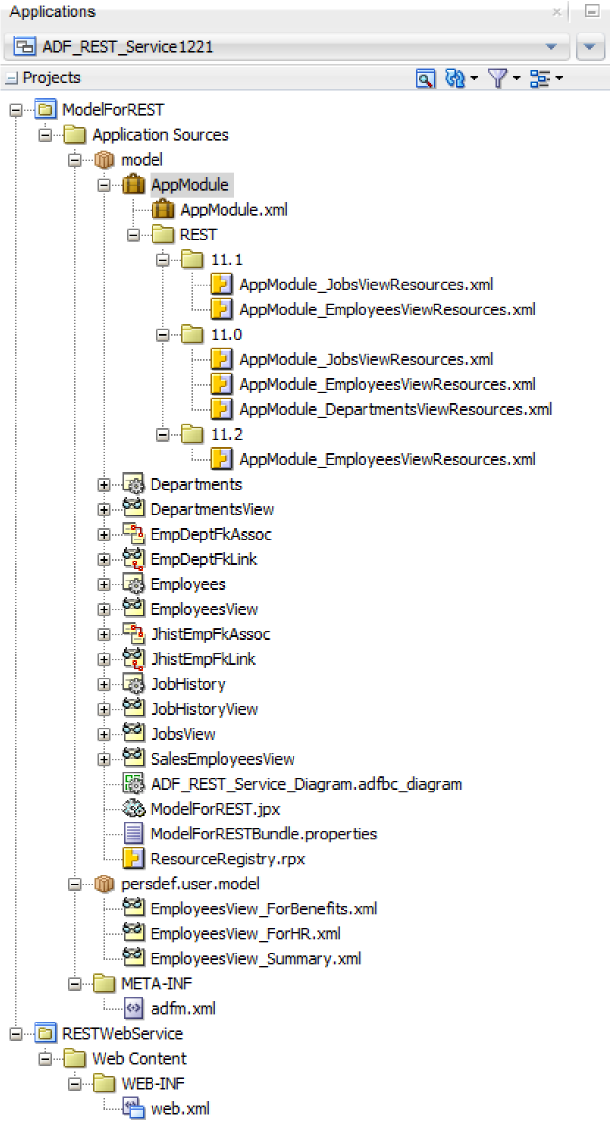
Figure 22-1   REST Sample Project - ADF Business Components View Instances



Sample Project Files and Data Model

As Figure 22-2 shows, the Model project and the business components it defines, includes the ADF REST resource definition files. JDeveloper also generates the **RESTWebService** project to enabling running and testing resources in JDeveloper.

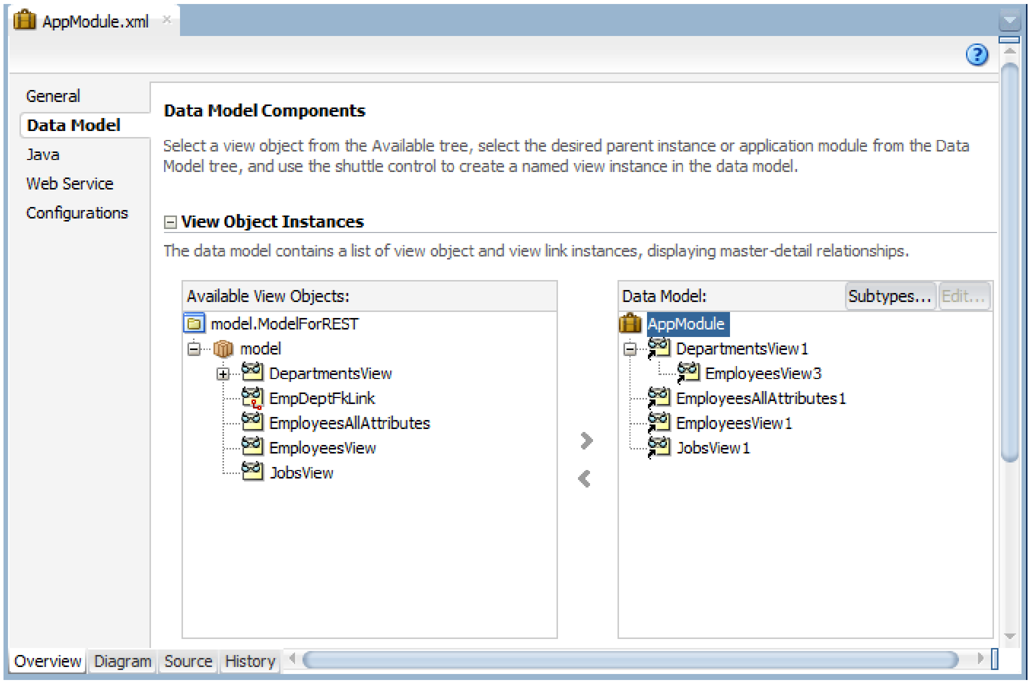
Figure 22-2   REST Sample Application - ADF Business Components



```
Applications
ADF_REST_Service1221
Projects
ModelForREST
    Application Sources
        model
            AppModule
                AppModule.xml
                REST
                    11.1
                        AppModule_JobsViewResources.xml
                        AppModule_EmployeesViewResources.xml
                    11.0
                        AppModule_JobsViewResources.xml
                        AppModule_EmployeesViewResources.xml
                        AppModule_DepartmentsViewResources.xml
                    11.2
                        AppModule_EmployeesViewResources.xml
            Departments
            DepartmentsView
            EmpDeptFkAssoc
            EmpDeptFkLink
            Employees
            EmployeesView
            JhistEmpFkAssoc
            JhistEmpFkLink
            JobHistory
            JobHistoryView
            JobsView
            SalesEmployeesView
            ADF_REST_Service_Diagram.adfbc_diagram
            ModelForREST.jpx
            ModelForRESTBundle.properties
            ResourceRegistry.rpx
        persdef.user.model
            EmployeesView_ForBenefits.xml
            EmployeesView_ForHR.xml
            EmployeesView_Summary.xml
        META-INF
            adfm.xml
RESTWebService
    Web Content
        WEB-INF
            web.xml
```

As Figure 22-3 shows, the `Departments` resource is backed by the `DepartmentsView` view object and the child resource `Employees` is enabled.

Figure 22-3   REST Sample Project - Application Module Data Model



Data Model Components

Select a view object from the Available tree, select the desired parent instance or application module from the Data Model tree, and use the shuttle control to create a named view instance in the data model.

View Object Instances

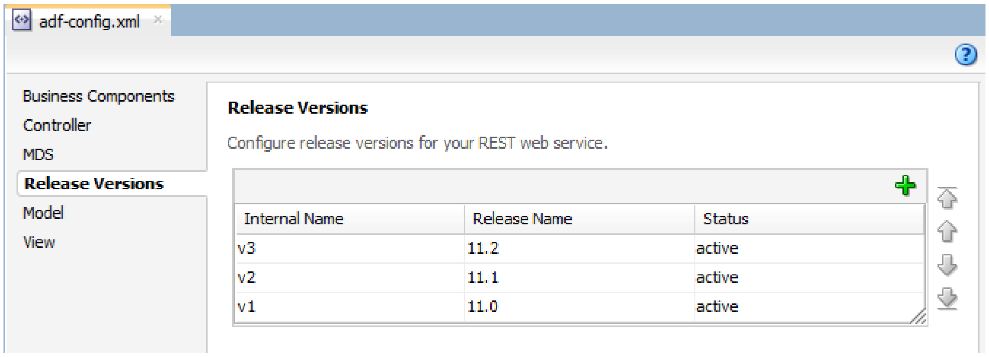
The data model contains a list of view object and view link instances, displaying master-detail relationships.

Sample Project Resources and Resource Version Identifiers

As Figure 22-4 shows, the application defines three version release names. Each ADF REST resource is assigned to one of these version identifiers and accessed at runtime by the version release name. For example, several versions of the `Employees` resource were created: one with a limited set of supported operations and another with a more complete set of operations.

Note that the order of the version identifiers determines the version sequence, with the most recent version appearing at the top of the list. In the REST samples, where multiple versions of the `Employees` resource exist, the release name `11.2` identifies the latest version.

Figure 22-4   REST Sample Project - Resource Versions



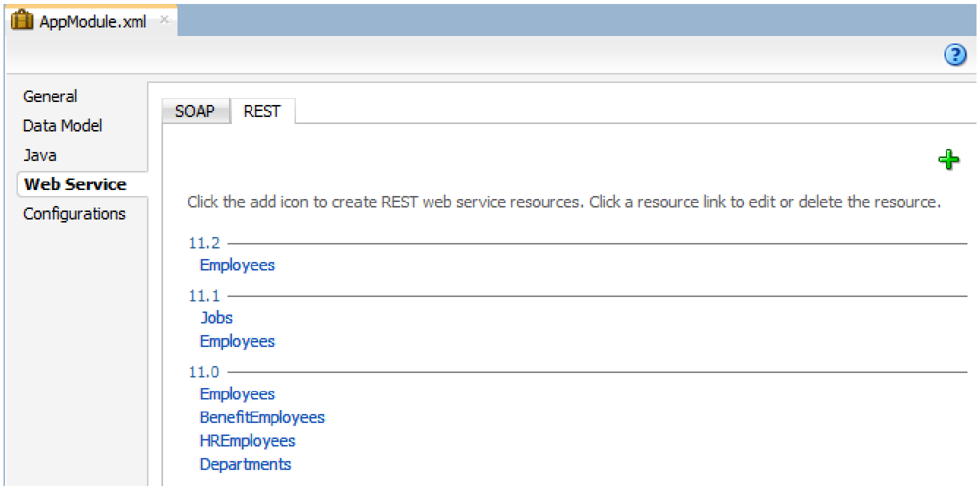
Release Versions

Configure release versions for your REST web service.

Internal Name	Release Name	Status
v3	11.2	active
v2	11.1	active
v1	11.0	active

As Figure 22-3 shows, the application module organizes the created resources by their assigned version release name. The Employees resource is originally created in version 11.0 and then revised in subsequent versions 11.1 and 11.2 with the addition of new mandatory attributes. The Jobs resource added in 11.1 supports create operations on the Employees resource with LOV-enabled attributes.

Figure 22-5 REST Sample Project - Application Module Resource Names



Click the add icon to create REST web service resources. Click a resource link to edit or delete the resource.

11.2
- Employees

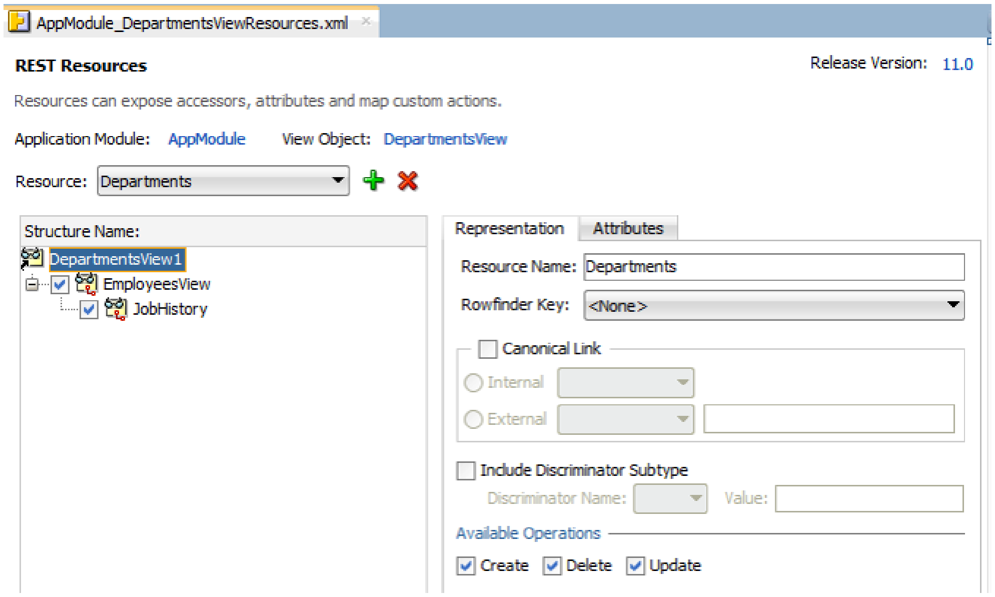
11.1
- Jobs
- Employees

11.0
- Employees
- BenefitEmployees
- HREmployees
- Departments

As Figure 22-6 shows, the Departments resource is backed by the DepartmentsView view object and the child resources Employees and JobHistory are enabled.

Note that the name of the child resource Employees is derived from the destination view instance name in the defining view link. In the sample, the destination view instance was renamed from EmployeesView1 to Employees. This supports accessing the child resource using the name Employees in the URL resource path instead of the default name EmployeesView1.

Figure 22-6 REST Sample Project - Departments Resource



REST Resources

Release Version: 11.0

Resources can expose accessors, attributes and map custom actions.

Application Module: AppModule View Object: DepartmentsView

Resource: Departments

☑ EmployeesView
☑ JobHistory

Resource Name: Departments

Rowfinder Key: <None>

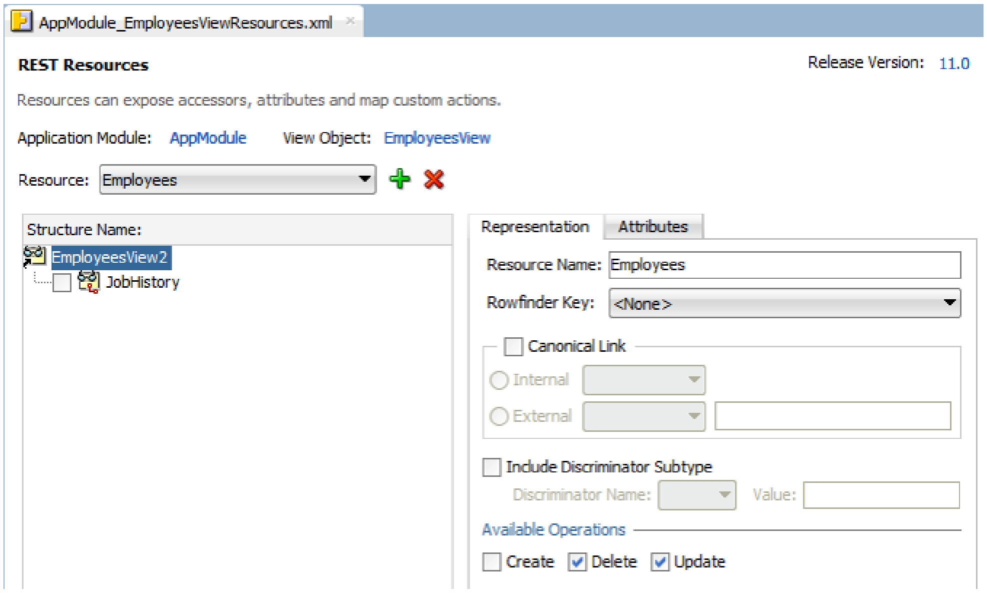
□ Canonical Link
○ Internal
○ External

□ Include Discriminator Subtype
Discriminator Name: Value:

☑ Create ☑ Delete ☑ Update

As Figure 22-7 shows, version `11.0` of the `Employees` root resource is backed by the `EmployeesView` view object. Because `Employees` was created as root resource, it has no child resource accessors.

Figure 22-7 REST Sample Project - Employees Resource Version 11.0



AppModule_EmployeesViewResources.xml

REST Resources

Release Version: 11.0

Resources can expose accessors, attributes and map custom actions.

Application Module: AppModule View Object: EmployeesView

Resource: Employees

Structure Name:

- EmployeesView2
  - □ JobHistory

Resource Name: Employees

Rowfinder Key: <None>

□ Canonical Link

○ Internal

○ External

□ Include Discriminator Subtype

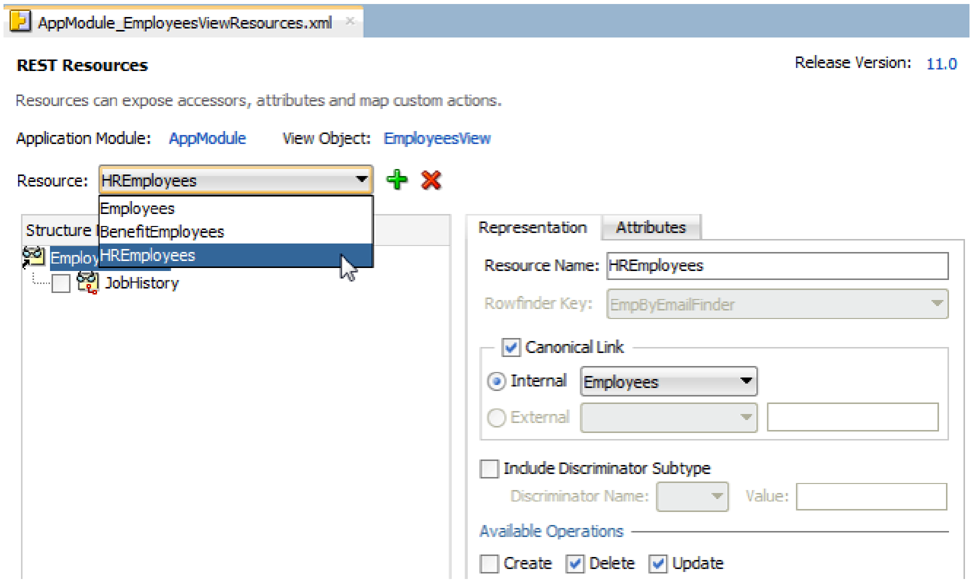
Discriminator Name: Value:

Available Operations

□ Create ☑ Delete ☑ Update

As Figure 22-8 shows, the `EmployeesView` view object has multiple resources defined in version `11.0` of the `Employees` root resource. The `HREmployees` and `BenefitsEmployees` resource are backed by unique view object shaping definitions and each of these resource defines the `Employees` resource as its canonical resource.

Figure 22-8 REST Sample Project - Canonical Employee Resource Version 11.0



AppModule_EmployeesViewResources.xml

REST Resources

Release Version: 11.0

Resources can expose accessors, attributes and map custom actions.

Application Module: AppModule View Object: EmployeesView

Resource: HREmployees

Resource Name: HREmployees

Rowfinder Key: EmpByEmailFinder

☑ Canonical Link

● Internal Employees

○ External

□ Include Discriminator Subtype

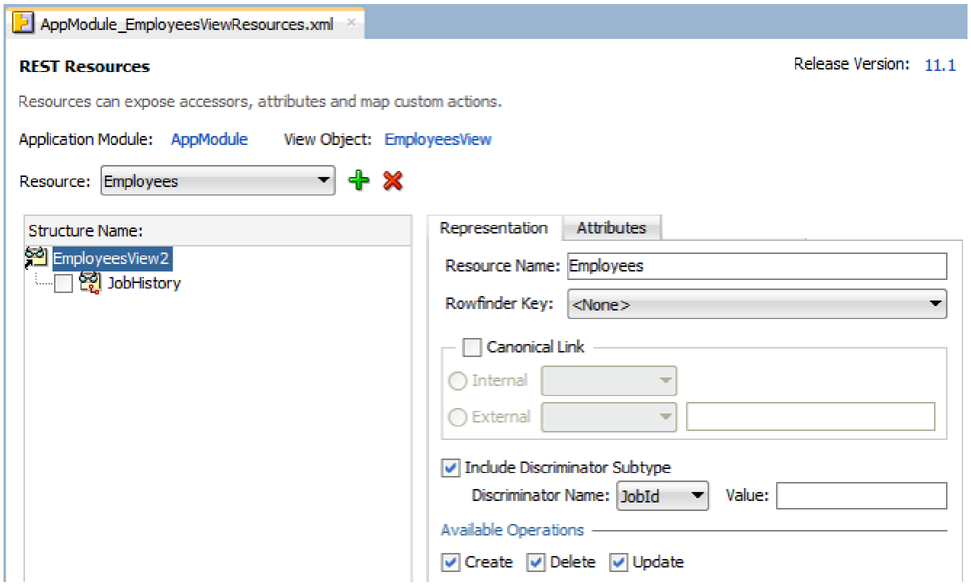
Discriminator Name: Value:

Available Operations

□ Create ☑ Delete ☑ Update

As Figure 22-9 shows, version 11.1 of the Employees root resource adds functionality, including a Create operation to enable creating resource items using the Jobs resource for the LOV-enabled EmployeesView attribute JobId. At runtime, the Jobs resource will supply values for the LOV-enabled attribute JobId without requiring the context of a row. This version also enables subtype usages in the Employees resource collection, where the JobId attribute serves as the discriminator for the subtype resource. At runtime, the Employees resource displays the commission attribute for employees with the SA_REP (Sales Representative) job ID.

Figure 22-9 REST Sample Project - Employees Resource Version 11.1


AppModule_EmployeesViewResources.xml

REST Resources

Release Version: 11.1

Resources can expose accessors, attributes and map custom actions.

Application Module: AppModule View Object: EmployeesView

Resource: Employees

Structure Name:
- EmployeesView2
  - □ JobHistory

Representation | Attributes

Resource Name: Employees

Rowfinder Key: <None>

□ Canonical Link
○ Internal
○ External

☑ Include Discriminator Subtype

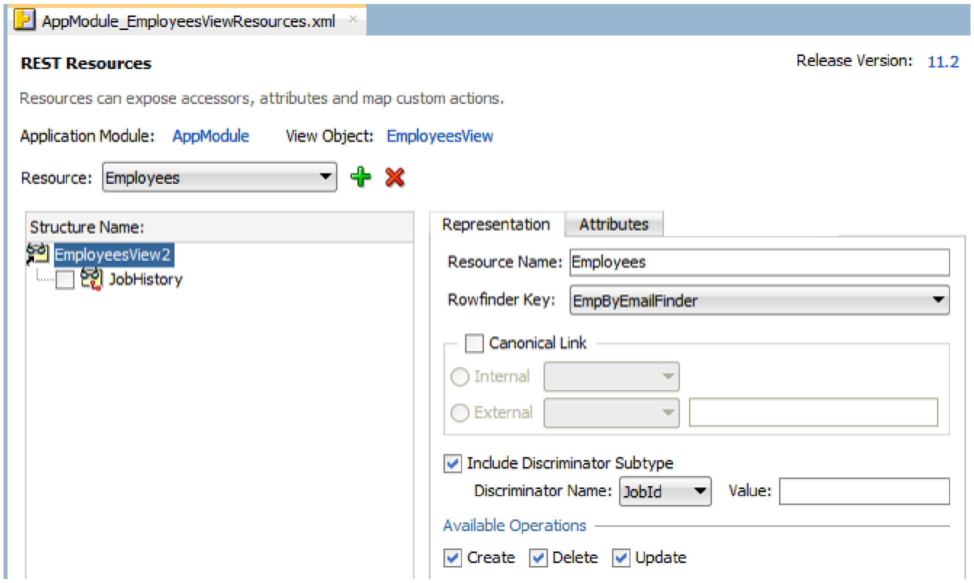
Discriminator Name: JobId Value:

Available Operations

☑ Create ☑ Delete ☑ Update

As Figure 22-10 shows, version 11.2 of the Employees root resource adds the EmpByEmailFinder row finder to enable locating employees by their email address.

Figure 22-10 REST Sample Project - Employees Resource Version 11.2


AppModule_EmployeesViewResources.xml

REST Resources

Release Version: 11.2

Resources can expose accessors, attributes and map custom actions.

Application Module: AppModule View Object: EmployeesView

Resource: Employees

Structure Name:
- EmployeesView2
  - □ JobHistory

Representation | Attributes

Resource Name: Employees

Rowfinder Key: EmpByEmailFinder

□ Canonical Link
○ Internal
○ External

☑ Include Discriminator Subtype

Discriminator Name: JobId Value:

Available Operations

☑ Create ☑ Delete ☑ Update

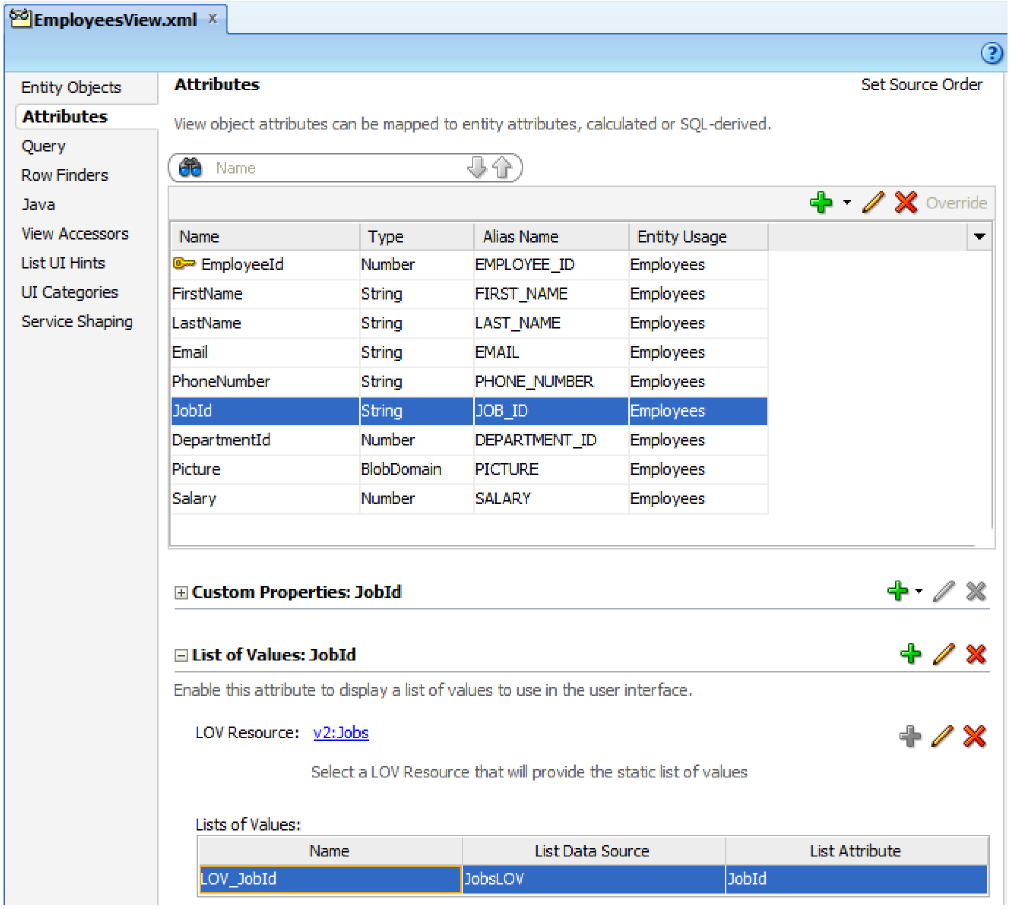
Entity Object and View Object Customization

AsFigure 22-11 shows, the `EmployeesView` view object defines the LOV-enabled attribute `JobId`. The LOV accessor resource returns the list of jobs by title, and presents them as choices for the corresponding job ID.

Note that JDeveloper intentionally does not expose LOV accessors in the resource definition. At runtime the ADF REST runtime automatically nests LOV accessor resources as children of the resource item's LOV-enabled attribute.

Additionally, Figure 22-11 shows support for creating a new resource item with a LOV-enabled attribute defined. The `EmployeesView` associates a specific version of an LOV accessor resource (identified as `v2:Jobs`) with the LOV-enabled attribute `JobId`. At runtime, the `Employees` resource describe exposes a link to the LOV resource. The POST request to create a resource item may be completed with the aid of the LOV resource, which relies on the view object LOV definition to return the correct values to the LOV-enabled attribute.

Figure 22-11   REST Sample Project - Employees LOV-Enabled Attribute



EmployeesView.xml

Entity Objects
Attributes
Query
Row Finders
Java
View Accessors
List UI Hints
UI Categories
Service Shaping

Attributes   Set Source Order

View object attributes can be mapped to entity attributes, calculated or SQL-derived.

Name   Override

Name	Type	Alias Name	Entity Usage
EmployeeId	Number	EMPLOYEE_ID	Employees
FirstName	String	FIRST_NAME	Employees
LastName	String	LAST_NAME	Employees
Email	String	EMAIL	Employees
PhoneNumber	String	PHONE_NUMBER	Employees
JobId	String	JOB_ID	Employees
DepartmentId	Number	DEPARTMENT_ID	Employees
Picture	BlobDomain	PICTURE	Employees
Salary	Number	SALARY	Employees

⊞ **Custom Properties: JobId**

⊟ **List of Values: JobId**

Enable this attribute to display a list of values to use in the user interface.

LOV Resource: v2:Jobs

Select a LOV Resource that will provide the static list of values

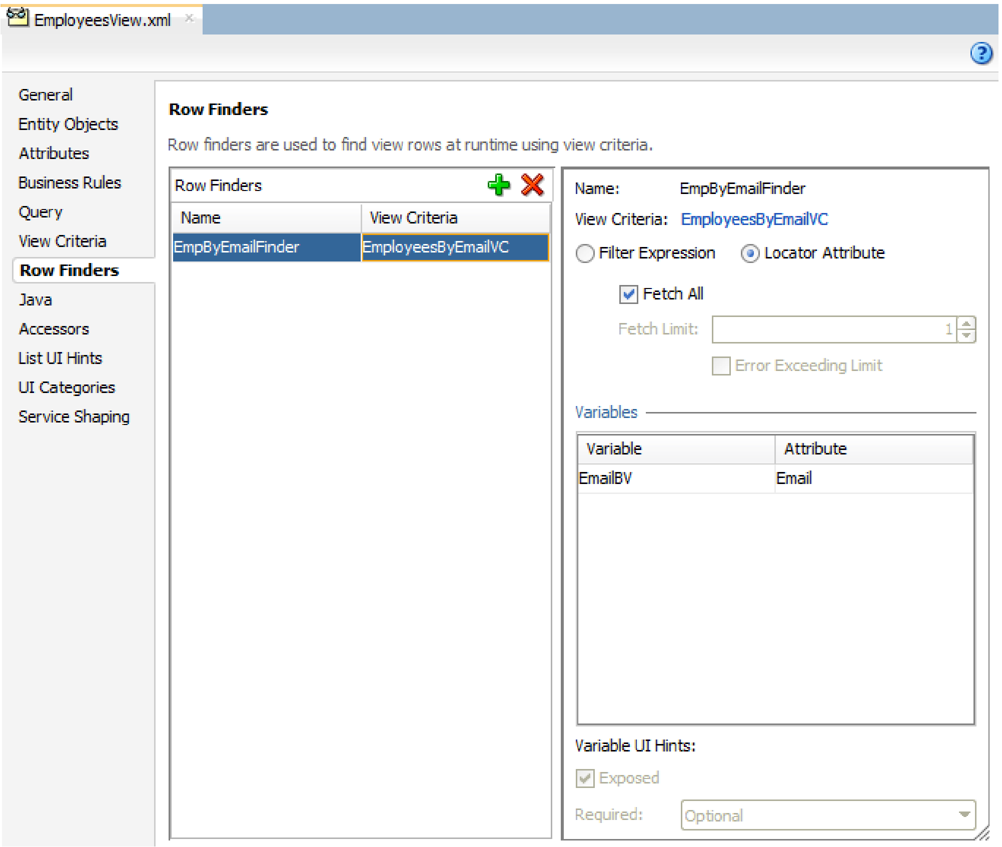
Lists of Values:

Name	List Data Source	List Attribute
LOV_JobId	JobsLOV	JobId

As Figure 22-12 shows, the `EmployeesView` view object defines the `EmpByEmailFinder` row finder. The row finder retrieves the employee using the employee's email address (`Email` attribute) instead of the employee's ID.

Note that row finders are exposed on all versions of the resource by default. However, when the row finder key attribute is defined in a particular resource definition, then it will be mandatory to use the finder name in the URL resource path. Until the row finder key is explicitly defined in the resource definition, the URL resource path will require the key attribute (`EmployeeId` in this sample).

Figure 22-12 REST Sample Project - Employees Row Finder



Row Finders

Row finders are used to find view rows at runtime using view criteria.

Row Finders

Name	View Criteria
EmpByEmailFinder	EmployeesByEmailVC

Name: EmpByEmailFinder
View Criteria: EmployeesByEmailVC
○ Filter Expression ● Locator Attribute
☑ Fetch All
Fetch Limit: 1
□ Error Exceeding Limit

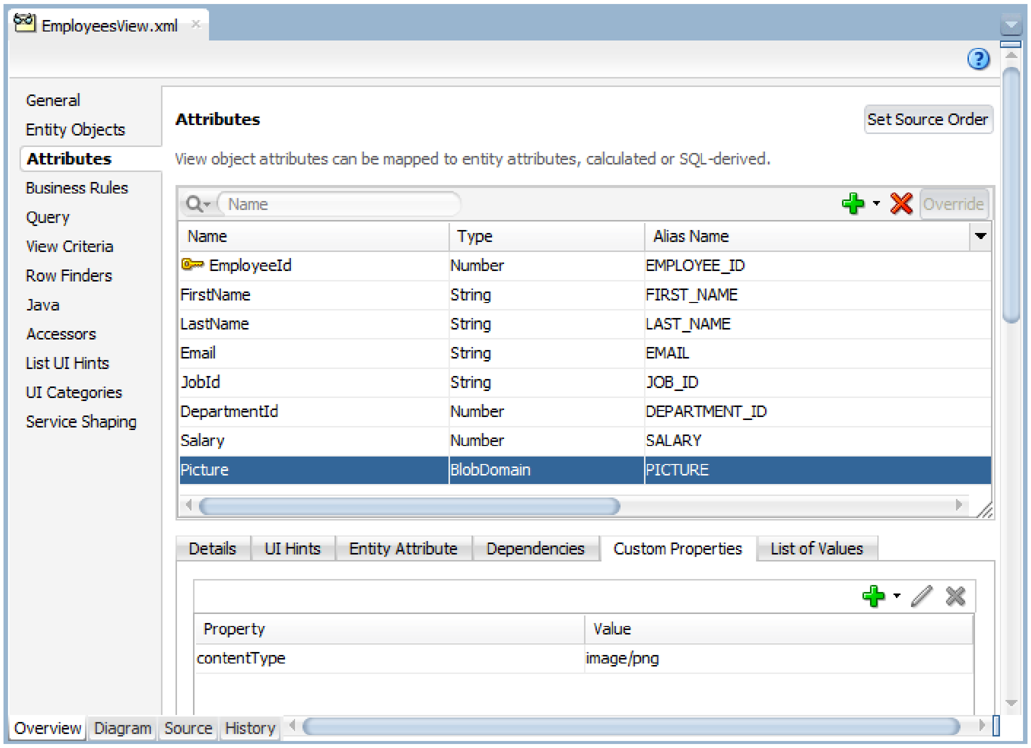
Variables

Variable	Attribute
EmailBV	Email

Variable UI Hints:
☑ Exposed
Required: Optional

As Figure 22-13 shows, the `EmployeesView` view object defines a custom content type `image/png` on the `Picture` attribute. This content type replaces the default `application/octet-stream` content type definition for the attribute and will be the expected accept type when streaming the image content.

Figure 22-13 REST Sample Project - Employees BLOB Content Type



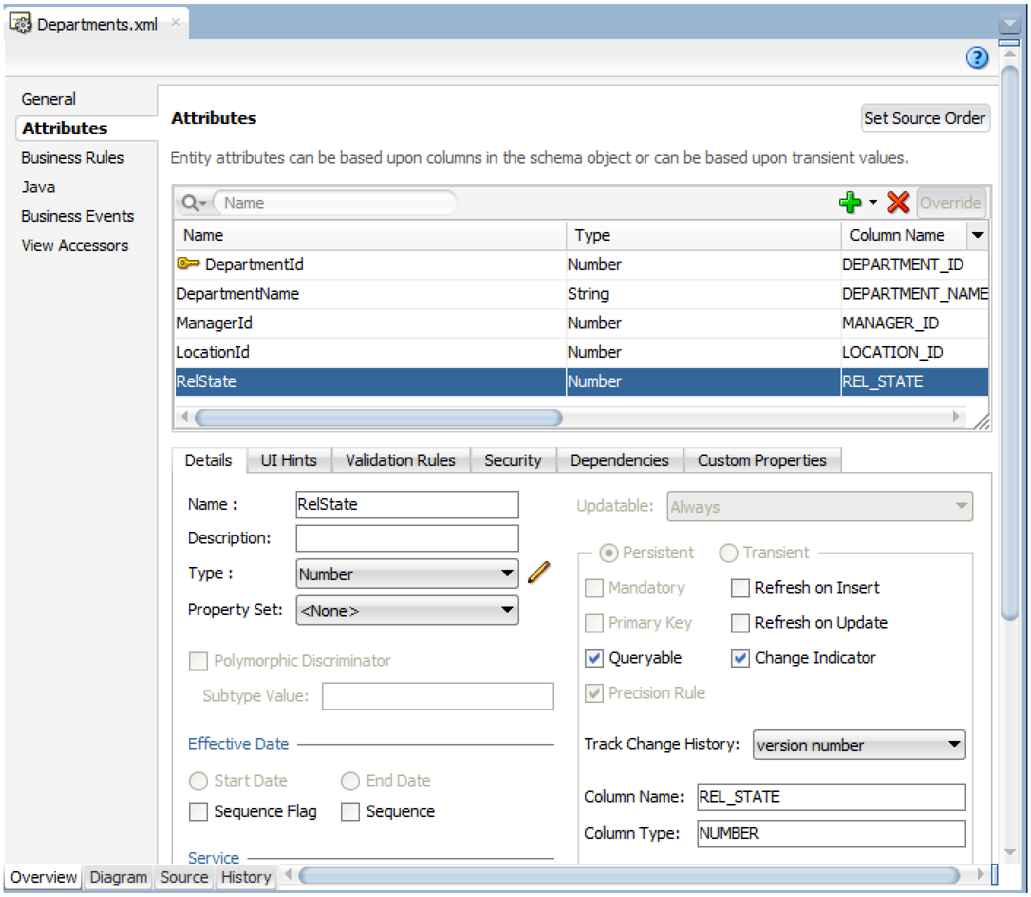
View object attributes can be mapped to entity attributes, calculated or SQL-derived.

Name	Type	Alias Name
EmployeeId	Number	EMPLOYEE_ID
FirstName	String	FIRST_NAME
LastName	String	LAST_NAME
Email	String	EMAIL
JobId	String	JOB_ID
DepartmentId	Number	DEPARTMENT_ID
Salary	Number	SALARY
Picture	BlobDomain	PICTURE

Property	Value
contentType	image/png

As Figure 22-14 shows, the `Departments` entity object defines the change-indicator attribute `RelState` to enable support for data consistency checking. The editor for the attributes has the **Change Indicator** and **History Column - version number** fields enabled to configure the change indicator.

Figure 22-14 REST Sample Project - Departments Change Indicator Attribute



Attributes

Entity attributes can be based upon columns in the schema object or can be based upon transient values.

Name	Type	Column Name
DepartmentId	Number	DEPARTMENT_ID
DepartmentName	String	DEPARTMENT_NAME
ManagerId	Number	MANAGER_ID
LocationId	Number	LOCATION_ID
RelState	Number	REL_STATE

Name : RelState

Description:

Type : Number

Property Set: <None>

□ Polymorphic Discriminator

Subtype Value:

Effective Date

○ Start Date ○ End Date

□ Sequence Flag □ Sequence

Service

Updatable: Always

● Persistent ○ Transient

□ Mandatory □ Refresh on Insert

□ Primary Key □ Refresh on Update

☑ Queryable ☑ Change Indicator

☑ Precision Rule

Track Change History: version number

Column Name: REL_STATE

Column Type: NUMBER

Understanding ADF Business Components Resources

With RESTful web services, ADF Business Components view objects are represented as a resource collection when the RESTful web service is invoked at runtime. ADF REST resources that service developers create in the ADF Business Components model project are based on view object instances that the project application module defines. For example:

- A `Departments` resource is based on a `DepartmentsVO` view instance and its accessor to an `EmployeesVO` view instance.
- An `Employees` resource is based on an `EmployeesVO` view instance.

At runtime, when an action of the RESTful web service is invoked, the payload returned by the service contains one or more resource collections, comprised of the row sets queried by the backing view objects and the individual attributes of the view object rows. The resource collections preserve the relationship of master-detail coordinating view instances.

As Table 22-1 shows, the *resource collection* is the RESTful web service payload representation of a view object instance. The *resource items* are the rows and attributes of the payload item object, which in turn correspond to view object rows.

> **Note:**
>
> The format of ADF resource collections and contained items are defined by specific ADF REST media types as JSON-encoded entities. For more details, see ADF REST Media Types.

Table 22-1 JSON Objects and ADF Business Component Representation

JSON Object	ADF Business Component
`resource collection`	View object instance comprised of one or more rows. `Departments`, `Employees` are examples of resource collections based on `DepartmentsVO` and `EmployeesVO` view instances.
`resource item`	A view object row. The specific department `10` or employee `1012` are examples of resource items from the `Departments` and `Employees` resource collections.

Retrieving the ADF REST Catalog Describe

When you invoke an HTTP GET method in RESTful web service to describe all the available resources in the resource catalog, it returns JSON objects that contain the attributes, actions, and links defined in the REST resource definitions for the web service.
The describe for the RESTful web service resource catalog allows you to identify the shape and actions allowed on an ADF RESTful web service defined for the service endpoint. By default the catalog describe request returns a JSON object that contains the information needed to understand all available public resources, including their attributes, collections, items, annotations, actions, and links.

The ADF REST framework supports the following catalog describe use cases:

- Retrieve the *full resource catalog describe*, including all resource details, such as their attributes, collections, items, annotations, actions, and links.
- Retrieve a *minimal resource catalog describe*, where the describe details will be limited to resource titles and links only and children, or nested resources, will be excluded.
- Retrieve a resource catalog describe (either full or minimal) based on resource visibility. For example, you can retrieve the describe for only public resources, only private (unlisted) resources, or both public and private resources.
- Retrieve a resource catalog describe (either full or minimal) but optionally exclude or include children resources nested within a parent resource and optionally exclude or include all resource annotations.

To retrieve the catalog describe of all the available, public resources of a specific version in the application, you append `/describe` to the version URL.

For example, the following URL returns the describe for the resource catalog of all available public resources designated as version `11.0` in the identified application:

```
http://host:port/context-root/rest/11.0/describe
```

Note:

Executing a GET request to retrieve a resource catalog describe requires the client to be in the context of a particular version. For information on how to get the version release names for the service end point, see Retrieving Resource Versions.

When you do not require an exhaustive describe that returns the full set of information for all resources in the same request, you can request a minimal describe. For example, the following URL with query parameter `metadataMode` set to `minimal` returns the describe for all parent resources but the retrieved describe information will be limited to just the titles and links of the resources:

```
http://host:port/context-root/rest/11.0/describe?metadataMode=minimal
```

Additionally, you can append URL query parameters on the request for a minimal catalog describe to retrieve specific details in the describe. For example, the following URL with appended query parameters retrieves a minimal catalog describe with all available children resources nested within their parent resources included.

```
http://host:port/context-root/rest/11.0/describe?
metadataMode=minimal&includeChildren=true
```

The following table identifies the URL query parameters the may be used with the catalog describe request. These query parameters let you control the amount of detail retrieved in the describe.

Table 22-2 Optional URL Query Parameters for Catalog Describe Requests

URL Query Parameter	Values	Description

`metadataMode`	`verbose` (default)
	`minimal`
	`list`
		Use to retrieve the description of resources with or without all details. By default, the full catalog is retrieved, performing an exhaustive describe that returns the complete set of information for all resources, including nested children resources. The full catalog describe therefore includes the following sections of information for each resource:
		title, attributes, collection, item, annotations, children, and links
		Note that annotations must be defined by the ADF REST resource developer in the application and may not be present on the resource.
		You can improve the readability of a large catalog and retrieve a shallow catalog limited to the titles and links of parent resources (does not include children resources) by appending the URL parameter `?metadataMode=minimal` to the describe request.
		If you do not want any metadata in the response but only self links, you can append `?metadataMode=list` to the describe request.
		Optionally, additional parameters may be appended to the minimal describe request to include children resources and / or resource annotations, as explained for the query parameters `showAnnotations` and `includeChildren`. For example, you can append `?metadataMode=minimal&includeChildren=true` to retrieve a minimal catalog describe with all children resources included.

`showAnnotations`	`true`, `false` (default depends on whether full or minimal catalog is retrieved)	Use to either exclude or include resource annotations, as specified in the application. The default depends on the `metadataMode` parameter value: • By default, a full catalog (`metadataMode=verbose`) describe includes annotations. To exclude resource annotations from the full catalog describe, you can append `?showAnnotations=false` on the describe request. • By default, a minimal catalog (`metadataMode=minimal`) describe excludes annotations. To include resource annotations in the minimal catalog describe, you can append `?showAnnotations=true` on the describe request. Note that annotations must be defined by the ADF REST resource developer in the application and may not be present on the resource You cannot use this parameter with `?metadataMode=list`.

Table 22-2 (Cont.) Optional URL Query Parameters for Catalog Describe Requests

URL Query Parameter	Values	Description
includeChildren	true, false (default depends on whether full or minimal catalog is retrieved)	Use to either exclude or include all available children resources nested within a parent resource. The default depends on the metadataMode parameter value: • By default, a full catalog (metadataMode=verbose) describe includes all children resources. To exclude children resources from the full catalog describe, you can append ?includeChildren=false on the describe request. • By default, a minimal catalog (metadataMode=minimal) describe excludes all children resources. To include children resources in the minimal catalog describe, you can append ?includeChildren=true on the describe request. You cannot use this parameter with ?metadataMode=list.
include	public (default), unlisted, all	Use to retrieve the description of resources of a certain visibility, as declared in the application. This URL parameter values specify including only public resources (public), only private resources (unlisted), or both types (all). By default, only resources that are public are shown in the describe. To view private resources, you must append either ?include=all or ?include=unlisted on the describe request. See How to Hide a Resource from the Catalog Describe.
resources	A comma separated list of resource collection names. Format:<resName1>,<resName2> Example: ?resources=Deparments,Employees	Use to retrieve the description of the named resources to filter resources at the catalog level.

Retrieving the Full Catalog Describe

ADF REST runtime supports describing all available resources for the application end point using a GET method and performs an exhaustive describe that returns the full set of information for all resources, including all children resources nested within parent resources.

To examine all available resources with full information included in the resource catalog:

1. Execute the resource catalog describe and locate the names of the resources in the describe. The children attribute identifies nested resources.

2. Examine these resource objects to understand the shape of each resource:
   - `attributes` specifies the list of available resource collection attributes.
   - `collection` specifies the shape of the collection and specifies `finders` for row finder, `links`, and available `actions` (including media types).
   - `item` specifies the shape of the instances of the collection and itself specifies `links` and available `actions`.
   - `children` specifies any nested resources (and itself contains `attributes`, `collection`, and `item` objects).

For example, the describe for a service with `Departments` and child `Employees` resources returns the following objects:

```
{
  "Resources" : {
    "Employees" : {
      "discrColumnType" : false,
      "attributes" : [ {
       ...
      } ],
      "collection" : {
        ...
        } ],
        "finders" : [ {
        } ]
        "links" : [ {
        } ]
        "actions" : [ {
        } ]
      },
      "item" : {
        "links" : [ {
        } ]
        "actions" : [ {
        } ]
      },
      "children" : {
        "Departments"
          ...
        } ],
      ...
      "links" : [ {
      } ]
  }
}
```

The following sample describes version `11.0` of all resources in the resource catalog. Note that the URL version parameter `11.0` may be replaced with an application-specific version identifier. For more details about specifying resource versions, see in Retrieving Resource Versions.

> **Note:**
>
> You can use the `include` query parameter to retrieve the full catalog describe with resources of a certain visibility. For example, to view both public and private resources, you can use`http://server/demo/rest/v1/describe?include=all`. You can use `all`, `unlisted`, or `public` as values for the `include` query parameter.

Request

- **URL**

  `http://server/demo/rest/11.0/describe`

- **HTTP Method**

  GET

- **Content-Type**

  none

- **Payload**

  none

Response

- **HTTP Code**

  `200`

- **Content-Type**

  `application/vnd.oracle.adf.description+json`

- **Payload**

```
{
  "Resources" : {
    "Employees" : {
      "discrColumnType" : false,
      "attributes" : [ {
        "name" : "EmployeeId",
        "type" : "integer",
        "updatable" : true,
        "mandatory" : true,
        "queryable" : true,
        "precision" : 6
      }, {
        "name" : "FirstName",
        "type" : "string",
        "updatable" : true,
        "mandatory" : false,
        "queryable" : true,
        "precision" : 20
      }, {
        "name" : "LastName",
        "type" : "string",
        "updatable" : true,
        "mandatory" : true,
        "queryable" : true,
        "precision" : 25
```

```
}, {
  "name" : "Email",
  "type" : "string",
  "updatable" : true,
  "mandatory" : true,
  "queryable" : true,
  "precision" : 25
}, {
  "name" : "JobId",
  "type" : "string",
  "updatable" : true,
  "mandatory" : true,
  "queryable" : true,
  "precision" : 10,
  "controlType" : "choice",
  "maxLength" : "10",
  "lov" : {
    "childRef" : "JobsLOV",
    "attributeMap" : [ {
      "source" : "JobTitle",
      "target" : "JobId"
    } ],
    "displayAttributes" : [ "JobTitle" ]
  }
}, {
  "name" : "DepartmentId",
  "type" : "integer",
  "updatable" : true,
  "mandatory" : false,
  "queryable" : true,
  "precision" : 4
}, {
  "name" : "Salary",
  "type" : "number",
  "updatable" : true,
  "mandatory" : false,
  "queryable" : true,
  "precision" : 8,
  "scale" : 2
}, {
  "name" : "Picture",
  "type" : "attachment",
  "updatable" : true,
  "mandatory" : false,
  "queryable" : false,
  "actions" : [ {
    "name" : "delete",
    "method" : "DELETE"
  }, {
    "name" : "get",
    "method" : "GET",
    "responseType" : [ "image/png" ]
  } ]
} ],
"collection" : {
  "rangeSize" : 24,
  "finders" : [ {
    "name" : "EmpByEmailFinder",
    "title" : "EmployeesByEmailVC",
    "attributes" : [ {
      "name" : "Email",
            "type" : "string",
            "updatable" : true,
            "required" : "Optional",
            "queryable" : false
          } ]
        }, {
          "name" : "PrimaryKey",
          "attributes" : [ {
            "name" : "EmployeeId",
            "type" : "integer",
            "updatable" : true,
            "mandatory" : true,
            "queryable" : true,
            "precision" : 6
          } ]
        } ],
        "links" : [ {
          "rel" : "self",
          "href" : "http://server/demo/rest/11.0/Employees",
          "name" : "self",
          "kind" : "collection"
        } ],
        "actions" : [ {
          "name" : "get",
          "method" : "GET",
          "responseType" : [ "application/json", "application/
vnd.oracle.adf.resourcecollection+json" ]
        }, {
          "name" : "create",
          "method" : "POST",
          "requestType" : [ "application/vnd.oracle.adf.resourceitem+json" ],
          "responseType" : [ "application/json", "application/
vnd.oracle.adf.resourceitem+json" ]
        } ]
      },
      "item" : {
        "links" : [ {
          "rel" : "lov",
          "href" : "http://server/demo/rest/11.0/Employees/{id}/lov/JobsLOV",
          "name" : "JobsLOV",
          "kind" : "collection"
        }, {
          "rel" : "self",
          "href" : "http://server/demo/rest/11.0/Employees/{id}",
          "name" : "self",
          "kind" : "item"
        }, {
          "rel" : "canonical",
          "href" : "http://server/demo/rest/11.0/Employees/{id}",
          "name" : "canonical",
          "kind" : "item"
        }, {
          "rel" : "enclosure",
          "href" : "http://server/demo/rest/11.0/Employees/enclosure/
Picture",
          "name" : "Picture",
          "kind" : "other"
        } ],
        "actions" : [ {
          "name" : "get",
          "method" : "GET",
     "responseType" : [ "application/json", "application/
vnd.oracle.adf.resourceitem+json" ]
    }, {
     "name" : "delete",
     "method" : "DELETE"
    }, {
     "name" : "multiplySalary",
     "parameters" : [ {
      "name" : "multiplicand",
      "type" : "number",
      "mandatory" : false
     } ],
     "resultType" : "number",
     "method" : "POST",
     "requestType" : [ "application/vnd.oracle.adf.action+json" ],
     "responseType" : [ "application/json", "application/
vnd.oracle.adf.actionresult+json" ]
    } ]
   },
   "children" : {
    "JobsLOV" : {
     "discrColumnType" : false,
     "attributes" : [ {
      "name" : "JobId",
      "type" : "string",
      "updatable" : false,
      "mandatory" : true,
      "queryable" : true,
      "precision" : 10
     }, {
      "name" : "JobTitle",
      "type" : "string",
      "updatable" : false,
      "mandatory" : true,
      "queryable" : true,
      "precision" : 35
     }, {
      "name" : "MinSalary",
      "type" : "integer",
      "updatable" : false,
      "mandatory" : false,
      "queryable" : true,
      "precision" : 6
     }, {
      "name" : "MaxSalary",
      "type" : "integer",
      "updatable" : false,
      "mandatory" : false,
      "queryable" : true,
      "precision" : 6
     } ],
     "collection" : {
      "rangeSize" : 0,
      "finders" : [ {
       "name" : "PrimaryKey",
       "attributes" : [ {
        "name" : "JobId",
        "type" : "string",
        "updatable" : false,
        "mandatory" : true,
        "queryable" : true,
                "precision" : 10
              } ]
            } ],
            "links" : [ {
              "rel" : "self",
              "href" : "http://server/demo/rest/11.0/Employees/{id}/child/
JobsLOV",
              "name" : "self",
              "kind" : "collection"
            } ],
            "actions" : [ {
              "name" : "get",
              "method" : "GET",
              "responseType" : [ "application/json", "application/
vnd.oracle.adf.resourcecollection+json" ]
            }, {
              "name" : "create",
              "method" : "POST",
              "requestType" : [ "application/
vnd.oracle.adf.resourceitem+json" ],
              "responseType" : [ "application/json", "application/
vnd.oracle.adf.resourceitem+json" ]
            } ]
          },
          "item" : {
            "links" : [ {
              "rel" : "self",
              "href" : "http://server/demo/rest/11.0/Employees/{id}/child/
JobsLOV/{id}",
              "name" : "self",
              "kind" : "item"
            }, {
              "rel" : "parent",
              "href" : "http://server/demo/rest/11.0/Employees/{id}",
              "name" : "parent",
              "kind" : "item"
            }, {
              "rel" : "canonical",
              "href" : "http://server/demo/rest/11.0/Employees/{id}/child/
JobsLOV/{id}",
              "name" : "canonical",
              "kind" : "item"
            } ],
            "actions" : [ {
              "name" : "get",
              "method" : "GET",
              "responseType" : [ "application/json", "application/
vnd.oracle.adf.resourceitem+json" ]
            }, {
              "name" : "update",
              "method" : "PATCH",
              "requestType" : [ "application/
vnd.oracle.adf.resourceitem+json" ],
              "responseType" : [ "application/json", "application/
vnd.oracle.adf.resourceitem+json" ]
            }, {
              "name" : "delete",
              "method" : "DELETE"
            } ]
          },
          "links" : [ {
            "rel" : "self",
            "href" : "http://server/demo/rest/11.0/Employees/{id}/child/
JobsLOV/describe",
            "name" : "self",
            "kind" : "describe"
          }, {
            "rel" : "canonical",
            "href" : "http://server/demo/rest/11.0/Employees/{id}/child/
JobsLOV/describe",
            "name" : "canonical",
            "kind" : "describe"
          } ]
        }
      },
      "links" : [ {
        "rel" : "self",
        "href" : "http://server/demo/rest/11.0/Employees/describe",
        "name" : "self",
        "kind" : "describe"
      }, {
        "rel" : "canonical",
        "href" : "http://server/demo/rest/11.0/Employees/describe",
        "name" : "canonical",
        "kind" : "describe"
      } ]
    },
    "Departments" : {
      "discrColumnType" : false,
      "attributes" : [ {
        "name" : "DepartmentId",
        "type" : "integer",
        "updatable" : true,
        "mandatory" : true,
        "queryable" : true,
        "precision" : 4
      }, {
        "name" : "DepartmentName",
        "type" : "string",
        "updatable" : true,
        "mandatory" : true,
        "queryable" : true,
        "precision" : 30
      }, {
        "name" : "RelState",
        "type" : "integer",
        "updatable" : true,
        "mandatory" : false,
        "queryable" : true
      } ],
      "collection" : {
        "rangeSize" : 24,
        "finders" : [ {
          "name" : "PrimaryKey",
          "attributes" : [ {
            "name" : "DepartmentId",
            "type" : "integer",
            "updatable" : true,
            "mandatory" : true,
            "queryable" : true,
            "precision" : 4
          } ]
        } ],
        "links" : [ {
          "rel" : "self",
          "href" : "http://server/demo/rest/11.0/Departments",
          "name" : "self",
          "kind" : "collection"
        } ],
        "actions" : [ {
          "name" : "get",
          "method" : "GET",
          "responseType" : [ "application/json", "application/
vnd.oracle.adf.resourcecollection+json" ]
        }, {
          "name" : "create",
          "method" : "POST",
          "requestType" : [ "application/vnd.oracle.adf.resourceitem+json" ],
          "responseType" : [ "application/json", "application/
vnd.oracle.adf.resourceitem+json" ]
        }, {
          "name" : "testImpl",
          "parameters" : [ {
            "name" : "testid",
            "type" : "string",
            "mandatory" : false
          } ],
          "resultType" : "string",
          "method" : "POST",
          "requestType" : [ "application/vnd.oracle.adf.action+json" ],
          "responseType" : [ "application/json", "application/
vnd.oracle.adf.actionresult+json" ]
        } ]
      },
      "item" : {
        "links" : [ {
          "rel" : "child",
          "href" : "http://server/demo/rest/11.0/Departments/{id}/child/
Employees",
          "name" : "Employees",
          "kind" : "collection",
          "cardinality" : {
            "value" : "1 to *",
            "sourceAttributes" : "DepartmentId",
            "destinationAttributes" : "DepartmentId"
          }
        }, {
          "rel" : "self",
          "href" : "http://server/demo/rest/11.0/Departments/{id}",
          "name" : "self",
          "kind" : "item"
        }, {
          "rel" : "canonical",
          "href" : "http://server/demo/rest/11.0/Departments/{id}",
          "name" : "canonical",
          "kind" : "item"
        } ],
        "actions" : [ {
          "name" : "get",
          "method" : "GET",
          "responseType" : [ "application/json", "application/
vnd.oracle.adf.resourceitem+json" ]
        }, {
          "name" : "update",
          "method" : "PATCH",
          "requestType" : [ "application/vnd.oracle.adf.resourceitem+json" ],
          "responseType" : [ "application/json", "application/
vnd.oracle.adf.resourceitem+json" ]
        }, {
          "name" : "delete",
          "method" : "DELETE"
        } ]
      },
      "children" : {
        "Employees" : {
          "discrColumnType" : false,
          "attributes" : [ {
            "name" : "EmployeeId",
            "type" : "integer",
            "updatable" : true,
            "mandatory" : true,
            "queryable" : true,
            "precision" : 6
          }, {
            "name" : "FirstName",
            "type" : "string",
            "updatable" : true,
            "mandatory" : false,
            "queryable" : true,
            "precision" : 20
          }, {
            "name" : "LastName",
            "type" : "string",
            "updatable" : true,
            "mandatory" : true,
            "queryable" : true,
            "precision" : 25
          }, {
            "name" : "Email",
            "type" : "string",
            "updatable" : true,
            "mandatory" : true,
            "queryable" : true,
            "precision" : 25
          }, {
            "name" : "JobId",
            "type" : "string",
            "updatable" : true,
            "mandatory" : true,
            "queryable" : true,
            "precision" : 10,
            "controlType" : "choice",
            "maxLength" : "10",
            "lov" : {
              "childRef" : "JobsLOV",
              "attributeMap" : [ {
                "source" : "JobTitle",
                "target" : "JobId"
              } ],
              "displayAttributes" : [ "JobTitle" ]
            }
          }, {
            "name" : "DepartmentId",
            "type" : "integer",
            "updatable" : true,
            "mandatory" : false,
            "queryable" : true,
            "precision" : 4
          }, {
            "name" : "Salary",
            "type" : "number",
            "updatable" : true,
            "mandatory" : false,
            "queryable" : true,
            "precision" : 8,
            "scale" : 2
          }, {
            "name" : "Picture",
            "type" : "attachment",
            "updatable" : true,
            "mandatory" : false,
            "queryable" : false,
            "actions" : [ {
              "name" : "delete",
              "method" : "DELETE"
            }, {
              "name" : "get",
              "method" : "GET",
              "responseType" : [ "image/png" ]
            } ]
          } ],
          "collection" : {
            "rangeSize" : 0,
            "finders" : [ {
              "name" : "EmpByEmailFinder",
              "title" : "EmployeesByEmailVC",
              "attributes" : [ {
                "name" : "Email",
                "type" : "string",
                "updatable" : true,
                "required" : "Optional",
                "queryable" : false
              } ]
            }, {
              "name" : "PrimaryKey",
              "attributes" : [ {
                "name" : "EmployeeId",
                "type" : "integer",
                "updatable" : true,
                "mandatory" : true,
                "queryable" : true,
                "precision" : 6
              } ]
            } ],
            "links" : [ {
              "rel" : "self",
              "href" : "http://server/demo/rest/11.0/Departments/{id}/child/
Employees",
              "name" : "self",
              "kind" : "collection"
            } ],
            "actions" : [ {
              "name" : "get",
              "method" : "GET",
              "responseType" : [ "application/json", "application/
```

```
vnd.oracle.adf.resourcecollection+json" ]
      }, {
       "name" : "create",
       "method" : "POST",
       "requestType" : [ "application/
vnd.oracle.adf.resourceitem+json" ],
       "responseType" : [ "application/json", "application/
vnd.oracle.adf.resourceitem+json" ]
      } ]
     },
     "item" : {
      "links" : [ {
       "rel" : "self",
       "href" : "http://server/demo/rest/11.0/Departments/{id}/child/
Employees/{id}",
       "name" : "self",
       "kind" : "item"
      }, {
       "rel" : "parent",
       "href" : "http://server/demo/rest/11.0/Departments/{id}",
       "name" : "parent",
       "kind" : "item"
      }, {
       "rel" : "canonical",
       "href" : "http://server/demo/rest/11.0/Departments/{id}/child/
Employees/{id}",
       "name" : "canonical",
       "kind" : "item"
      }, {
       "rel" : "enclosure",
       "href" : "http://server/demo/rest/11.0/Departments/{id}/child/
Employees/{id}/enclosure/Picture",
       "name" : "Picture",
       "kind" : "other"
      } ],
      "actions" : [ {
       "name" : "get",
       "method" : "GET",
       "responseType" : [ "application/json", "application/
vnd.oracle.adf.resourceitem+json" ]
      }, {
       "name" : "update",
       "method" : "PATCH",
       "requestType" : [ "application/
vnd.oracle.adf.resourceitem+json" ],
       "responseType" : [ "application/json", "application/
vnd.oracle.adf.resourceitem+json" ]
      }, {
       "name" : "delete",
       "method" : "DELETE"
      }, {
       "name" : "multiplySalary",
       "parameters" : [ {
        "name" : "multiplicand",
        "type" : "number",
        "mandatory" : false
       } ],
       "resultType" : "number",
       "method" : "POST",
       "requestType" : [ "application/vnd.oracle.adf.action+json" ],
       "responseType" : [ "application/json", "application/
```

```
vnd.oracle.adf.actionresult+json" ]
            } ]
          },
          "links" : [ {
            "rel" : "self",
            "href" : "http://server/demo/rest/11.0/Departments/{id}/child/
Employees/describe",
            "name" : "self",
            "kind" : "describe"
          }, {
            "rel" : "canonical",
            "href" : "http://server/demo/rest/11.0/Departments/{id}/child/
Employees/describe",
            "name" : "canonical",
            "kind" : "describe"
          } ]
        }
      },
      "links" : [ {
        "rel" : "self",
        "href" : "http://server/demo/rest/11.0/Departments/describe",
        "name" : "self",
        "kind" : "describe"
      }, {
        "rel" : "canonical",
        "href" : "http://server/demo/rest/11.0/Departments/describe",
        "name" : "canonical",
        "kind" : "describe"
      } ]
    }
  }
}
```

Retrieving a Minimal Catalog Describe

ADF REST runtime supports describing all available resources while retrieving a reduced amount of information for the application end point using a GET method. The reduced or minimal catalog describe helps improve the readability of the describe by limiting the resource information to resource titles, links, and available annotations.

To examine the minimal describe for all available resources in the resource catalog:

1. Execute the minimal resource catalog describe and locate the names of the resources in the describe. Note that nested resources or children resources are not shown by default.
2. Examine these resource objects `links`.

For example, the minimal describe for a service with a `Departments` resource returns the following objects:

```
{
  "Resources" : {
    "Departments" : {
      Departments,
      "links" : [ {
        "rel" : "self",
        "href" : "http://server/demo/rest/11.1/Departments/describe",
        "name" : "self",
        "kind" : "describe"
      } ]
    },
      ...
  }
}
```

By default children resources are not included in the minimal describe. Use the `includeChildren` query parameter to retrieve the minimal catalog describe with all available children resources nested within the parent resources. For example, to view children resources in the minimal describe, you can use a request like the following:

```
http://server/demo/rest/11.1/describe?
metadataMode=minimal&includeChildren=true
```

The minimal describe with the `includeChildren` query parameter set to `true` for a service with a `Departments` resource that includes a child resource `Employees` returns the following objects:

```
{
  "Resources" : {
    "Departments" : {
      Departments,
     "children" :  {
          "Employees" :  {
            "links" : [ {
              "rel" : "self",
              "href" : "http://server/demo/rest/11.1/Departments/{id}/child/
Employees/describe",
              "name" : "self",
              "kind" : "describe"
            } ]
         }
       },
      "links" : [ {
        "rel" : "self",
        "href" : "http://server/demo/rest/11.1/Departments/describe",
        "name" : "self",
        "kind" : "describe"
      } ]
    },
      ...
  }
}
```

The following sample retrieves a minimal resource catalog describe, including children resources, where the `Employees` resource is nested within the `Departments` resource and the `JobsLOV` resource is nested in the Employees resource.

Request

- **URL**

  `http://server/demo/rest/11.1/describe?`
  `metadataMode=minimal&includeChildren=true`

- **HTTP Method**

  GET

- **Content-Type**

  none

- **Payload**

  none

Response

- **HTTP Code**

  `200`

- **Content-Type**

  `application/vnd.oracle.adf.description+json`

- **Payload**

```
{
  "Resources" :  {
    "Employees" :  {
        "children" :  {
          "JobsLOV" :  {
            "links" : [ {
              "rel" : "self",
              "href" : "http://server/demo/rest/11.1/Employees/{id}/child/
JobsLOV/describe",
              "name" : "self",
              "kind" : "describe"
            } ]
         }
       },
       "links" : [ {
          "rel" : "self",
          "href" : "http://server/demo/rest/11.1/Employees/describe",
          "name" : "self",
          "kind" : "describe"
        } ]
      }
    },
    "Departments" :  {
       "children" :  {
         "Employees" :  {
            "links" : [
            {
              "rel" : "self",
              "href" : "http://server/demo/rest/11.1/Departments/{id}/child/
Employees/describe",
              "name" : "self",
              "kind" : "describe"
            } ]
        }
      },
       "links" : [
        {
          "rel" : "self",
          "href" : "http://server/demo/rest/11.1/Departments/describe",
          "name" : "self",
          "kind" : "describe"
        }
      ]
    }
  }
}
```

Retrieving the Catalog Describe Based on Resource Visibility Declaration

ADF REST runtime supports describing all available resources with a URL parameter that includes resources of a certain visibility for the application end point using a GET method, optionally including those resources that the application identifies as private.

By default, only resources that are public are shown in the describe. This URL parameter can specify only public resources, only private (unlisted) resources, or both types. To view private resources, you must append either `?include=all` (both public and private resources) or `?include=unlisted` (only private resources) on the describe request.

For more information about how the application developer declares visibility in the resource definition, see How to Hide a Resource from the Catalog Describe.

To view private resources in the resource catalog:

1. Execute the resource catalog describe with the URL parameter `?include=all` or `?include=unlisted` and locate the names of the private resources in the describe.
2. Examine these resource objects to understand the shape of each resource:

The following sample retrieves a minimal resource catalog describe, including both public and private resources, where the `Locations` resource is configured as `unlisted`.

Request

- **URL**

  `http://server/demo/rest/11.1/describe?metadataMode=minimal&include=all`
- **HTTP Method**

  GET
- **Content-Type**

  none
- **Payload**

  none

Response

- **HTTP Code**

  `200`
- **Content-Type**

  `application/vnd.oracle.adf.description+json`
- **Payload**

```
{
  "Resources" :  {
     "Employees" :  {
       "links" : [
        {
           "rel" : "self",
           "href" : "http://server/demo/rest/11.1/Employees/describe",
           "name" : "self",
           "kind" : "describe"
        }
       ]
     },
     "Departments" :  {
       "links" : [
         {
           "rel" : "self",
           "href" : "http://server/demo/rest/11.1/Departments/describe",
           "name" : "self",
           "kind" : "describe"
        }
       ]
     },
     "Locations" :  {
       "links" : [
        {
          "rel" : "self",
          "href" : "http://server/demo/rest/11.1/Locations/describe",
          "name" : "self",
          "kind" : "describe"
        }
      ]
    }
  }
}
```

Retrieving the ADF REST Resource Describe

When you invoke an HTTP GET method in RESTful web service to describe a single resource, all the available resources, or the nested resources, it returns a JSON object that contains the attributes, actions, and links defined in the REST resource definition. The describe for the RESTful web service allows you to identify the shape and actions allowed on an ADF RESTful web service. It returns a JSON object that contains the attributes, actions, and links defined in the REST resource definition.

The ADF REST framework supports the following describe use cases for the service end point:

- Describe a single resource collection.
- Describe a single resource item.
- Describe a nested resource in a parent-child relationship.
- Describe two or more named resource collections.
- Describe all available resources (resource catalog). For details, see Retrieving the ADF REST Catalog Describe.

To retrieve the describe, invoke an HTTP GET with `/describe` appended to the resource URL.

> **Note:**
>
> Executing a GET request to retrieve a resource describe also requires the client to be in the context of a particular version. For information on how to get the version release names for the service end point, see Retrieving Resource Versions.

For example, the following URL returns the describe for the `Employees` resource of version `11.0` in the identified application:

```
http://host:port/context-root/rest/11.0/Employees/describe
```

To retrieve the describe of all the resources of a specific version in the application, you append `/describe` to the version URL.

For example, the following URL returns the describe for all resources of version `11.0` in the identified application:

```
http://host:port/context-root/rest/11.0/describe
```

Describing a Resource Collection

The ADF REST runtime supports describing resource collections using a GET method.

To examine a resource collection:

1. Execute the resource collection describe and locate the names of the resources in the describe.
2. Examine these resource objects to understand the shape of each resource:
   - `attributes` specifies the list of available resource collection attributes.
   - `collection` specifies the shape of the collection and specifies `finder` for row finder, `links`, and available `actions` (including media types).
   - `item` specifies the shape of the instances of the collection and itself specifies `links` and available `actions`.
   - `children` specifies any nested resources (and itself contains `attributes`, `collection`, and `item` objects).

For example, the describe for the `Departments` resource returns the following objects:

```
{
  "Resources" : {
    "Departments" : {
      "discrColumnType" : false,
      "attributes" : [ {
       ...
      } ],
      "collection" : {
        ...
        } ],
        "finders" : [ {
        } ]
        "links" : [ {
        } ]
        "actions" : [ {
        } ]
      },
      "item" : {
        "links" : [ {
        } ]
        "actions" : [ {
        } ]
      },
      "children" : {
        "Employees"
          ...
        } ],
      ...
      "links" : [ {
      } ]
  }
}
```

The following sample describes version `11.0` of the `Departments` resource.

Request

- **URL**

  `http://server/demo/rest/11.0/Departments/describe`

- **HTTP Method**

  GET

- **Content-Type**

  none

- **Payload**

  none

Response

- **HTTP Code**

  `200`

- **Content-Type**

  `application/vnd.oracle.adf.description+json`

- **Payload**

```
{
  "Resources" : {
    "Departments" : {
      "discrColumnType" : false,
      "attributes" : [ {
        "name" : "DepartmentId",
        "type" : "integer",
        "updatable" : true,
        "mandatory" : true,
        "queryable" : true,
        "precision" : 4
      }, {
        "name" : "DepartmentName",
        "type" : "string",
        "updatable" : true,
        "mandatory" : true,
        "queryable" : true,
        "precision" : 30
      }, {
        "name" : "RelState",
        "type" : "integer",
        "updatable" : true,
        "mandatory" : false,
        "queryable" : true
      } ],
      "collection" : {
        "rangeSize" : 25,
        "finders" : [ {
          "name" : "PrimaryKey",
          "attributes" : [ {
            "name" : "DepartmentId",
            "type" : "integer",
            "updatable" : true,
            "mandatory" : true,
            "queryable" : true,
            "precision" : 4
          } ]
        } ],
        "links" : [ {
          "rel" : "self",
          "href" : "http://server/demo/rest/11.0/Departments",
          "name" : "self",
          "kind" : "collection"
        } ],
        "actions" : [ {
          "name" : "get",
          "method" : "GET",
          "responseType" : [ "application/json", "application/
vnd.oracle.adf.resourcecollection+json" ]
        }, {
          "name" : "create",
          "method" : "POST",
          "requestType" : [ "application/vnd.oracle.adf.resourceitem+json" ],
          "responseType" : [ "application/json", "application/
vnd.oracle.adf.resourceitem+json" ]
        }, {
          "name" : "testImpl",
          "parameters" : [ {
            "name" : "testid",
            "type" : "string",
            "mandatory" : false
          } ],
          "resultType" : "string",
          "method" : "POST",
          "requestType" : [ "application/vnd.oracle.adf.action+json" ],
          "responseType" : [ "application/json", "application/
vnd.oracle.adf.actionresult+json" ]
        } ]
      },
      "item" : {
        "links" : [ {
          "rel" : "child",
          "href" : "http://server/demo/rest/11.0/Departments/{id}/child/
Employees",
          "name" : "Employees",
          "kind" : "collection",
          "cardinality" : {
            "value" : "1 to *",
            "sourceAttributes" : "DepartmentId",
            "destinationAttributes" : "DepartmentId"
          }
        }, {
          "rel" : "self",
          "href" : "http://server/demo/rest/11.0/Departments/{id}",
          "name" : "self",
          "kind" : "item"
        }, {
          "rel" : "canonical",
          "href" : "http://server/demo/rest/11.0/Departments/{id}",
          "name" : "canonical",
          "kind" : "item"
        } ],
        "actions" : [ {
          "name" : "get",
          "method" : "GET",
          "responseType" : [ "application/json", "application/
vnd.oracle.adf.resourceitem+json" ]
        }, {
          "name" : "update",
          "method" : "PATCH",
          "requestType" : [ "application/vnd.oracle.adf.resourceitem+json" ],
          "responseType" : [ "application/json", "application/
vnd.oracle.adf.resourceitem+json" ]
        }, {
          "name" : "delete",
          "method" : "DELETE"
        } ]
      },
      "children" : {
        "Employees" : {
          "discrColumnType" : false,
          "attributes" : [ {
            "name" : "EmployeeId",
            "type" : "integer",
            "updatable" : true,
            "mandatory" : true,
            "queryable" : true,
            "precision" : 6
          }, {
            "name" : "FirstName",
            "type" : "string",
            "updatable" : true,
            "mandatory" : false,
            "queryable" : true,
            "precision" : 20
          }, {
            "name" : "LastName",
            "type" : "string",
            "updatable" : true,
            "mandatory" : true,
            "queryable" : true,
            "precision" : 25
          }, {
            "name" : "Email",
            "type" : "string",
            "updatable" : true,
            "mandatory" : true,
    "queryable" : true,
    "precision" : 25
  }, {
    "name" : "JobId",
    "type" : "string",
    "updatable" : true,
    "mandatory" : true,
    "queryable" : true,
    "precision" : 10,
    "controlType" : "choice",
    "maxLength" : "10",
    "lov" : {
      "childRef" : "JobsLOV",
      "attributeMap" : [ {
        "source" : "JobTitle",
        "target" : "JobId"
      } ],
      "displayAttributes" : [ "JobTitle" ]
    }
  }, {
    "name" : "DepartmentId",
    "type" : "integer",
    "updatable" : true,
    "mandatory" : false,
    "queryable" : true,
    "precision" : 4
  }, {
    "name" : "Salary",
    "type" : "number",
    "updatable" : true,
    "mandatory" : false,
    "queryable" : true,
    "precision" : 8,
    "scale" : 2
  }, {
    "name" : "Picture",
    "type" : "attachment",
    "updatable" : true,
    "mandatory" : false,
    "queryable" : false,
    "actions" : [ {
      "name" : "delete",
      "method" : "DELETE"
    }, {
      "name" : "get",
      "method" : "GET",
      "responseType" : [ "image/png" ]
    } ]
  } ],
  "collection" : {
    "rangeSize" : 0,
    "finders" : [ {
      "name" : "EmpByEmailFinder",
      "title" : "EmployeesByEmailVC",
      "attributes" : [ {
        "name" : "Email",
        "type" : "string",
        "updatable" : true,
        "required" : "Optional",
        "queryable" : false
      } ]
```

```
            }, {
              "name" : "PrimaryKey",
              "attributes" : [ {
                "name" : "EmployeeId",
                "type" : "integer",
                "updatable" : true,
                "mandatory" : true,
                "queryable" : true,
                "precision" : 6
              } ]
            } ],
            "links" : [ {
              "rel" : "self",
              "href" : "http://server/demo/rest/11.0/Departments/{id}/child/
Employees",
              "name" : "self",
              "kind" : "collection"
            } ],
            "actions" : [ {
              "name" : "get",
              "method" : "GET",
              "responseType" : [ "application/json", "application/
vnd.oracle.adf.resourcecollection+json" ]
            }, {
              "name" : "create",
              "method" : "POST",
              "requestType" : [ "application/
vnd.oracle.adf.resourceitem+json" ],
              "responseType" : [ "application/json", "application/
vnd.oracle.adf.resourceitem+json" ]
            } ]
          },
          "item" : {
            "links" : [ {
              "rel" : "self",
              "href" : "http://server/demo/rest/11.0/Departments/{id}/child/
Employees/{id}",
              "name" : "self",
              "kind" : "item"
            }, {
              "rel" : "parent",
              "href" : "http://server/demo/rest/11.0/Departments/{id}",
              "name" : "parent",
              "kind" : "item"
            }, {
              "rel" : "canonical",
              "href" : "http://server/demo/rest/11.0/Departments/{id}/child/
Employees/{id}",
              "name" : "canonical",
              "kind" : "item"
            }, {
              "rel" : "enclosure",
              "href" : "http://server/demo/rest/11.0/Departments/{id}/child/
Employees/{id}/enclosure/Picture",
              "name" : "Picture",
              "kind" : "other"
            } ],
            "actions" : [ {
              "name" : "get",
              "method" : "GET",
              "responseType" : [ "application/json", "application/
vnd.oracle.adf.resourceitem+json" ]
            }, {
              "name" : "update",
              "method" : "PATCH",
              "requestType" : [ "application/
vnd.oracle.adf.resourceitem+json" ],
              "responseType" : [ "application/json", "application/
vnd.oracle.adf.resourceitem+json" ]
            }, {
              "name" : "delete",
              "method" : "DELETE"
            }, {
              "name" : "multiplySalary",
              "parameters" : [ {
                "name" : "multiplicand",
                "type" : "number",
                "mandatory" : false
              } ],
              "resultType" : "number",
              "method" : "POST",
              "requestType" : [ "application/vnd.oracle.adf.action+json" ],
              "responseType" : [ "application/json", "application/
vnd.oracle.adf.actionresult+json" ]
            } ]
          },
          "links" : [ {
            "rel" : "self",
            "href" : "http://server/demo/rest/11.0/Departments/{id}/child/
Employees/describe",
            "name" : "self",
            "kind" : "describe"
          }, {
            "rel" : "canonical",
            "href" : "http://server/demo/rest/11.0/Departments/{id}/child/
Employees/describe",
            "name" : "canonical",
            "kind" : "describe"
          } ]
        }
      },
      "links" : [ {
        "rel" : "self",
        "href" : "http://server/demo/rest/11.0/Departments/describe",
        "name" : "self",
        "kind" : "describe"
      }, {
        "rel" : "canonical",
        "href" : "http://server/demo/rest/11.0/Departments/describe",
        "name" : "canonical",
        "kind" : "describe"
      } ]
    }
  }
}
```

Describing a Resource Item

The ADF REST runtime supports describing a resource item using a GET method.

Note that a resource item describe typically provides the same information as the collection describe. In the case of a polymorphic collection, however, a resource item describe will have more information than a resource collection describe. Consider, for example, the polymorphic collection vehicles, where the vehicles resource contains an item automobile and an item airplane. The description of vehicles will return a general shape of all vehicles, whereas, a resource item describe of an automobile will return the specific shape of that object.

To examine a resource item:

1. Execute the resource item describe and locate the names of the resources in the describe. The `children` attribute identifies nested resources.
2. Examine the resource item describe to understand its shape.
   - `attributes` specifies the list of available resource collection attributes.
   - `item` specifies `links` to the item, including defined child collections, and available `actions`.
   - `links` specifies describe links.

For example, the describe for a `Departments` resource item returns the following objects:

```
{
  "Resources" : {
    "Departments" : {
      "discrColumnType" : false,
      "attributes" : [ {
       ...
      } ],
      "item" : {
        "links" : [ {
        } ]
        "actions" : [ {
        } ]
      },
      "links" : [ {
      } ]
  }
}
```

The following sample describes version `11.0` of an instance of the `Departments` resource collection.

Note: To recursively include all children of the resource item on the requested describe, provide the query parameter `?includeChildren=true` on the describe URL.

Request

- **URL**

  `http://server/demo/rest/11.0/Departments/10/describe`
- **HTTP Method**

  GET
- **Content-Type**

  none
- **Payload**

none

Response

- **HTTP Code**

  `200`

- **Content-Type**

  `application/vnd.oracle.adf.description+json`

- **Payload**

```
{
  "Resources" : {
    "Departments" : {
      "discrColumnType" : false,
      "attributes" : [ {
        "name" : "DepartmentId",
        "type" : "integer",
        "updatable" : true,
        "mandatory" : true,
        "queryable" : true,
        "precision" : 4
      }, {
        "name" : "DepartmentName",
        "type" : "string",
        "updatable" : true,
        "mandatory" : true,
        "queryable" : true,
        "precision" : 30
      }, {
        "name" : "RelState",
        "type" : "integer",
        "updatable" : true,
        "mandatory" : false,
        "queryable" : true
      } ],
      "item" : {
        "links" : [ {
          "rel" : "child",
          "href" : "http://server/demo/rest/11.0/Departments/10/child/
Employees",
          "name" : "Employees",
          "kind" : "collection",
          "cardinality" : {
            "value" : "1 to *",
            "sourceAttributes" : "DepartmentId",
            "destinationAttributes" : "DepartmentId"
          }
        }, {
          "rel" : "self",
          "href" : "http://server/demo/rest/11.0/Departments/10",
          "name" : "self",
          "kind" : "item"
        }, {
          "rel" : "canonical",
          "href" : "http://server/demo/rest/11.0/Departments/10",
          "name" : "canonical",
          "kind" : "item"
        } ],
        "actions" : [ {
          "name" : "get",
          "method" : "GET",
          "responseType" : [ "application/json", "application/
vnd.oracle.adf.resourceitem+json" ]
        }, {
          "name" : "update",
          "method" : "PATCH",
          "requestType" : [ "application/vnd.oracle.adf.resourceitem+json" ],
          "responseType" : [ "application/json", "application/
vnd.oracle.adf.resourceitem+json" ]
        }, {
          "name" : "delete",
          "method" : "DELETE"
        } ]
      },
      "links" : [ {
        "rel" : "self",
        "href" : "http://server/demo/rest/11.0/Departments/10/describe",
        "name" : "self",
        "kind" : "describe"
      }, {
        "rel" : "canonical",
        "href" : "http://server/demo/rest/11.0/Departments/10/describe",
        "name" : "canonical",
        "kind" : "describe"
      } ]
    }
  }
}
```

Describing a Nested Resource

The ADF REST runtime supports describing an ADF REST nested resource using a GET method.

To examine nested resources in the resource catalog:

1. Execute the nested resource describe and locate the names of the resources in the describe. The `children` attribute identifies nested resources.
2. Examine these resource objects to understand the shape of each resource:
   - `attributes` specifies the list of available resource collection attributes.
   - `collection` specifies the shape of the collection and specifies `finder` for row finder, `links`, and available `actions` (including media types).
   - `item` specifies the shape of the instances of the collection and itself specifies `links` and available `actions`.
   - `children` specifies the nested resources (and itself contains `attributes`, `collection`, and `item` objects).

For example, the describe for the nested resources `Departments` and `Employees` returns the following objects:

```
{
  "Resources" : {
    "Employees" : {
      "discrColumnType" : false,
      "attributes" : [ {
       ...
```

```
   } ],
   "collection" : {
    ...
    } ],
    "finders" : [ {
    } ]
    "links" : [ {
    } ]
    "actions" : [ {
    } ]
   },
   "item" : {
    "links" : [ {
    } ]
    "actions" : [ {
    } ]
   },
   "children" : {
    "Departments"
     ...
    } ],
   ...
   "links" : [ {
   } ]
 }
}
```

The following sample (URL1) describes version `11.0` of the `Employees` resource which can be found in the context of a `Department`. The second sample (URL2), describes an item of the same nested resource `Employees`, where the ID of the employee identifies the nested resource item.

Note: To recursively include all children of the resource item on the requested describe, provide the query parameter `?includeChildren=true` on the describe URL.

Requests

- **URL 1**

  `http://server/demo/rest/11.0/Departments/10/child/Employees/describe`

- **URL 2**

  `http://server/demo/rest/11.0/Departments/10/child/Employees/200/describe`

- **HTTP Method**

  GET

- **Content-Type**

  none

- **Payload**

  none

Responses

- **HTTP Code**

  `200`

- **Content-Type**

```
application/vnd.oracle.adf.description+json
```

- **Payload 1**

```
{
  "Resources" : {
    "Employees" : {
      "discrColumnType" : false,
      "attributes" : [ {
        "name" : "EmployeeId",
        "type" : "integer",
        "updatable" : true,
        "mandatory" : true,
        "queryable" : true,
        "precision" : 6
      }, {
        "name" : "FirstName",
        "type" : "string",
        "updatable" : true,
        "mandatory" : false,
        "queryable" : true,
        "precision" : 20
      }, {
        "name" : "LastName",
        "type" : "string",
        "updatable" : true,
        "mandatory" : true,
        "queryable" : true,
        "precision" : 25
      }, {
        "name" : "Email",
        "type" : "string",
        "updatable" : true,
        "mandatory" : true,
        "queryable" : true,
        "precision" : 25
      }, {
        "name" : "JobId",
        "type" : "string",
        "updatable" : true,
        "mandatory" : true,
        "queryable" : true,
        "precision" : 10,
        "controlType" : "choice",
        "maxLength" : "10",
        "lov" : {
          "childRef" : "JobsLOV",
          "attributeMap" : [ {
            "source" : "JobTitle",
            "target" : "JobId"
          } ],
          "displayAttributes" : [ "JobTitle" ]
        }
      }, {
        "name" : "DepartmentId",
        "type" : "integer",
        "updatable" : true,
        "mandatory" : false,
        "queryable" : true,
        "precision" : 4
      }, {
        "name" : "Salary",
        "type" : "number",
        "updatable" : true,
        "mandatory" : false,
        "queryable" : true,
        "precision" : 8,
        "scale" : 2
      }, {
        "name" : "Picture",
        "type" : "attachment",
        "updatable" : true,
        "mandatory" : false,
        "queryable" : false,
        "actions" : [ {
          "name" : "delete",
          "method" : "DELETE"
        }, {
          "name" : "get",
          "method" : "GET",
          "responseType" : [ "image/png" ]
        } ]
      } ],
      "collection" : {
        "rangeSize" : 25,
        "finders" : [ {
          "name" : "EmpByEmailFinder",
          "title" : "EmployeesByEmailVC",
          "attributes" : [ {
            "name" : "Email",
            "type" : "string",
            "updatable" : true,
            "required" : "Optional",
            "queryable" : false
          } ]
        }, {
          "name" : "PrimaryKey",
          "attributes" : [ {
            "name" : "EmployeeId",
            "type" : "integer",
            "updatable" : true,
            "mandatory" : true,
            "queryable" : true,
            "precision" : 6
          } ]
        } ],
        "links" : [ {
          "rel" : "self",
          "href" : "http://server/demo/rest/11.0/Departments/10/child/
Employees",
          "name" : "self",
          "kind" : "collection"
        } ],
        "actions" : [ {
          "name" : "get",
          "method" : "GET",
          "responseType" : [ "application/json", "application/
vnd.oracle.adf.resourcecollection+json" ]
        }, {
          "name" : "create",
          "method" : "POST",
          "requestType" : [ "application/vnd.oracle.adf.resourceitem+json" ],
          "responseType" : [ "application/json", "application/
vnd.oracle.adf.resourceitem+json" ]
        } ]
      },
      "item" : {
        "links" : [ {
          "rel" : "self",
          "href" : "http://server/demo/rest/11.0/Departments/10/child/
Employees/{id}",
          "name" : "self",
          "kind" : "item"
        }, {
          "rel" : "parent",
          "href" : "http://server/demo/rest/11.0/Departments/10",
          "name" : "parent",
          "kind" : "item"
        }, {
          "rel" : "canonical",
          "href" : "http://server/demo/rest/11.0/Departments/10/child/
Employees/{id}",
          "name" : "canonical",
          "kind" : "item"
        }, {
          "rel" : "enclosure",
          "href" : "http://server/demo/rest/11.0/Departments/10/child/
Employees/enclosure/Picture",
          "name" : "Picture",
          "kind" : "other"
        } ],
        "actions" : [ {
          "name" : "get",
          "method" : "GET",
          "responseType" : [ "application/json", "application/
vnd.oracle.adf.resourceitem+json" ]
        }, {
          "name" : "update",
          "method" : "PATCH",
          "requestType" : [ "application/vnd.oracle.adf.resourceitem+json" ],
          "responseType" : [ "application/json", "application/
vnd.oracle.adf.resourceitem+json" ]
        }, {
          "name" : "delete",
          "method" : "DELETE"
        }, {
          "name" : "multiplySalary",
          "parameters" : [ {
            "name" : "multiplicand",
            "type" : "number",
            "mandatory" : false
          } ],
          "resultType" : "number",
          "method" : "POST",
          "requestType" : [ "application/vnd.oracle.adf.action+json" ],
          "responseType" : [ "application/json", "application/
vnd.oracle.adf.actionresult+json" ]
        } ]
      },
      "links" : [ {
        "rel" : "self",
        "href" : "http://server/demo/rest/11.0/Departments/10/child/
Employees/describe",
        "name" : "self",
        "kind" : "describe"
      }, {
        "rel" : "canonical",
        "href" : "http://server/demo/rest/11.0/Departments/10/child/
Employees/describe",
        "name" : "canonical",
        "kind" : "describe"
      } ]
    }
  }
}
```

- **Payload 2**

```
{
  "Resources" : {
    "Employees" : {
      "discrColumnType" : false,
      "attributes" : [ {
        "name" : "EmployeeId",
        "type" : "integer",
        "updatable" : true,
        "mandatory" : true,
        "queryable" : true,
        "precision" : 6
      }, {
        "name" : "FirstName",
        "type" : "string",
        "updatable" : true,
        "mandatory" : false,
        "queryable" : true,
        "precision" : 20
      }, {
        "name" : "LastName",
        "type" : "string",
        "updatable" : true,
        "mandatory" : true,
        "queryable" : true,
        "precision" : 25
      }, {
        "name" : "Email",
        "type" : "string",
        "updatable" : true,
        "mandatory" : true,
        "queryable" : true,
        "precision" : 25
      }, {
        "name" : "JobId",
        "type" : "string",
        "updatable" : true,
        "mandatory" : true,
        "queryable" : true,
        "precision" : 10,
        "controlType" : "choice",
        "maxLength" : "10",
        "lov" : {
          "childRef" : "JobsLOV",
          "attributeMap" : [ {
            "source" : "JobTitle",
            "target" : "JobId"
          } ],
          "displayAttributes" : [ "JobTitle" ]
        }
      }, {
        "name" : "DepartmentId",
        "type" : "integer",
        "updatable" : true,
        "mandatory" : false,
        "queryable" : true,
        "precision" : 4
      }, {
        "name" : "Salary",
        "type" : "number",
        "updatable" : true,
        "mandatory" : false,
        "queryable" : true,
        "precision" : 8,
        "scale" : 2
      }, {
        "name" : "Picture",
        "type" : "attachment",
        "updatable" : true,
        "mandatory" : false,
        "queryable" : false,
        "actions" : [ {
          "name" : "delete",
          "method" : "DELETE"
        }, {
          "name" : "get",
          "method" : "GET",
          "responseType" : [ "image/png" ]
        } ]
      } ],
      "item" : {
        "links" : [ {
          "rel" : "self",
          "href" : "http://server/demo/rest/11.0/Departments/10/child/
Employees/200",
          "name" : "self",
          "kind" : "item"
        }, {
          "rel" : "parent",
          "href" : "http://server/demo/rest/11.0/Departments/10",
          "name" : "parent",
          "kind" : "item"
        }, {
          "rel" : "canonical",
          "href" : "http://server/demo/rest/11.0/Departments/10/child/
Employees/200",
          "name" : "canonical",
          "kind" : "item"
        }, {
          "rel" : "enclosure",
          "href" : "http://server/demo/rest/11.0/Departments/10/child/
Employees/200/enclosure/Picture",
          "name" : "Picture",
          "kind" : "other"
        } ],
        "actions" : [ {
          "name" : "get",
          "method" : "GET",
          "responseType" : [ "application/json", "application/
vnd.oracle.adf.resourceitem+json" ]
```

```
        }, {
          "name" : "update",
          "method" : "PATCH",
          "requestType" : [ "application/vnd.oracle.adf.resourceitem+json" ],
          "responseType" : [ "application/json", "application/
vnd.oracle.adf.resourceitem+json" ]
        }, {
          "name" : "delete",
          "method" : "DELETE"
        }, {
          "name" : "multiplySalary",
          "parameters" : [ {
            "name" : "multiplicand",
            "type" : "number",
            "mandatory" : false
          } ],
          "resultType" : "number",
          "method" : "POST",
          "requestType" : [ "application/vnd.oracle.adf.action+json" ],
          "responseType" : [ "application/json", "application/
vnd.oracle.adf.actionresult+json" ]
        } ]
      },
      "links" : [ {
        "rel" : "self",
        "href" : "http://server/demo/rest/11.0/Departments/10/child/
Employees/200/describe",
        "name" : "self",
        "kind" : "describe"
      }, {
        "rel" : "canonical",
        "href" : "http://server/demo/rest/11.0/Departments/10/child/
Employees/200/describe",
        "name" : "canonical",
        "kind" : "describe"
      } ]
    }
  }
}
```

Describing Multiple Resource Collections

REST APIs support describing two or more named resource collections.

To examine the resource collections:

1. Execute the resource collection describe with the named resources using the `resources` query parameter.
2. Locate the names of the resources in the describe.
3. Examine these resource objects to understand the shape of each resource:
   - `attributes` specifies the list of available resource collection attributes.
   - `collection` specifies the shape of the collection and specifies `finders` for row finder, `links`, and available `actions` (including media types).
   - `item` specifies the shape of the instances of the collection and itself specifies `links` and available `actions`.

- `children` specifies any nested resources (and itself contains `attributes`, `collection`, and `item` objects).

For example, the describe for the `Departments` resource returns the following objects:

```
{
  "Resources" : {
    "Departments" : {
      "discrColumnType" : false,
      "attributes" : [ {
       ...
      } ],
      "collection" : {
        ...
        } ],
        "finders" : [ {
        } ]
        "links" : [ {
        } ]
        "actions" : [ {
        } ]
      },
      "item" : {
        "links" : [ {
        } ]
        "actions" : [ {
        } ]
      },
      "children" : {
        "Employees"
          ...
        } ],
      ...
      "links" : [ {
      } ]
  }
}
```

The following sample uses the `resources` query parameter to describe version `11.0` of the `Departments` and `Employees` resources named on the request.

Request

- **URL**

  `http://server/demo/rest/11.0/Departments/describe?resources=Departments,Employees`

- **HTTP Method**

  GET

- **Content-Type**

  none

- **Payload**

  none

Response

- **HTTP Code**

```
200
```

- **Content-Type**

```
application/vnd.oracle.adf.description+json
```

- **Payload**

```
{
  "Resources" : {
    "Departments" : {
      "discrColumnType" : false,
      "attributes" : [ {
        "name" : "DepartmentId",
        "type" : "integer",
        "updatable" : true,
        "mandatory" : true,
        "queryable" : true,
        "precision" : 4
      }, {
        "name" : "DepartmentName",
        "type" : "string",
        "updatable" : true,
        "mandatory" : true,
        "queryable" : true,
        "precision" : 30
      }, {
        "name" : "RelState",
        "type" : "integer",
        "updatable" : true,
        "mandatory" : false,
        "queryable" : true
      } ],
      "collection" : {
        "rangeSize" : 25,
        "finders" : [ {
          "name" : "PrimaryKey",
          "attributes" : [ {
            "name" : "DepartmentId",
            "type" : "integer",
            "updatable" : true,
            "mandatory" : true,
            "queryable" : true,
            "precision" : 4
          } ]
        } ],
        "links" : [ {
          "rel" : "self",
          "href" : "http://server/demo/rest/11.0/Departments",
          "name" : "self",
          "kind" : "collection"
        } ],
        "actions" : [ {
          "name" : "get",
          "method" : "GET",
          "responseType" : [ "application/json", "application/
vnd.oracle.adf.resourcecollection+json" ]
        }, {
          "name" : "create",
          "method" : "POST",
          "requestType" : [ "application/vnd.oracle.adf.resourceitem+json" ],
          "responseType" : [ "application/json", "application/
```

```
vnd.oracle.adf.resourceitem+json" ]
        }, {
          "name" : "testImpl",
          "parameters" : [ {
            "name" : "testid",
            "type" : "string",
            "mandatory" : false
          } ],
          "resultType" : "string",
          "method" : "POST",
          "requestType" : [ "application/vnd.oracle.adf.action+json" ],
          "responseType" : [ "application/json", "application/
vnd.oracle.adf.actionresult+json" ]
        } ]
      },
      "item" : {
        "links" : [ {
          "rel" : "child",
          "href" : "http://server/demo/rest/11.0/Departments/{id}/child/
Employees",
          "name" : "Employees",
          "kind" : "collection",
          "cardinality" : {
            "value" : "1 to *",
            "sourceAttributes" : "DepartmentId",
            "destinationAttributes" : "DepartmentId"
          }
        }, {
          "rel" : "self",
          "href" : "http://server/demo/rest/11.0/Departments/{id}",
          "name" : "self",
          "kind" : "item"
        }, {
          "rel" : "canonical",
          "href" : "http://server/demo/rest/11.0/Departments/{id}",
          "name" : "canonical",
          "kind" : "item"
        } ],
        "actions" : [ {
          "name" : "get",
          "method" : "GET",
          "responseType" : [ "application/json", "application/
vnd.oracle.adf.resourceitem+json" ]
        }, {
          "name" : "update",
          "method" : "PATCH",
          "requestType" : [ "application/vnd.oracle.adf.resourceitem+json" ],
          "responseType" : [ "application/json", "application/
vnd.oracle.adf.resourceitem+json" ]
        }, {
          "name" : "delete",
          "method" : "DELETE"
        } ]
      },
      "children" : {
        "Employees" : {
          "discrColumnType" : false,
          "attributes" : [ {
            "name" : "EmployeeId",
            "type" : "integer",
            "updatable" : true,
      "mandatory" : true,
      "queryable" : true,
      "precision" : 6
    }, {
      "name" : "FirstName",
      "type" : "string",
      "updatable" : true,
      "mandatory" : false,
      "queryable" : true,
      "precision" : 20
    }, {
      "name" : "LastName",
      "type" : "string",
      "updatable" : true,
      "mandatory" : true,
      "queryable" : true,
      "precision" : 25
    }, {
      "name" : "Email",
      "type" : "string",
      "updatable" : true,
      "mandatory" : true,
      "queryable" : true,
      "precision" : 25
    }, {
      "name" : "JobId",
      "type" : "string",
      "updatable" : true,
      "mandatory" : true,
      "queryable" : true,
      "precision" : 10,
      "controlType" : "choice",
      "maxLength" : "10",
      "lov" : {
        "childRef" : "JobsLOV",
        "attributeMap" : [ {
          "source" : "JobTitle",
          "target" : "JobId"
        } ],
        "displayAttributes" : [ "JobTitle" ]
      }
    }, {
      "name" : "DepartmentId",
      "type" : "integer",
      "updatable" : true,
      "mandatory" : false,
      "queryable" : true,
      "precision" : 4
    }, {
      "name" : "Salary",
      "type" : "number",
      "updatable" : true,
      "mandatory" : false,
      "queryable" : true,
      "precision" : 8,
      "scale" : 2
    }, {
      "name" : "Picture",
      "type" : "attachment",
      "updatable" : true,
      "mandatory" : false,
            "queryable" : false,
            "actions" : [ {
              "name" : "delete",
              "method" : "DELETE"
            }, {
              "name" : "get",
              "method" : "GET",
              "responseType" : [ "image/png" ]
            } ]
          } ],
          "collection" : {
            "rangeSize" : 0,
            "finders" : [ {
              "name" : "EmpByEmailFinder",
              "title" : "EmployeesByEmailVC",
              "attributes" : [ {
                "name" : "Email",
                "type" : "string",
                "updatable" : true,
                "required" : "Optional",
                "queryable" : false
              } ]
            }, {
              "name" : "PrimaryKey",
              "attributes" : [ {
                "name" : "EmployeeId",
                "type" : "integer",
                "updatable" : true,
                "mandatory" : true,
                "queryable" : true,
                "precision" : 6
              } ]
            } ],
            "links" : [ {
              "rel" : "self",
              "href" : "http://server/demo/rest/11.0/Departments/{id}/child/
Employees",
              "name" : "self",
              "kind" : "collection"
            } ],
            "actions" : [ {
              "name" : "get",
              "method" : "GET",
              "responseType" : [ "application/json", "application/
vnd.oracle.adf.resourcecollection+json" ]
            }, {
              "name" : "create",
              "method" : "POST",
              "requestType" : [ "application/
vnd.oracle.adf.resourceitem+json" ],
              "responseType" : [ "application/json", "application/
vnd.oracle.adf.resourceitem+json" ]
            } ]
          },
          "item" : {
            "links" : [ {
              "rel" : "self",
              "href" : "http://server/demo/rest/11.0/Departments/{id}/child/
Employees/{id}",
              "name" : "self",
              "kind" : "item"
            }, {
              "rel" : "parent",
              "href" : "http://server/demo/rest/11.0/Departments/{id}",
              "name" : "parent",
              "kind" : "item"
            }, {
              "rel" : "canonical",
              "href" : "http://server/demo/rest/11.0/Departments/{id}/child/
Employees/{id}",
              "name" : "canonical",
              "kind" : "item"
            }, {
              "rel" : "enclosure",
              "href" : "http://server/demo/rest/11.0/Departments/{id}/child/
Employees/{id}/enclosure/Picture",
              "name" : "Picture",
              "kind" : "other"
            } ],
            "actions" : [ {
              "name" : "get",
              "method" : "GET",
              "responseType" : [ "application/json", "application/
vnd.oracle.adf.resourceitem+json" ]
            }, {
              "name" : "update",
              "method" : "PATCH",
              "requestType" : [ "application/
vnd.oracle.adf.resourceitem+json" ],
              "responseType" : [ "application/json", "application/
vnd.oracle.adf.resourceitem+json" ]
            }, {
              "name" : "delete",
              "method" : "DELETE"
            }, {
              "name" : "multiplySalary",
              "parameters" : [ {
                "name" : "multiplicand",
                "type" : "number",
                "mandatory" : false
              } ],
              "resultType" : "number",
              "method" : "POST",
              "requestType" : [ "application/vnd.oracle.adf.action+json" ],
              "responseType" : [ "application/json", "application/
vnd.oracle.adf.actionresult+json" ]
            } ]
          },
          "links" : [ {
            "rel" : "self",
            "href" : "http://server/demo/rest/11.0/Departments/{id}/child/
Employees/describe",
            "name" : "self",
            "kind" : "describe"
          }, {
            "rel" : "canonical",
            "href" : "http://server/demo/rest/11.0/Departments/{id}/child/
Employees/describe",
            "name" : "canonical",
            "kind" : "describe"
          } ]
        }
```

```
      },
      "links" : [ {
        "rel" : "self",
        "href" : "http://server/demo/rest/11.0/Departments/describe",
        "name" : "self",
        "kind" : "describe"
      }, {
        "rel" : "canonical",
        "href" : "http://server/demo/rest/11.0/Departments/describe",
        "name" : "canonical",
        "kind" : "describe"
      } ]
    }
    "Employees" : {
      "discrColumnType" : false,
      "attributes" : [ {
        "name" : "EmployeeId",
        "type" : "integer",
        "updatable" : true,
        "mandatory" : true,
        "queryable" : true,
        "precision" : 6
      }, {
        "name" : "FirstName",
        "type" : "string",
        "updatable" : true,
        "mandatory" : false,
        "queryable" : true,
        "precision" : 20
      }, {
        "name" : "LastName",
        "type" : "string",
        "updatable" : true,
        "mandatory" : true,
        "queryable" : true,
        "precision" : 25
      }, {
        "name" : "Email",
        "type" : "string",
        "updatable" : true,
        "mandatory" : true,
        "queryable" : true,
        "precision" : 25
      }, {
        "name" : "JobId",
        "type" : "string",
        "updatable" : true,
        "mandatory" : true,
        "queryable" : true,
        "precision" : 10,
        "controlType" : "choice",
        "maxLength" : "10",
        "lov" : {
              "childRef" : "JobsLOV",
              "attributeMap" : [ {
                "source" : "JobTitle",
                "target" : "JobId"
              } ],
              "displayAttributes" : [ "JobTitle" ]
        }
      }, {
        "name" : "DepartmentId",
        "type" : "integer",
        "updatable" : true,
        "mandatory" : false,
        "queryable" : true,
        "precision" : 4
      }, {
        "name" : "Salary",
        "type" : "number",
        "updatable" : true,
        "mandatory" : false,
        "queryable" : true,
        "precision" : 8,
        "scale" : 2
      }, {
        "name" : "Picture",
        "type" : "attachment",
        "updatable" : true,
        "mandatory" : false,
        "queryable" : false,
      "actions" : [ {
        "name" : "delete",
        "method" : "DELETE"
      }, {
        "name" : "get",
        "method" : "GET",
        "responseType" : [ "image/png" ]
      } ]
    } ],
      "collection" : {
        "rangeSize" : 0,
        "finders" : [ {
        "name" : "EmpByEmailFinder",
        "title" : "EmployeesByEmailVC",
        "attributes" : [ {
           "name" : "Email",
           "type" : "string",
           "updatable" : true,
           "required" : "Optional",
           "queryable" : false
         } ]
      }, {
        "name" : "PrimaryKey",
        "attributes" : [ {
                "name" : "EmployeeId",
                "type" : "integer",
                "updatable" : true,
                "mandatory" : true,
                "queryable" : true,
                "precision" : 6
        } ]
      } ],
            "links" : [ {
              "rel" : "self",
              "href" : "http://server/demo/rest/11.0/Departments/{id}/child/
Employees",
              "name" : "self",
              "kind" : "collection"
            } ],
            "actions" : [ {
              "name" : "get",
              "method" : "GET",
              "responseType" : [ "application/json", "application/
vnd.oracle.adf.resourcecollection+json" ]
            }, {
              "name" : "create",
              "method" : "POST",
              "requestType" : [ "application/
vnd.oracle.adf.resourceitem+json" ],
              "responseType" : [ "application/json", "application/
vnd.oracle.adf.resourceitem+json" ]
            } ]
          },
          "item" : {
            "links" : [ {
              "rel" : "self",
              "href" : "http://server/demo/rest/11.0/Departments/{id}/child/
Employees/{id}",
              "name" : "self",
              "kind" : "item"
            }, {
              "rel" : "parent",
              "href" : "http://server/demo/rest/11.0/Departments/{id}",
              "name" : "parent",
              "kind" : "item"
            }, {
              "rel" : "canonical",
              "href" : "http://server/demo/rest/11.0/Departments/{id}/child/
Employees/{id}",
              "name" : "canonical",
              "kind" : "item"
            }, {
              "rel" : "enclosure",
              "href" : "http://server/demo/rest/11.0/Departments/{id}/child/
Employees/{id}/enclosure/Picture",
              "name" : "Picture",
              "kind" : "other"
            } ],
            "actions" : [ {
              "name" : "get",
              "method" : "GET",
              "responseType" : [ "application/json", "application/
vnd.oracle.adf.resourceitem+json" ]
            }, {
              "name" : "update",
              "method" : "PATCH",
              "requestType" : [ "application/
vnd.oracle.adf.resourceitem+json" ],
              "responseType" : [ "application/json", "application/
vnd.oracle.adf.resourceitem+json" ]
            }, {
              "name" : "delete",
              "method" : "DELETE"
            }, {
              "name" : "multiplySalary",
              "parameters" : [ {
                "name" : "multiplicand",
                "type" : "number",
                "mandatory" : false
              } ],
              "resultType" : "number",
              "method" : "POST",
              "requestType" : [ "application/vnd.oracle.adf.action+json" ],
              "responseType" : [ "application/json", "application/
vnd.oracle.adf.actionresult+json" ]
            } ]
          },
          "links" : [ {
            "rel" : "self",
            "href" : "http://server/demo/rest/11.0/Departments/{id}/child/
Employees/describe",
            "name" : "self",
            "kind" : "describe"
          }, {
            "rel" : "canonical",
            "href" : "http://server/demo/rest/11.0/Departments/{id}/child/
Employees/describe",
            "name" : "canonical",
            "kind" : "describe"
          } ]
        }
      },
      "links" : [ {
        "rel" : "self",
        "href" : "http://server/demo/rest/11.0/Departments/describe",
        "name" : "self",
        "kind" : "describe"
      }, {
        "rel" : "canonical",
        "href" : "http://server/demo/rest/11.0/Departments/describe",
        "name" : "canonical",
        "kind" : "describe"
      } ]
    }
   }
}
```

Partially Describing a Nested Resource Collection

The ADF REST runtime supports obtaining a partial describe for a nested resource using a GET method and the parameter `partialDescription` to limit the depth of the hierarchy to the next child resource only.

The `partialDescription` parameter used with a describe request ensures the response payload contains only the nested collection to a depth of one nested object in the nested parent-child hierarchy. The describe for a child of the nested parent resource is not expand and provides only a link to retrieve that partial describe for the next resource in the hierarchy.

To examine the partial describe of a nested resource collection and its child resource, if any:

1. Execute the nested resource collection describe with the `partialDescription` parameter and locate the name of the nested resource in the describe.
2. Examine these resource objects to understand the shape of the resource. Specifically examine the `children` element to identify the partial describe link for the next child resource in the resource hierarchy (to a depth of one nested resource).
   - `attributes` specifies the list of available resource collection attributes.

- `collection` specifies the shape of the collection and specifies `finder` for row finder, `links`, and available `actions` (including media types).
- `item` specifies the shape of the instances of the collection and itself specifies `links` and available `actions`.
- `children` identifies any nested resource to a depth of one object and contains a describe link with the `?partialDescription` parameter to retrieve its partial description (this child resource describe specifically does not contain the `attributes`, `collection`, and `item` objects).

For example, the partial describe for the nested `Employees` resource returns the following objects. Note that in this sample, the `Employees` collection contains the child collection `Bonus`. The partial describe request limits the depth to the nested resource and a single child of the nested resource, in this case, `Bonus`.

```
{
  "Resources" : {
    "Employees" : {
      "discrColumnType" : false,
      "attributes" : [ {
       ...
      } ],
      "collection" : {
        ...
        } ],
        "finders" : [ {
        } ]
        "links" : [ {
        } ]
        "actions" : [ {
        } ]
      },
      "item" : {
        "links" : [ {
        } ]
        "actions" : [ {
        } ]
      },
      "children" : {
        "Bonus" : {
          "$ref" : "http://server/demo/rest/12.0/Departments/describe?
partialDescription=Employees.Bonus"
        } ],
      ...
      "links" : [ {
      } ]
  }
}
```

The following sample retrieves a partial describe of version `12.0` of the nested `Employees` resource and contains a link to retrieve the partial describe for the next child in the resource hierarchy, `Bonus`.

Request

- **URL**

  `http://server/demo/rest/12.0/Departments/describe?partialDescription=Employees`

- **HTTP Method**

  GET

- **Content-Type**

  none

- **Payload**

  none

Response

- **HTTP Code**

  `200`

- **Content-Type**

  `application/vnd.oracle.adf.description+json`

- **Payload**

```
{
  "Resources" : {
    "Employees" : {
      "discrColumnType" : false,
      "ServiceConfiguration" : {
        "Cache-Control" : "max-age=30"
      },
      "attributes" : [ {
        "name" : "EmployeeId",
        "type" : "integer",
        "updatable" : true,
        "mandatory" : true,
        "queryable" : true,
        "allowChanges" : "always",
        "precision" : 6
      }, {
        "name" : "FirstName",
        "type" : "string",
        "updatable" : true,
        "mandatory" : false,
        "queryable" : true,
        "allowChanges" : "always",
        "precision" : 20
      }, {
        "name" : "LastName",
        "type" : "string",
        "updatable" : true,
        "mandatory" : true,
        "queryable" : true,
        "allowChanges" : "always",
        "precision" : 25
      }, {
        "name" : "Email",
        "type" : "string",
        "updatable" : true,
        "mandatory" : true,
        "queryable" : true,
        "allowChanges" : "always",
        "precision" : 25
      }, {
        "name" : "PhoneNumber",
      "type" : "string",
      "updatable" : true,
      "mandatory" : false,
      "queryable" : true,
      "allowChanges" : "always",
      "precision" : 20
    }, {
      "name" : "HireDate",
      "type" : "datetime",
      "updatable" : true,
      "mandatory" : true,
      "queryable" : true,
      "allowChanges" : "always"
    }, {
      "name" : "JobId",
      "type" : "string",
      "updatable" : true,
      "mandatory" : true,
      "queryable" : true,
      "allowChanges" : "always",
      "precision" : 10
    }, {
      "name" : "Salary",
      "type" : "number",
      "updatable" : true,
      "mandatory" : false,
      "queryable" : true,
      "allowChanges" : "always",
      "precision" : 8,
      "scale" : 2
    }, {
      "name" : "CommissionPct",
      "type" : "number",
      "updatable" : true,
      "mandatory" : false,
      "queryable" : true,
      "allowChanges" : "always",
      "precision" : 2,
      "scale" : 2
    }, {
      "name" : "ManagerId",
      "type" : "integer",
      "updatable" : true,
      "mandatory" : false,
      "queryable" : true,
      "allowChanges" : "always",
      "precision" : 6
    }, {
      "name" : "DepartmentId",
      "type" : "integer",
      "updatable" : true,
      "mandatory" : false,
      "queryable" : true,
      "allowChanges" : "always",
      "precision" : 4
    } ],
    "collection" : {
      "rangeSize" : 0,
      "finders" : [ {
        "name" : "PrimaryKey",
        "attributes" : [ {
            "name" : "EmployeeId",
            "type" : "integer",
            "updatable" : true,
            "mandatory" : true,
            "queryable" : true,
            "allowChanges" : "always",
            "precision" : 6
          } ]
        } ],
        "links" : [ {
          "rel" : "self",
          "href" : "http://server/demo/rest/12.0/Departments/{id}/child/
Employees",
          "name" : "self",
          "kind" : "collection"
        } ],
        "actions" : [ {
          "name" : "get",
          "method" : "GET",
          "responseType" : [ "application/
vnd.oracle.adf.resourcecollection+json", "application/json" ]
        }, {
          "name" : "create",
          "method" : "POST",
          "requestType" : [ "application/vnd.oracle.adf.resourceitem+json",
"application/json" ],
          "responseType" : [ "application/vnd.oracle.adf.resourceitem+json",
"application/json" ]
        }, {
          "name" : "upsert",
          "method" : "POST",
          "header" : "Upsert-Mode=true",
          "requestType" : [ "application/vnd.oracle.adf.resourceitem+json",
"application/json" ],
          "responseType" : [ "application/vnd.oracle.adf.resourceitem+json",
"application/json" ]
        } ]
      },
      "item" : {
        "links" : [ {
          "rel" : "self",
          "href" : "http://server/demo/rest/12.0/Departments/{id}/child/
Employees/{id}",
          "name" : "self",
          "kind" : "item"
        }, {
          "rel" : "parent",
          "href" : "http://server/demo/rest/12.0/Departments/{id}",
          "name" : "parent",
          "kind" : "item"
        }, {
          "rel" : "canonical",
          "href" : "http://server/demo/rest/12.0/Departments/{id}/child/
Employees/{id}",
          "name" : "canonical",
          "kind" : "item"
        } ],
        "actions" : [ {
          "name" : "get",
          "method" : "GET",
          "responseType" : [ "application/vnd.oracle.adf.resourceitem+json",
"application/json" ]
        }, {
          "name" : "update",
          "method" : "PATCH",
          "requestType" : [ "application/vnd.oracle.adf.resourceitem+json",
"application/json" ],
          "responseType" : [ "application/vnd.oracle.adf.resourceitem+json",
"application/json" ]
        }, {
          "name" : "delete",
          "method" : "DELETE"
        } ]
      },
      "children" : {
        "Bonus" : {
          "$ref" : "http://server/demo/rest/12.0/Departments/describe?
partialDescription=Employees.Bonus"
        }
      }
      "links" : [ {
        "rel" : "self",
        "href" : "http://server/demo/rest/12.0/Departments/{id}/child/
Employees/describe",
        "name" : "self",
        "kind" : "describe"
      }, {
        "rel" : "canonical",
        "href" : "http://server/demo/rest/12.0/Departments/{id}/child/
Employees/describe",
        "name" : "canonical",
        "kind" : "describe"
      } ]
    }
  }
}
```

Describing a Resource Collection with Specified Fields

The ADF REST runtime supports obtaining a limited describe for resource collections using a GET method and the parameter `fields`. The `fields` parameter used with a describe request ensures the response payload contains only the specified attributes of the collection.

To examine a resource collection:

1. Execute the resource collection describe and locate the names of the resources in the describe.
2. Examine these resource objects to understand the shape of each resource:
   - `attributes` specifies the list of available resource collection attributes.
   - `collection` specifies the shape of the collection and specifies `finder` for row finder, `links`, and available `actions` (including media types).
   - `item` specifies the shape of the instances of the collection and itself specifies `links` and available `actions`.
   - `children` specifies any nested resources (and itself contains `attributes`, `collection`, and `item` objects).

For example, the describe for the `Departments` resource returns the following objects:

```
{
  "Resources" : {
    "Departments" : {
      "discrColumnType" : false,
      "attributes" : [ {
       ...
      } ],
      "collection" : {
        ...
        } ],
        "finders" : [ {
        } ]
        "links" : [ {
        } ]
        "actions" : [ {
        } ]
      },
      "item" : {
        "links" : [ {
        } ]
        "actions" : [ {
        } ]
      },
      "children" : {
        "Employees"
          ...
        } ],
      ...
      "links" : [ {
      } ]
  }
}
```

The following samples show retrieving a limited describe for version `12.0` of the `Departments` and nested `Employees` resources. The first request (URL 1) uses the `fields` parameter to retrieve a single attribute of the `Departments` resource collection. The second request (URL 2) uses the `fields` parameter and retrieves all attributes of the `Departments` resource collection by specifying the wildcard character (`*`) and then also retrieves the single `EmployeeId` attribute of the nested `Employees` child resource collection.

Request

- **URL 1**

  ```
  http://server/demo/rest/12.0/Departments/describe?
  fields=DepartmentName
  ```

  URL 2

  ```
  http://server/demo/rest/12.0/Departments/describe?
  fields=*;Employees:EmployeeId
  ```

- **HTTP Method**

  GET

- **Content-Type**

  none

- **Payload**

  none

Response

- **HTTP Code**

  `200`

- **Content-Type**

  `application/vnd.oracle.adf.description+json`

- **Payload 1**

```
{
  "Resources" : {
    "Departments" : {
      "discrColumnType" : false,
      "attributes" : [ {
        "name" : "DepartmentName",
        "type" : "string",
        "updatable" : true,
        "mandatory" : true,
        "queryable" : true,
        "precision" : 30
      } ],
      "collection" : {
        "rangeSize" : 25,
        "finders" : [ {
          "name" : "PrimaryKey",
          "attributes" : [ {
            "name" : "DepartmentId",
            "type" : "integer",
            "updatable" : true,
            "mandatory" : true,
            "queryable" : true,
            "precision" : 4
          } ]
        } ],
        "links" : [ {
          "rel" : "self",
          "href" : "http://server/demo/rest/12.0/Departments",
          "name" : "self",
          "kind" : "collection"
        } ],
        "actions" : [ {
          "name" : "get",
          "method" : "GET",
          "responseType" : [ "application/json", "application/
vnd.oracle.adf.resourcecollection+json" ]
        }, {
          "name" : "create",
          "method" : "POST",
          "requestType" : [ "application/vnd.oracle.adf.resourceitem+json" ],
          "responseType" : [ "application/json", "application/
vnd.oracle.adf.resourceitem+json" ]
        }, {
          "name" : "upsert",
          "method" : "POST",
          "header" : "Upsert-Mode=true",
          "requestType" : [ "application/json", "application/
vnd.oracle.adf.resourceitem+json" ],
          "responseType" : [ "application/json", "application/
vnd.oracle.adf.resourceitem+json" ]
        }, {        } ]
      },
      "item" : {
        "links" : [ {
          "rel" : "child",
          "href" : "http://server/demo/rest/12.0/Departments/{id}/child/
Employees",
          "name" : "Employees",
          "kind" : "collection",
          "cardinality" : {
            "value" : "1 to *",
            "sourceAttributes" : "DepartmentId",
            "destinationAttributes" : "DepartmentId"
          }
        }, {
          "rel" : "self",
          "href" : "http://server/demo/rest/12.0/Departments/{id}",
          "name" : "self",
          "kind" : "item"
        }, {
          "rel" : "canonical",
          "href" : "http://server/demo/rest/12.0/Departments/{id}",
          "name" : "canonical",
          "kind" : "item"
        } ],
        "actions" : [ {
          "name" : "get",
          "method" : "GET",
          "responseType" : [ "application/json", "application/
vnd.oracle.adf.resourceitem+json" ]
        }, {
          "name" : "update",
          "method" : "PATCH",
          "requestType" : [ "application/vnd.oracle.adf.resourceitem+json" ],
          "responseType" : [ "application/json", "application/
vnd.oracle.adf.resourceitem+json" ]
        }, {
          "name" : "delete",
          "method" : "DELETE"
        } ]
      },
      "children" : {
        "Employees" : {
          "$ref" : "http://server/demo/rest/12.0/Departments/describe?
partialDescription=Employees"
        }
      },
      "links" : [ {
        "rel" : "self",
        "href" : "http://server/demo/rest/12.0/Departments/describe",
        "name" : "self",
        "kind" : "describe"
      }, {
        "rel" : "canonical",
        "href" : "http://server/demo/rest/12.0/Departments/describe",
        "name" : "canonical",
        "kind" : "describe"
      } ]
```

```
    }
  }
}
```

Payload 2

```
{
    "Resources" : {
    "Departments" : {
      "discrColumnType" : false,
      "ServiceConfiguration" : {
        "Cache-Control" : "max-age=30"
      },
      "attributes" : [ {
        "name" : "DepartmentId",
        "type" : "integer",
        "updatable" : true,
        "mandatory" : true,
        "queryable" : true,
        "allowChanges" : "always",
        "precision" : 4
      }, {
        "name" : "DepartmentName",
        "type" : "string",
        "updatable" : true,
        "mandatory" : true,
        "queryable" : true,
        "allowChanges" : "always",
        "precision" : 30
      }, {
        "name" : "ManagerId",
        "type" : "integer",
        "updatable" : true,
        "mandatory" : false,
        "queryable" : true,
        "allowChanges" : "always",
        "precision" : 6
      }, {
        "name" : "LocationId",
        "type" : "integer",
        "updatable" : true,
        "mandatory" : false,
        "queryable" : true,
        "allowChanges" : "always",
        "precision" : 4
      } ],
      "collection" : {
        "rangeSize" : 25,
        "finders" : [ {
          "name" : "PrimaryKey",
          "attributes" : [ {
            "name" : "DepartmentId",
            "type" : "integer",
            "updatable" : true,
            "mandatory" : true,
            "queryable" : true,
            "allowChanges" : "always",
            "precision" : 4
          } ]
        } ],
        "links" : [ {
          "rel" : "self",
          "href" : "http://server/demo/rest/12.0/Departments",
          "name" : "self",
          "kind" : "collection"
        } ],
        "actions" : [ {
          "name" : "get",
          "method" : "GET",
          "responseType" : [ "application/
vnd.oracle.adf.resourcecollection+json", "application/json" ]
        }, {
          "name" : "create",
          "method" : "POST",
          "requestType" : [ "application/vnd.oracle.adf.resourceitem+json",
"application/json" ],
          "responseType" : [ "application/vnd.oracle.adf.resourceitem+json",
"application/json" ]
        }, {
          "name" : "upsert",
          "method" : "POST",
          "header" : "Upsert-Mode=true",
          "requestType" : [ "application/vnd.oracle.adf.resourceitem+json",
"application/json" ],
          "responseType" : [ "application/vnd.oracle.adf.resourceitem+json",
"application/json" ]
        } ]
      },
      "item" : {
        "links" : [ {
          "rel" : "child",
          "href" : "http://server/demo/rest/12.0/Departments/{id}/child/
Employees",
          "name" : "Employees",
          "kind" : "collection",
          "cardinality" : {
            "value" : "1 to *",
            "sourceAttributes" : "DepartmentId",
            "destinationAttributes" : "DepartmentId"
          }
        }, {
          "rel" : "self",
          "href" : "http://server/demo/rest/12.0/Departments/{id}",
          "name" : "self",
          "kind" : "item"
        }, {
          "rel" : "canonical",
          "href" : "http://server/demo/rest/12.0/Departments/{id}",
          "name" : "canonical",
          "kind" : "item"
        } ],
        "actions" : [ {
          "name" : "get",
          "method" : "GET",
          "responseType" : [ "application/vnd.oracle.adf.resourceitem+json",
"application/json" ]
        }, {
          "name" : "update",
          "method" : "PATCH",
          "requestType" : [ "application/vnd.oracle.adf.resourceitem+json",
"application/json" ],
          "responseType" : [ "application/vnd.oracle.adf.resourceitem+json",
"application/json" ]
        }, {
          "name" : "delete",
          "method" : "DELETE"
        } ]
      },
      "children" : {
        "Employees" : {
          "discrColumnType" : false,
          "ServiceConfiguration" : {
            "Cache-Control" : "max-age=30"
          },
          "attributes" : [ {
            "name" : "EmployeeId",
            "type" : "integer",
            "updatable" : true,
            "mandatory" : true,
            "queryable" : true,
            "allowChanges" : "always",
            "precision" : 6
          } ],
          "collection" : {
            "rangeSize" : 0,
            "finders" : [ {
              "name" : "PrimaryKey",
              "attributes" : [ {
                "name" : "EmployeeId",
                "type" : "integer",
                "updatable" : true,
                "mandatory" : true,
                "queryable" : true,
                "allowChanges" : "always",
                "precision" : 6
              } ]
            } ],
            "links" : [ {
              "rel" : "self",
              "href" : "http://server/demo/rest/12.0/Departments/{id}/child/
Employees",
              "name" : "self",
              "kind" : "collection"
            } ],
            "actions" : [ {
              "name" : "get",
              "method" : "GET",
              "responseType" : [ "application/
vnd.oracle.adf.resourcecollection+json", "application/json" ]
            }, {
              "name" : "create",
              "method" : "POST",
              "requestType" : [ "application/
vnd.oracle.adf.resourceitem+json", "application/json" ],
              "responseType" : [ "application/
vnd.oracle.adf.resourceitem+json", "application/json" ]
            }, {
              "name" : "upsert",
              "method" : "POST",
              "header" : "Upsert-Mode=true",
              "requestType" : [ "application/
vnd.oracle.adf.resourceitem+json", "application/json" ],
              "responseType" : [ "application/
vnd.oracle.adf.resourceitem+json", "application/json" ]
            } ]
          },
          "item" : {
            "links" : [ {
              "rel" : "self",
              "href" : "http://server/demo/rest/12.0/Departments/{id}/child/
Employees/{id}",
              "name" : "self",
              "kind" : "item"
            }, {
              "rel" : "parent",
              "href" : "http://server/demo/rest/12.0/Departments/{id}",
              "name" : "parent",
              "kind" : "item"
            }, {
              "rel" : "canonical",
              "href" : "http://server/demo/rest/12.0/Departments/{id}/child/
Employees/{id}",
              "name" : "canonical",
              "kind" : "item"
            } ],
            "actions" : [ {
              "name" : "get",
              "method" : "GET",
              "responseType" : [ "application/
vnd.oracle.adf.resourceitem+json", "application/json" ]
            }, {
              "name" : "update",
              "method" : "PATCH",
              "requestType" : [ "application/
vnd.oracle.adf.resourceitem+json", "application/json" ],
              "responseType" : [ "application/
vnd.oracle.adf.resourceitem+json", "application/json" ]
            }, {
              "name" : "delete",
              "method" : "DELETE"
            } ]
          },
          "links" : [ {
            "rel" : "self",
            "href" : "http://server/demo/rest/12.0/Departments/{id}/child/
Employees/describe",
            "name" : "self",
            "kind" : "describe"
          }, {
            "rel" : "canonical",
            "href" : "http://server/demo/rest/12.0/Departments/{id}/child/
Employees/describe",
            "name" : "canonical",
            "kind" : "describe"
          } ]
        }
      },
      "links" : [ {
        "rel" : "self",
        "href" : "http://server/demo/rest/12.0/Departments/describe",
        "name" : "self",
        "kind" : "describe"
      }, {
        "rel" : "canonical",
                "href" : "http://server/demo/rest/12.0/Departments/describe",
                "name" : "canonical",
                "kind" : "describe"
              } ]
            }
          }
        }
```

Using ETags in Describe Requests

ETags provide the ability for service clients to determine the metadata changes in the web server since the last time it fetched the describe. This is accomplished by returning an ETag header in the response of a describe request.

An ETag is an identifier assigned by a web server to a specific version of a REST resource describe. If the REST resource describe representation ever changes, a new and different ETag is assigned. ETags are used in the HTTP header and are unique. For more information on using ETags, see Checking for Data Consistency.

When the REST resource describe is retrieved, the web server will return the REST resource's current representation along with its corresponding ETag value, which is placed in an HTTP response header `ETag` field. For example:

```
ETag: "5265766973696F6E33"
```

The client may cache the representation, along with its ETag. If the client wants to retrieve the same describe again, it will send its previously saved copy of the ETag along with the request in a `If-None-Match` field. For example:

```
If-None-Match: "5265766973696F6E33"
```

On this subsequent request, the server compares the client's ETag with the ETag for the current version of the resource. If the ETag values match, which indicates that the resource has not changed, then the server sends a response with a `HTTP 304 Not Modified` status. This status indicates that the cached version is good for use.

The following samples describe the ETag behavior for the `employees` resource.

Example 22-1 When ETag Values Match

Request

- **URL**

  `/rest/v1/employees/describe`

- **HTTP Method**

  GET

- **If-None-Match**

  `"5265766973696F6E33"`

- **Metadata-Context**

  `label=latest`

- **Payload**

none

Response

- **HTTP Code**

  `304`

- **ETag**

  `"5265766973696F6E33"`

- **Metadata-Context**

  `label="Revision3"`

- **Payload**

  none

Example 22-2   When ETag Values Do Not Match

Request

- **URL**

  `/rest/v1/employees/describe`

- **HTTP Method**

  GET

- **If-None-Match**

  `"5265766973696F6E33"`

- **Metadata-Context**

  `label=latest`

- **Payload**

  none

Response

- **HTTP Code**

  `200`

- **ETag**

  `"5265766973696F6E34"`

- **Metadata-Context**

  `label="Revision4"`

- **Payload**

```
{
  "Resources" : {
    "employees" : {
      ...
    }
  }
}
```

> **Note:**
>
> ETags are not returned if the request metadata context is not configured to read the latest MDS label.

Retrieving Resource Versions

In ADF, you can create version identifiers for an ADF REST resource and assign lifecycle status so that you can manage and update an existing resource definition with ease.
In JDeveloper, ADF REST resource developers create version identifiers and associate these identifiers with the resources they create. The use of the version identifier allows the REST resource developer to manage updating an existing resource definition. When a resource needs to be updated with a change that is not backward compatible with the previous version, the resource developer may create a new version of the resource and assign it a unique version identifier.

The ADF REST runtime supports getting the versions defined in the application, which can include retrieving all available versions or only a specific version.

Additionally, the REST resource developer can assign a lifecycle status (`active`, `desupported`, `deprecated`) when they create or update a version identifier. Resources that have been tagged `deprecated` are handled as valid resources and that status has no runtime impact. Whereas, resources that have been tagged with the lifecycle status `desupported` will no longer be accessible by the service client and requests for desupported versions will return the following error message:

```
JBO-29151: The requested version 'x' has been desupported.
```

If the service client requests a resource that does not exist for the requested version, the ADF REST runtime will fallback to a previous version with status `active` or `deprecated`. Thus, if the `Departments` resource versions `11.0` and `11.1` exist but `11.2` does not, the request for version `11.2` will return the next more recent resource, version `11.1`.

Retrieving All Available Release Version Names

The ADF REST runtime supports retrieving the release version names defined for the service end point using a GET method.

To examine the release version names:

1. Execute the service end point describe and locate the available release version names in the describe.
2. Examine these elements to understand the order of the release versions:
   - `version` specifies the version name, as defined in the `adf-config.xml` file.
   - `isLatest` property specifies the version name of the most recent release version.
   - `predecessor-version` specifies the link to the previous release version.
   - `successor-version` specifies the link to the next latest release version.

- `current` specifies the link to the most recent release version.

3. Optionally, examine these elements to understand the version of the ADF REST framework version that will be used to process requests for a specific release version:
    - `defaultFrameworkVersion` property specifies the default ADF REST framework version that has been associated with a particular release version, as optionally defined by the ADF REST resource developer in the `adf-config.xml` file. Note that a new ADF REST framework version may introduce new functionality. Thus, associating a specific framework version with each release version ensures that service clients interact with the appropriate level of functionality. See Working with ADF REST Framework Versions.
    - `allowedFrameworkVersions` property specifies the list of ADF REST framework versions that are supported for a particular release version. Service clients may override the default framework version with any value in the list by specifying the value in the `REST-Framework-Version` header.

For example, the describe for a service end point with two release versions returns the following objects:

```
{
    "items" : [
        {
            "version" : "version_identifier_latest",
            "isLatest" : true,
            "adf:extension" : {
                    "defaultFrameworkVersion" : "framework_identifier",
                    "allowedFrameworkVersions" : [ "framework_identifier1",
"framework_identifier2", ... ]

            },
            "links" : [
                {
                    "rel" : "self",
                    ...

                },
                {
                    "rel" : "canonical",
                    ...

                },
                {
                    "rel" : "predecessor-version",
                    ...

                },
                {
                    "rel" : "describe",
                    ...

                }
            ]
        },
        {
            "version" : "version_identifier_previous",
            "adf:extension" : {
                    "defaultFrameworkVersion" : "framework_identifier",
                    "allowedFrameworkVersions" : [ "framework_identifier1",
"framework_identifier2", ... ]
            },
            "links" : [
                {
                    "rel" : "self",
                    ...
                },
                {
                    "rel" : "canonical",
                    ...
                },
                {
                    "rel" : "successor-version",
                    ...
                },
                {
                    "rel" : "describe",
                    ...
                }
            ]
        }
    ],
    "links" : [
     ...
        {
            "rel" : "current",
                    ...
        }
    ]
}
```

The following sample retrieves all available release versions defined in the resource catalog of the `demo` application. In the sample, the three release versions are `11.0`, `11.1`, and `11.2`, where `11.2` is the current (or most recent) release version. Note that links of type `predecessor-version` and `successor-version` specify the location of each release version relative to their position in the list of versions. In the sample, versions `11.0` and `11.1` are explicitly associated with ADF REST framework version `1` and release version `11.2` is associated with ADF REST framework version `3`. Note that a framework version refers to a specific version of the ADF REST framework, corresponding to a particular Oracle JDeveloper release, where a new framework version introduces new functionality.

Request

- **URL**

  `http://server/demo/rest`

- **HTTP Method**

  GET

- **Content-Type**

  none

- **Payload**

  none

Response

- **HTTP Code**

`200`

- **Content-Type**

  `application/vnd.oracle.adf.description+json`

- **Payload**

```
{
    "items" : [
        {
            "version" : "11.2",
            "isLatest" : true,
            "adf:extension" : {
                    "defaultFrameworkVersion" : "3",
                    "allowedFrameworkVersions" : [ "1", "2", "3" ]
            },
            "links" : [
                {
                    "rel" : "self",
                    "href" : "http://server/demo/rest/11.2",
                    "name" : "self",
                    "kind" : "item"
                },
                {
                    "rel" : "canonical",
                    "href" : "http://server/demo/rest/11.2",
                    "name" : "canonical",
                    "kind" : "item"
                },
                {
                    "rel" : "predecessor-version",
                    "href" : "http://server/demo/rest/11.1",
                    "name" : "predecessor-version",
                    "kind" : "item"
                },
                {
                    "rel" : "describe",
                    "href" : "http://server/demo/rest/11.2/describe",
                    "name" : "describe",
                    "kind" : "describe"
                }
            ]
        },
        {
            "version" : "11.1",
            "adf:extension" : {
                    "defaultFrameworkVersion" : "1",
                    "allowedFrameworkVersions" : [ "1", "2", "3" ]
            },
            "links" : [
                {
                    "rel" : "self",
                    "href" : "http://server/demo/rest/11.1",
                    "name" : "self",
                    "kind" : "item"
                },
                {
                    "rel" : "canonical",
                    "href" : "http://server/demo/rest/11.1",
                    "name" : "canonical",
                    "kind" : "item"
                },
                {
                    "rel" : "predecessor-version",
                    "href" : "http://server/demo/rest/11.0",
                    "name" : "predecessor-version",
                    "kind" : "item"
                },
                {
                    "rel" : "successor-version",
                    "href" : "http://server/demo/rest/11.2",
                    "name" : "successor-version",
                    "kind" : "item"
                },
                {
                    "rel" : "describe",
                    "href" : "http://server/demo/rest/11.1/describe",
                    "name" : "describe",
                    "kind" : "describe"
                }
            ]
        },
        {
            "version" : "11.0",
            "adf:extension" : {
                    "defaultFrameworkVersion" : "1",
                    "allowedFrameworkVersions" : [ "1", "2", "3" ]
            },
            "links" : [
                {
                    "rel" : "self",
                    "href" : "http://server/demo/rest/11.0",
                    "name" : "self",
                    "kind" : "item"
                },
                {
                    "rel" : "canonical",
                    "href" : "http://server/demo/rest/11.0",
                    "name" : "canonical",
                    "kind" : "item"
                },
                {
                    "rel" : "successor-version",
                    "href" : "http://server/demo/rest/11.1",
                    "name" : "successor-version",
                    "kind" : "item"
                },
                {
                    "rel" : "describe",
                    "href" : "http://server/demo/rest/11.0/describe",
                    "name" : "describe",
                    "kind" : "describe"
                }
            ]
        }
    ],
    "links" : [
        {
            "rel" : "self",
            "href" : "http://server/demo/rest",
            "name" : "self",
            "kind" : "collection"
        },
        {
            "rel" : "canonical",
            "href" : "http://server/demo/rest",
            "name" : "canonical",
            "kind" : "collection"
        },
        {
            "rel" : "current",
            "href" : "http://server/demo/rest/11.2",
            "name" : "current",
            "kind" : "item"
        }
    ]
}
```

Retrieving a Resource By a Specific Version

The ADF REST runtime supports retrieving a specific version of the resource using a GET method. The specific resource version is one that has been associated at design time with the resource. Release version identifiers are defined in the Release Version page of the `adf-config.xml` overview editor.

The following samples describe two versions of the `Departments` resource. The first resource describe (request 1) returns version `11.0` and the second resource describe (request 2) returns version `11.1`. The resource describe for `11.1` shows additional mandatory attributes `HireDate` and `PhoneNumber` have been defined on the resource.

Request Version 1

- **URL 1**

  `http://server/demo/rest/11.0/Departments`

- **HTTP Method**

  GET

- **Content-Type**

  none

- **Payload**

  none

Response

- **HTTP Code**

  `200`

- **Content-Type**

  `application/vnd.oracle.adf.describe+json`

- **Payload 1**

  ```
  {
    "Resources" : {
      "Employees" : {
        "discrColumnType" : false,
        "attributes" : [ {
          "name" : "EmployeeId",
  ```

```
      "type" : "integer",
      "updatable" : true,
      "mandatory" : true,
      "queryable" : true,
      "precision" : 6
    }, {
      "name" : "FirstName",
      "type" : "string",
      "updatable" : true,
      "mandatory" : false,
      "queryable" : true,
      "precision" : 20
    }, {
      "name" : "LastName",
      "type" : "string",
      "updatable" : true,
      "mandatory" : true,
      "queryable" : true,
      "precision" : 25
    }, {
      "name" : "Email",
      "type" : "string",
      "updatable" : true,
      "mandatory" : true,
      "queryable" : true,
      "precision" : 25
    }, {
      "name" : "JobId",
      "type" : "string",
      "updatable" : true,
      "mandatory" : true,
      "queryable" : true,
      "precision" : 10,
      "controlType" : "choice",
      "maxLength" : "10",
    }, {
      "name" : "DepartmentId",
      "type" : "integer",
      "updatable" : true,
      "mandatory" : false,
      "queryable" : true,
      "precision" : 4
    }, {
      "name" : "Salary",
      "type" : "number",
      "updatable" : true,
      "mandatory" : false,
      "queryable" : true,
      "precision" : 8,
      "scale" : 2
    }, {
      "name" : "Picture",
      "type" : "attachment",
      "updatable" : true,
      "mandatory" : false,
      "queryable" : false,
      "actions" : [ {
        "name" : "delete",
        "method" : "DELETE"
      }, {
        "name" : "get",
     "method" : "GET",
     "responseType" : [ "image/png" ]
    } ]
   } ],
   "collection" : {
      ...
     } ]
    } ],
    "links" : [ {
     "rel" : "self",
     "href" : "http://server/demo/rest/11.0/Employees",
     "name" : "self",
     "kind" : "collection"
    } ],
    "actions" : [ {
     ...
    } ]
   },
   "item" : {
    "links" : [ {
     ...
    } ],
   },
   "links" : [ {
    "rel" : "self",
    "href" : "http://server/demo/rest/11.0/Employees/describe",
    "name" : "self",
    "kind" : "describe"
   }, {
    "rel" : "canonical",
    "href" : "http://server/demo/rest/11.0/Employees/describe",
    "name" : "canonical",
    "kind" : "describe"
   } ]
  }
 }
}
```

Request Version 2

- **URL 2**

  `http://server/demo/rest/11.1/Departments`

- **HTTP Method**

  GET

- **Content-Type**

  none

- **Payload**

  none

Response

- **HTTP Code**

  `200`

- **Content-Type**

  `application/vnd.oracle.adf.describe+json`

- **Payload 2**

```
{
  "Resources" : {
    "Employees" : {
      "discrColumnType" : false,
      "title" : "Employees All Attributes",
        "name" : "EmployeeId",
        "type" : "integer",
        "updatable" : true,
        "mandatory" : true,
        "queryable" : true,
        "precision" : 6
      }, {
        "name" : "FirstName",
        "type" : "string",
        "updatable" : true,
        "mandatory" : false,
        "queryable" : true,
        "precision" : 20
      }, {
        "name" : "LastName",
        "type" : "string",
        "updatable" : true,
        "mandatory" : true,
        "queryable" : true,
        "precision" : 25
      }, {
        "name" : "Email",
        "type" : "string",
        "updatable" : true,
        "mandatory" : true,
        "queryable" : true,
        "precision" : 25
      }, {
        "name" : "JobId",
        "type" : "string",
        "updatable" : true,
        "mandatory" : true,
        "queryable" : true,
        "precision" : 10
      }, {
        "name" : "DepartmentId",
        "type" : "integer",
        "updatable" : true,
        "mandatory" : false,
        "queryable" : true,
        "precision" : 4
      }, {
        "name" : "Salary",
        "type" : "number",
        "updatable" : true,
        "mandatory" : false,
        "queryable" : true,
        "precision" : 8,
        "scale" : 2
      }, {
        "name" : "Picture",
        "type" : "attachment",
        "updatable" : true,
        "mandatory" : false,
        "queryable" : false,
        "actions" : [ {
          "name" : "delete",
          "method" : "DELETE"
        }, {
          "name" : "get",
          "method" : "GET",
          "responseType" : [ "image/png" ]
        } ]
      }, {
        "name" : "HireDate",
        "type" : "string",
        "updatable" : true,
        "mandatory" : false,
        "queryable" : true
      }, {
        "name" : "PhoneNumber",
        "type" : "string",
        "updatable" : true,
        "mandatory" : false,
        "queryable" : true,
        "precision" : 20
      } ],
      "collection" : {
           ...
          } ]
        } ],
        "links" : [ {
          "rel" : "self",
          "href" : "http://server/demo/rest/11.1/Employees",
          "name" : "self",
          "kind" : "collection"
        } ],
        "actions" : [ {
          ...
        } ]
      },
      "item" : {
        "links" : [ {
          ...
        } ]
      },
      "links" : [ {
        "rel" : "self",
        "href" : "http://server/demo/rest/11.1/Employees/describe",
        "name" : "self",
        "kind" : "describe"
      }, {
        "rel" : "canonical",
        "href" : "http://server/demo/rest/11.1/Employees/describe",
        "name" : "canonical",
        "kind" : "describe"
      } ]
    }
  }
}
```

Retrieving the ADF REST Resource

The ADF REST runtime supports the GET method on REST resources to retrieve a resource or nested resource, page a resource, filter a resource collection with a query, or sort a resource collection.
The ADF REST runtime supports the following GET method use cases:

- Fetching a resource collection.
- Fetching a paged resource collection.
- Filtering a resource payload using primary key finder parameters.
- Filtering a resource payload using query parameters.
- Fetching a resource item.
- Fetching nested child resources.
- Fetching a sorted resource collection.
- Fetching a resource collection or resource item and grouping item details under the `@context` element.

Fetching a Resource Collection

The ADF REST runtime supports fetching a resource collection using a GET method.

The following sample fetches version `11.0` of the `Departments` resource collection and five items.

Request

- **URL**

  `http://server/sample/rest/11.0/Departments`

- **HTTP Method**

  GET

- **Content-Type**

  none

- **Payload**

  none

Response

- **HTTP Code**

  `200`

- **Content-Type**

  `application/vnd.oracle.adf.resourcecollection+json`

- **Payload**

```
{
  "items" : [ {
    "DepartmentId" : 10,
    "DepartmentName" : "Administration",
    "links" : [ {
      "rel" : "self",
      "href" : "http://server/demo/rest/11.0/Departments/10",
      "name" : "Departments",
      "kind" : "item"
    }, {
      "rel" : "canonical",
      "href" : "http://server/demo/rest/11.0/Departments/10",
      "name" : "Departments",
      "kind" : "item"
    }, {
      "rel" : "child",
      "href" : "http://server/demo/rest/11.0/Departments/10/child/Employees",
      "name" : "Employees",
      "kind" : "collection"
    } ]
  }, {
    "DepartmentId" : 20,
    "DepartmentName" : "Marketing",
    "links" : [ {
      "rel" : "self",
      "href" : "http://server/demo/rest/11.0/Departments/20",
      "name" : "Departments",
      "kind" : "item"
    }, {
      "rel" : "canonical",
      "href" : "http://server/demo/rest/11.0/Departments/20",
      "name" : "Departments",
      "kind" : "item"
    }, {
      "rel" : "child",
      "href" : "http://server/demo/rest/11.0/Departments/20/child/Employees",
      "name" : "Employees",
      "kind" : "collection"
    } ]
  }, {
    "DepartmentId" : 30,
    "DepartmentName" : "Purchasing",
    "links" : [ {
      "rel" : "self",
      "href" : "http://server/demo/rest/11.0/Departments/30",
      "name" : "Departments",
      "kind" : "item"
    }, {
      "rel" : "canonical",
      "href" : "http://server/demo/rest/11.0/Departments/30",
      "name" : "Departments",
      "kind" : "item"
    }, {
      "rel" : "child",
      "href" : "http://server/demo/rest/11.0/Departments/30/child/Employees",
      "name" : "Employees",
      "kind" : "collection"
    } ]
  }, {
    "DepartmentId" : 40,
    "DepartmentName" : "Human Resources",
    "links" : [ {
      "rel" : "self",
      "href" : "http://server/demo/rest/11.0/Departments/40",
      "name" : "Departments",
      "kind" : "item"
    }, {
      "rel" : "canonical",
      "href" : "http://server/demo/rest/11.0/Departments/40",
      "name" : "Departments",
      "kind" : "item"
    }, {
      "rel" : "child",
      "href" : "http://server/demo/rest/11.0/Departments/40/child/Employees",
      "name" : "Employees",
      "kind" : "collection"
    } ]
  }, {
    "DepartmentId" : 50,
    "DepartmentName" : "Shipping",
    "links" : [ {
      "rel" : "self",
      "href" : "http://server/demo/rest/11.0/Departments/50",
      "name" : "Departments",
      "kind" : "item"
    }, {
      "rel" : "canonical",
      "href" : "http://server/demo/rest/11.0/Departments/50",
      "name" : "Departments",
      "kind" : "item"
    }, {
      "rel" : "child",
      "href" : "http://server/demo/rest/11.0/Departments/50/child/Employees",
      "name" : "Employees",
      "kind" : "collection"
    } ]
  } ],
  "count" : 5,
  "hasMore" : true,
  "limit" : 25,
  "offset" : 0,
  "links" : [ {
    "rel" : "self",
    "href" : "http://server/demo/rest/11.0/Departments",
    "name" : "Departments",
    "kind" : "collection"
  } ]
}
```

Paging a Resource Collection

The ADF REST runtime supports retrieving resource collections with row set pagination using a GET method. Paging a resource collection with the GET method is performed using the following URI query parameters:

- **limit** restricts the number of resources returned inside the resource collection. If the limit exceeds the resource count, then the framework will return all available resources. The value is the maximum number of resources to be returned.
- **offset** defines a zero-based index into the collection (where 0 is the first position). The index identifies the starting position of the resource collection. If offset exceeds the resource count, then no resources are returned.

In the following sample, where a `Departments` resource collection has five items, the first request (URL1) retrieves two items (at index 0 and 1), and because `offset` is omitted, the starting position of the response is the first item. To display another set of resource items, a second request (URL2) may be made with an `offset` of `2` to correspond to the third item and a `limit` of `2` to retrieve only two more items (at index 2 and 3), and the last request (URL3) with an `offset` of `4` returns the last item of the five item resource collection (at index 4).

Each time a new set of resource items is retrieved, the `hasMore` attribute of the response indicates whether more items may be returned from the collection. In this example, because the collection contains only five items, the response for URL3 shows `hasMore` set to `false`, indicating that the last set of items had been retrieved.

Note that when the `limit` parameter is omitted from the paging URL, the ADF REST runtime assumes a `limit` of `25` (as determined by the default `RangeSize` value on the resource's iterator binding definition). In this case, up to twenty-five items will be returned with each request. For this reason, it is a best practice when paging through a collection to always include the `limit` query parameter to ensure only the desired number of resource items are returned and not more.

Requests

- **URL 1**

  `http://server/demo/rest/11.0/Departments?limit=2`

- **URL 2**

  `http://server/demo/rest/11.0/Departments?offset=2&limit=2`

- **URL 3**

  `http://server/demo/rest/11.0/Departments?offset=4&limit=2`

- **HTTP Method**

  GET

- **Content-Type**

  none

- **Payload**

  none

Responses

- **HTTP Code**

  `200`

- **Content-Type**

  `application/vnd.oracle.adf.resourcecollection+json`

- **Payload 1**

```
{
  "items" : [ {
    "DepartmentId" : 10,
    "DepartmentName" : "Administration",
    "links" : [ {
      "rel" : "self",
      "href" : "http://server/demo/rest/11.0/Departments/10",
      "name" : "Departments",
      "kind" : "item"
    }, {
      "rel" : "canonical",
      "href" : "http://server/demo/rest/11.0/Departments/10",
      "name" : "Departments",
      "kind" : "item"
    }, {
      "rel" : "child",
      "href" : "http://server/demo/rest/11.0/Departments/10/child/Employees",
      "name" : "Employees",
      "kind" : "collection"
    } ]
  }, {
    "DepartmentId" : 20,
    "DepartmentName" : "Marketing",
    "links" : [ {
      "rel" : "self",
      "href" : "http://server/demo/rest/11.0/Departments/20",
      "name" : "Departments",
      "kind" : "item"
    }, {
      "rel" : "canonical",
      "href" : "http://server/demo/rest/11.0/Departments/20",
      "name" : "Departments",
      "kind" : "item"
    }, {
      "rel" : "child",
      "href" : "http://server/demo/rest/11.0/Departments/20/child/Employees",
      "name" : "Employees",
      "kind" : "collection"
    } ]
  } ],
  "count" : 2,
  "hasMore" : true,
  "limit" : 2,
  "offset" : 0,
  "links" : [ {
    "rel" : "self",
    "href" : "http://server/demo/rest/11.0/Departments",
    "name" : "Departments",
    "kind" : "collection"
  } ]
}
```

- **Payload 2**

```
{
  "items" : [ {
    "DepartmentId" : 30,
    "DepartmentName" : "Purchasing",
    "links" : [ {
      "rel" : "self",
      "href" : "http://server/demo/rest/11.0/Departments/30",
      "name" : "Departments",
      "kind" : "item"
    }, {
      "rel" : "canonical",
      "href" : "http://server/demo/rest/11.0/Departments/30",
      "name" : "Departments",
      "kind" : "item"
    }, {
      "rel" : "child",
      "href" : "http://server/demo/rest/11.0/Departments/30/child/Employees",
      "name" : "Employees",
      "kind" : "collection"
    } ]
  }, {
    "DepartmentId" : 40,
    "DepartmentName" : "Human Resources",
    "links" : [ {
      "rel" : "self",
      "href" : "http://server/demo/rest/11.0/Departments/40",
      "name" : "Departments",
      "kind" : "item"
    }, {
      "rel" : "canonical",
      "href" : "http://server/demo/rest/11.0/Departments/40",
      "name" : "Departments",
      "kind" : "item"
    }, {
      "rel" : "child",
      "href" : "http://server/demo/rest/11.0/Departments/40/child/Employees",
      "name" : "Employees",
      "kind" : "collection"
    } ]
  } ],
  "count" : 2,
  "hasMore" : true,
  "limit" : 2,
  "offset" : 2,
  "links" : [ {
    "rel" : "self",
    "href" : "http://server/demo/rest/11.0/Departments",
    "name" : "Departments",
    "kind" : "collection"
  } ]
}
```

- **Payload 3**

```
{
  "items" : [ {
    "DepartmentId" : 50,
    "DepartmentName" : "Shipping",
    "links" : [ {
      "rel" : "self",
      "href" : "http://server/demo/rest/11.0/Departments/50",
      "name" : "Departments",
      "kind" : "item"
    }, {
      "rel" : "canonical",
      "href" : "http://server/demo/rest/11.0/Departments/50",
      "name" : "Departments",
      "kind" : "item"
    }, {
      "rel" : "child",
      "href" : "http://server/demo/rest/11.0/Departments/50/child/Employees",
      "name" : "Employees",
      "kind" : "collection"
    } ]
  } ],
  "count" : 1
  "hasMore" : false,
  "limit" : 2
  "offset" : 4,
  "links" : [ {
        "rel" : "self",
        "href" : "http://server/demo/rest/11.0/Departments",
        "name" : "Departments",
        "kind" : "collection"
      } ]
    }
```

Filtering a Resource Collection with Primary Key Values

The ADF REST runtime supports fetching a resource collection using a GET method and a fixed URL that includes a finder defined by one or more primary key attributes of the resource. Every view object upon which the resource is based defines at least one primary key attribute. The ADF REST runtime supports passing primary key values in the finder query string to filter the collection.

Filtering with a primary key (PK) is performed using the **finder** query string to specify one or more primary key values. The **finder** with primary key query string parameter format is:

```
finder=PrimaryKey;<PKattr1>=<PKvalue1>,<PKattr2>=<PKvalue2>,...
```

For example, a resource collection Employees defines the primary key attribute `EmployeeId`. To filter the collection using the primary key finder for a specific employee, the finder query string may be specified as the following example shows.

```
finder=PrimaryKey;EmployeeId=101
```

When the resource is defined by a multi-part primary key, the finder allows you to pass the corresponding number of primary key values to filter the collection. For example, a resource collection Inventory might have `ProductId` and `WarehouseId` as its primary key attributes. To filter the collection using the primary key finder for a specific inventory item, both primary key values must be supplied in the finder query string as the example below shows.

```
finder=PrimaryKey;ProductId=8568,WarehouseId=45
```

The resource collection describe explains the finder and primary key attributes. To work with the primary key finder:

1. Execute the resource describe and locate the `finders` attribute in the collection element. The `name` attribute identifies the finder as `PrimaryKey`. Also locate the name of the primary key attributes under `attributes`.
2. Execute a GET with the query parameter `finder` and pass the primary key attributes using the finder name `PrimaryKey`.

For example, the Employees resources describe returns the following:

```
"collection" : {
     "rangeSize" : 24,
     "finders" : [ {
       "name" : "PrimaryKey",
       "attributes" : [ {
         "name" : "EmployeeId",
         "type" : "integer",
         "updatable" : true,
         "mandatory" : true,
         "queryable" : false
         "precision" : 4,
} ]
```

The following sample fetches the `Employees` collection specified by the finder `PrimaryKey` where the `EmployeeId` attribute value `101` is passed.

Note: To filter the collection by a non-primary key attribute, see Filtering a Resource Collection with a Row Finder.

Request

- **URL**

  ```
  http://server/demo/rest/11.1/Employees?
  finder=PrimaryKey;EmployeeId=101
  ```

- **HTTP Method**

  GET

- **Content-Type**

  none

- **Payload**

  none

Response

- **HTTP Code**

  `200`

- **Content-Type**

  `application/vnd.oracle.adf.resourceitem+json`

- **Payload**

  ```
  {
    "items" : [ {
      "EmployeeId" : 101,
      "FirstName" : "Neena",
      "LastName" : "Smith",
      "Email" : "NSMITH",
      "JobId" : "AD_VP",
      "DepartmentId" : 90,
      "Salary" : 2000,
      "links" : [ {
        "rel" : "self",
        "href" : "http://server/demo/rest/11.1/Employees/101",
        "name" : "Employees",
        "kind" : "item"
      }, {
        "rel" : "canonical",
        "href" : "http://server/demo/rest/11.1/Employees/101",
        "name" : "Employees",
        "kind" : "item"
      } ]
    } ],
    "count" : 1,
    "hasMore" : false,
    "limit" : 25,
    "offset" : 0,
    "links" : [ {
      "rel" : "self",
      "href" : "http://server/demo/rest/11.1/Employees",
  ```

```
        "name" : "Employees",
        "kind" : "collection"
      } ]
    }
```

Filtering a Resource Collection with a Query Parameter

The ADF REST runtime supports fetching a resource collection using a GET method using query parameters.

The resource collection may be queried using expressions that differ in syntax depending on the ADF REST framework version that has been registered for the ADF REST service client. For details about the ADF REST framework versions, see What You May Need to Know About Versioning the ADF REST Runtime Framework.

The following samples are based on two different versions of the `Departments` resource. The URL sample showing resource `11.0` reflects the query-by-example query parameter syntax supported only by version 1 of the ADF REST framework. While the URL sample showing resource `11.1`, reflects the rowmatch query parameter syntax supported in ADF REST framework version 2 (and later). In both framework scenarios, the samples fetch fields of the `Departments` resource collection.

> **Note:**
>
> For a REST web service request, reserved characters that appear in a query parameter value should be encoded. For example, the + character in a timestamp value must be encoded as `%2B`. Additionally, by default, resource and resource items names used in query parameter operations are case sensitive. If case insensitive filtering is desired, you may set the flag `restV1QueryCaseSensitive` in the `adf-config.xml` file to `false`. The default value of the flag is `true` which implies case sensitive filtering.

ADF REST Framework Version 2 (and later)

Starting with version 2 of the ADF REST framework, service clients may use an advanced query syntax, also known as rowmatch expressions, to fetch resources. For a complete description of the query syntax available in version 2 (and later), What You May Need to Know About the Advanced Query Syntax in ADF REST Framework Version 2.

The following sample fetches all departments with at least one employee whose salary is equal to 10000. This is an example of fetching a parent object (`Departments`) and filtering it by a child object attribute (`Employees.Salary`).

Request Example 1 Made With Framework Version 2

- **URL**

  `http://server/demo/rest/11.2/Departments?q=Employees.Salary = 10000`

- **HTTP Method**

  GET

- **Content-Type**

none

- **Payload**

  none

Response Example 1 From Framework Version 2

- **HTTP Code**

  `200`

- **Content-Type**

  `application/vnd.oracle.adf.resourcecollection+json`

- **Payload**

```
{
  "items" : [ {
    "DepartmentId" : 70,
    "DepartmentName" : "Public Relations",
    "links" : [ {
      "rel" : "self",
      "href" : "http://server/demo/rest/11.2/Departments/70",
      "name" : "Departments",
      "kind" : "item"
    }, {
      "rel" : "canonical",
      "href" : "http://server/demo/rest/11.2/Departments/70",
      "name" : "Departments",
      "kind" : "item"
    }, {
      "rel" : "child",
      "href" : "http://server/demo/rest/11.2/Departments/70/child/Employees",
      "name" : "Employees",
      "kind" : "collection"
    } ]
  }, {
    "DepartmentId" : 80,
    "DepartmentName" : "Sales",
    "links" : [ {
      "rel" : "self",
      "href" : "http://server/demo/rest/11.2/Departments/820",
      "name" : "Departments",
      "kind" : "item"
    }, {
      "rel" : "canonical",
      "href" : "http://server/demo/rest/11.2/Departments/80",
      "name" : "Departments",
      "kind" : "item"
    }, {
      "rel" : "child",
      "href" : "http://server/demo/rest/11.2/Departments/80/child/Employees",
      "name" : "Employees",
      "kind" : "collection"
    } ]
  } ],
  "count" : 2,
  "hasMore" : false,
  "limit" : 25,
  "offset" : 0,
  "links" : [ {
    "rel" : "self",
      "href" : "http://server/demo/rest/11.2/Departments",
      "name" : "Departments",
      "kind" : "collection"
    } ]
  }
```

The following sample fetches all departments with the numeric ID of `10` or a department name that begins with the letter "H".

Request Example 2 Made With Framework Version 2

- **URL**

  `http://server/demo/rest/11.1/Departments?q=DepartmentId = 10 or DepartmentName like 'H*'`

- **HTTP Method**

  GET

- **Content-Type**

  none

- **Payload**

  none

Response Example 2 From Framework Version 2

- **HTTP Code**

  `200`

- **Content-Type**

  `application/vnd.oracle.adf.resourcecollection+json`

- **Payload**

```
{
  "items" : [ {
    "DepartmentId" : 10,
    "DepartmentName" : "Administration",
    "links" : [ {
      "rel" : "self",
      "href" : "http://server/demo/rest/11.2/Departments/10",
      "name" : "Departments",
      "kind" : "item"
    }, {
      "rel" : "canonical",
      "href" : "http://server/demo/rest/11.2/Departments/10",
      "name" : "Departments",
      "kind" : "item"
    }, {
      "rel" : "child",
      "href" : "http://server/demo/rest/11.2/Departments/10/child/Employees",
      "name" : "Employees",
      "kind" : "collection"
    } ]
  }, {
    "DepartmentId" : 40,
    "DepartmentName" : "Human Resources",
    "links" : [ {
      "rel" : "self",
      "href" : "http://server/demo/rest/11.2/Departments/40",
      "name" : "Departments",
      "kind" : "item"
    }, {
      "rel" : "canonical",
      "href" : "http://server/demo/rest/11.2/Departments/40",
      "name" : "Departments",
      "kind" : "item"
    }, {
      "rel" : "child",
      "href" : "http://server/demo/rest/11.2/Departments/40/child/Employees",
      "name" : "Employees",
      "kind" : "collection"
    } ]
  } ],
  "count" : 2,
  "hasMore" : false,
  "limit" : 25,
  "offset" : 0,
  "links" : [ {
    "rel" : "self",
    "href" : "http://server/demo/rest/11.2/Departments",
    "name" : "Departments",
    "kind" : "collection"
  } ]
}
```

ADF REST Framework Version 1

Version 1 of the ADF REST framework supports a query-by-example syntax. No application configuration changes are required to use this syntax that is supported only in versions 1. For a description of the query syntax available in version 1 of the ADF REST framework, see GET Method Operations.

The following sample fetches departments assigned a `DepartmentId` value less than `30`.

> **Note:**
>
> In version 1 of the ADF REST framework, when you create a query with a string matching filter parameter and the string to match contains a query syntax reserved word (such as AND or OR), then the quoted string must be delimited by a space character to separate it from other parameters in the query expression. For example, the following query attempts to filter on the quoted string `‘Accounting and Finance’`. Since the string contains the reserved word AND, the string matching filter parameter requires a space before and after the single quotes to be viable in version 1.
>
> ```
> ?q=DepartmentName= 'Accounting and Finance'
> &fields=DepartmentName,Location
> ```
>
> Note that starting in framework version 2, the use of a space character is no longer required to delimit a string matching filter that contains a reserved word.

Request Made With Framework Version 1

- **URL**

`http://server/demo/rest/11.0/Departments?q=DepartmentId<30`

- **HTTP Method**

  GET

- **Content-Type**

  none

- **Payload**

  none

Response From Framework Version 1

- **HTTP Code**

  `200`

- **Content-Type**

  `application/vnd.oracle.adf.resourcecollection+json`

- **Payload**

```
{
  "items" : [ {
    "DepartmentId" : 10,
    "DepartmentName" : "Administration",
    "links" : [ {
      "rel" : "self",
      "href" : "http://server/demo/rest/11.0/Departments/10",
      "name" : "Departments",
      "kind" : "item"
    }, {
      "rel" : "canonical",
      "href" : "http://server/demo/rest/11.0/Departments/10",
      "name" : "Departments",
      "kind" : "item"
    }, {
      "rel" : "child",
      "href" : "http://server/demo/rest/11.0/Departments/10/child/Employees",
      "name" : "Employees",
      "kind" : "collection"
    } ]
  }, {
    "DepartmentId" : 20,
    "DepartmentName" : "Marketing",
    "links" : [ {
      "rel" : "self",
      "href" : "http://server/demo/rest/11.0/Departments/20",
      "name" : "Departments",
      "kind" : "item"
    }, {
      "rel" : "canonical",
      "href" : "http://server/demo/rest/11.0/Departments/20",
      "name" : "Departments",
      "kind" : "item"
    }, {
      "rel" : "child",
      "href" : "http://server/demo/rest/11.0/Departments/20/child/Employees",
      "name" : "Employees",
      "kind" : "collection"
    } ]
  } ],
  "count" : 2,
  "hasMore" : false,
  "limit" : 25,
  "offset" : 0,
  "links" : [ {
    "rel" : "self",
    "href" : "http://server/demo/rest/11.0/Departments",
    "name" : "Departments",
    "kind" : "collection"
  } ]
}
```

Fetching a Resource Item

The ADF REST runtime supports fetching an item of the resource collection using a GET method.

The following sample fetches version `11.0` of all fields of an instance of the `Departments` resource collection.

Request

- **URL**

  `http://server/demo/rest/11.0/Departments/50`

- **HTTP Method**

  GET

- **Content-Type**

  none

- **Payload**

  none

Response

- **HTTP Code**

  `200`

- **Content-Type**

  `application/vnd.oracle.adf.resourceitem+json`

- **Payload**

```
{
  "DepartmentId" : 50,
  "DepartmentName" : "Shipping",
  "links" : [ {
    "rel" : "self",
    "href" : "http://server/demo/rest/11.0/Departments/50",
    "name" : "Departments",
    "kind" : "item"
  }, {
    "rel" : "canonical",
    "href" : "http://server/demo/rest/11.0/Departments/50",
    "name" : "Departments",
    "kind" : "item"
  }, {
        "rel" : "child",
        "href" : "http://server/demo/rest/11.0/Departments/50/child/Employees",
        "name" : "Employees",
        "kind" : "collection"
      } ]
    }
```

Fetching Nested Child Resources

The ADF REST runtime supports retrieving nested resources using a GET method.

The payload structure of nested child resource differs depending on the ADF REST framework version that has been registered for the ADF REST service client. For details about the ADF REST framework versions, see What You May Need to Know About Versioning the ADF REST Runtime Framework.

The following samples are based on two different versions of the `Employees` resource. The URL samples showing resource `11.0` reflect a response payload structure supported by ADF REST framework versions 1 and 2. While the URL samples showing resource `11.1`, reflect the response payload structure supported in ADF REST framework version 3 (and later). In both framework scenarios, the samples fetch the `Employees` resource as a child of the `Departments` resource.

Note that the name used to identify the nested resource is determined at design time when the resource is created in the ADF Business Components model project. The default name for nested resources created in the model project is the destination view instance of the nested resource's defining view link accessor (such as `EmployeesView1` for the view link `DeptToEmpFkLink`). For this reason, it is a best practice for ADF REST resource developers to rename the resource to something more suitable as a URL parameter. In the following examples, the resource name was changed to `Employees` to follow the naming convention for the parent resource name `Departments`.

ADF REST Framework Version 3 (and later)

Starting with version 3 of the ADF REST framework, the ADF REST runtime returns a nested child resource in the response payload as a resource collection, instead of as an array of resource items. This functionality, available in framework version 3 (and later), allows service clients to make a request for additional records after determining how many items were left unfetched in the initial request. The attributes `hasMore` and `count` on the child resource indicate whether more items may be returned from the resource collection. For details about using the pagination attributes from the response payload when you opt into ADF REST framework version 3, see Paging a Resource Collection.

The following sample illustrates functionality for ADF REST framework version 3 (and later). The response payload represents the nested child resource as a resource collection, where the collection object includes the `hasMore` and `count` attributes. A link is provided should it be necessary to query the child resource for additional resource items. In this sample, items of the `Employees` child resource are fetched with a `count` of `3` in the payload. The response payload shows the `hasMore` attribute is `false`, suggesting that no items remain unfetched.

Request Made With Framework Version 3

- **URL**

  `http://server/demo/rest/11.1/Departments/50?expand=Employees`

- **HTTP Method**

  GET
- **Content-Type**

  none
- **Payload**

  none

Response From Framework Version 3

- **HTTP Code**

  `200`
- **Content-Type**

  `application/vnd.oracle.adf.resourceitem+json`
- **Payload**

```
{
  "DepartmentId" : 50,
  "DepartmentName" : "Shipping",
  "Employees" : {
    "items" : [ {
       "EmployeeId" : 120,
       "FirstName" : "Matthew",
       "LastName" : "Weiss",
       "Email" : "MWEISS",
       "JobId" : "ST_MAN",
       "DepartmentId" : 50,
       "Salary" : 8000,
       "links" : [ {
         "rel" : "self",
         "href" : "http://server/demo/rest/11.1/Departments/50/child/
Employees/120",
         "name" : "Employees",
         "kind" : "item"
       }, {
         "rel" : "canonical",
         "href" : "http://server/demo/rest/11.1/Departments/50/child/
Employees/120",
         "name" : "Employees",
         "kind" : "item"
       }, {
         "rel" : "parent",
         "href" : "http://server/demo/rest/11.1/Departments/50",
         "name" : "Departments",
         "kind" : "item"
       } ]
     }, {
       "EmployeeId" : 121,
       "FirstName" : "Adam",
       "LastName" : "Fripp",
       "Email" : "AFRIPP",
       "JobId" : "ST_MAN",
       "DepartmentId" : 50,
       "Salary" : 8200,
       "links" : [ {
         "rel" : "self",
         "href" : "http://server/demo/rest/11.1/Departments/50/child/
Employees/121",
         "name" : "Employees",
         "kind" : "item"
       }, {
         "rel" : "canonical",
         "href" : "http://server/demo/rest/11.1/Departments/50/child/
Employees/121",
         "name" : "Employees",
         "kind" : "item"
       }, {
         "rel" : "parent",
         "href" : "http://server/demo/rest/11.1/Departments/50",
         "name" : "Departments",
         "kind" : "item"
       } ]
     }, {
       ...
       } ]
     } ],
     "count" : 3,
     "hasMore" : false,
     "limit" : 25,
     "offset" : 0,
     "links" : [ {
         "rel" : "self",
         "href" : "http://server/demo/rest/11.1/Departments/50/child/
Employees",
         "name" : "Employees",
         "kind" : "collection"
     } ]
   },
   "links" : [ {
       "rel" : "self",
       "href" : "http://server/demo/rest/11.1/Departments/50",
       "name" : "Departments",
       "kind" : "item"
    }, {
       "rel" : "canonical",
       "href" : "http://server/demo/rest/11.1/Departments/50",
       "name" : "Departments",
       "kind" : "item"
    } ]
}
```

ADF REST Framework Version 1 or Version 2

Version 1 and version 2 of the ADF REST framework return the nested child resource expanded in the response payload as an array of resource items. If the resource collection being fetched is large, you may have to make several requests since the array of resource items has a limit.

The following samples illustrate functionality for ADF REST framework version 1 and version 2.

The first request sample (URL 1) retrieves a single child resource item identified by employee `120`. The URL parameter `child` identifies the relationship of the requested resource `Employees`.

The second request (URL 2) shows the use of the query parameter `expand` to ensure that all nested `Employees` resource items will be returned with `Departments` resource collection `50`.

The third request (URL 3) shows the use of accessor dot notation (for example, `Employees.JobHistory`) in combination with the query parameter `expand` to ensure that all nested `JobHistory` resource items will be returned with the `Employees` resource items for the `Departments` resource collection `80`.

Request Made With Framework Version 1 or Version 2

- **URL 1**

  `http://server/demo/rest/11.0/Departments/50/child/Employees/120`

- **URL 2**

  `http://server/demo/rest/11.0/Departments/50?expand=Employees`

- **URL 3**

  ```
  http://server/demo/rest/11.0/Departments/80?
  expand=Employees.JobHistory&onlyData=true
  ```

- **HTTP Method**

  GET

- **Content-Type**

  none

- **Payload**

  none

Response From Framework Version 1 or Version 2

- **HTTP Code**

  `200`

- **Content-Type**

  `application/vnd.oracle.adf.resourceitem+json`

- **Payload 1**

  ```
  {
    "EmployeeId" : 120,
    "FirstName" : "Matthew",
    "LastName" : "Weiss",
    "Email" : "MWEISS",
    "JobId" : "ST_MAN",
    "DepartmentId" : 50,
    "Salary" : 8000,
    "links" : [ {
      "rel" : "self",
      "href" : "http://server/demo/rest/11.0/Departments/50/child/Employees/
  120",
      "name" : "Employees",
      "kind" : "item"
    }, {
      "rel" : "canonical",
      "href" : "http://server/demo/rest/11.0/Departments/50/child/Employees/
  120",
      "name" : "Employees",
      "kind" : "item"
    }, {
      "rel" : "parent",
  ```

```
    "href" : "http://server/demo/rest/11.0/Departments/50",
    "name" : "Departments",
    "kind" : "item"
  } ]
}
```

- **Payload 2**

```
{
  "DepartmentId" : 50,
  "DepartmentName" : "Shipping",
  "Employees" : [ {
    "EmployeeId" : 120,
    "FirstName" : "Matthew",
    "LastName" : "Weiss",
    "Email" : "MWEISS",
    "JobId" : "ST_MAN",
    "DepartmentId" : 50,
    "Salary" : 8000,
    "links" : [ {
      "rel" : "self",
      "href" : "http://server/demo/rest/11.0/Departments/50/child/Employees/
120",
      "name" : "Employees",
      "kind" : "item"
    }, {
      "rel" : "canonical",
      "href" : "http://server/demo/rest/11.0/Departments/50/child/Employees/
120",
      "name" : "Employees",
      "kind" : "item"
    }, {
      "rel" : "parent",
      "href" : "http://server/demo/rest/11.0/Departments/50",
      "name" : "Departments",
      "kind" : "item"
    } ]
  }, {
    "EmployeeId" : 121,
    "FirstName" : "Adam",
    "LastName" : "Fripp",
    "Email" : "AFRIPP",
    "JobId" : "ST_MAN",
    "DepartmentId" : 50,
    "Salary" : 8200,
    "links" : [ {
      "rel" : "self",
      "href" : "http://server/demo/rest/11.0/Departments/50/child/Employees/
121",
      "name" : "Employees",
      "kind" : "item"
    }, {
      "rel" : "canonical",
      "href" : "http://server/demo/rest/11.0/Departments/50/child/Employees/
121",
      "name" : "Employees",
      "kind" : "item"
    }, {
      "rel" : "parent",
      "href" : "http://server/demo/rest/11.0/Departments/50",
      "name" : "Departments",
      "kind" : "item"
    } ]
  }, {
     ...
    } ]
  } ],
  "links" : [ {
    "rel" : "self",
    "href" : "http://server/demo/rest/11.0/Departments/50",
    "name" : "Departments",
    "kind" : "item"
  }, {
    "rel" : "canonical",
    "href" : "http://server/demo/rest/11.0/Departments/50",
    "name" : "Departments",
    "kind" : "item"
  } ]
}
```

- **Payload 3**

```
{
    "DepartmentId" : 80,
    "DepartmentName" : "Sales",
    "RelState" : null,
    "Employees" : [
        ...
        {
            "EmployeeId" : 176,
            "FirstName" : "Jonathon",
            "LastName" : "Taylor",
            "Email" : "JTAYLOR",
            "JobId" : "SA_REP",
            "DepartmentId" : 80,
            "Salary" : 8600,
            "CommissionPct" : 0.2,
            "JobHistory" : [
                {
                    "EmployeeId" : 176,
                    "StartDate" : "2011-03-24",
                    "EndDate" : "2012-12-31",
                    "JobId" : "SA_REP",
                    "DepartmentId" : 80
                },
                {
                    "EmployeeId" : 176,
                    "StartDate" : "2013-01-01",
                    "EndDate" : "2015-03-31",
                    "JobId" : "SA_MAN",
                    "DepartmentId" : 80
                }
            ]
        },
       ...
    ]
}
```

Sorting a Resource Collection

The ADF REST runtime supports sorting the fetched resource collection using a GET method.

Sorting a resource collection is performed using the **orderBy** query string parameter in combination with one or more attribute names. The following optional sort order flags may be associated with each attribute:

- **asc** sorts in ascending order. (Default)
- **desc** sorts in descending order.

The **orderBy** query string parameter format is:

```
<orderBy_attribute1_name>:<(asc/desc)>, <orderBy_attribute2_name>:<(asc/
desc)>
```

Example: `attribute1:desc,attribute2`

In order to perform case-insensitive sorting on a resource collection using a GET method, the **orderBy** query string parameter must follow this format:

`upper(<orderBy_attribute1_name>):<(asc)>` or
`lower(<orderBy_attribute2_name>)`

The following sample (URL1) fetches the `Departments` collection sorted by the `DepartmentName` attribute. The second sample (URL2) fetches the child `Employees` collection sorted by the `salary` attribute. Since the sort order flag is not specified for either request sample, the response is ascending order.

Request

- **URL 1**

  `http://server/demo/rest/11.0/Departments?orderBy=DepartmentName`
- **URL 2**

  `http://server/demo/rest/11.0/Departments/50/child/Employees?orderBy=Salary`
- **HTTP Method**

  GET
- **Content-Type**

  none
- **Payload**

  none

Response

- **HTTP Code**

  `200`
- **Content-Type**

  `application/vnd.oracle.adf.resourceitem+json`
- **Payload 1**

```
{
  "items" : [ {
    "DepartmentId" : 10,
    "DepartmentName" : "Administration",
    "links" : [ {
      "rel" : "self",
        "href" : "http://server/demo/rest/11.0/Departments/10",
        "name" : "Departments",
        "kind" : "item"
      }, {
        "rel" : "canonical",
        "href" : "http://server/demo/rest/11.0/Departments/10",
        "name" : "Departments",
        "kind" : "item"
      }, {
        "rel" : "child",
        "href" : "http://server/demo/rest/11.0/Departments/10/child/Employees",
        "name" : "Employees",
        "kind" : "collection"
      } ]
    }, {
      "DepartmentId" : 40,
      "DepartmentName" : "Human Resources",
      "links" : [ {
        "rel" : "self",
        "href" : "http://server/demo/rest/11.0/Departments/40",
        "name" : "Departments",
        "kind" : "item"
      }, {
        "rel" : "canonical",
        "href" : "http://server/demo/rest/11.0/Departments/40",
        "name" : "Departments",
        "kind" : "item"
      }, {
        "rel" : "child",
        "href" : "http://server/demo/rest/11.0/Departments/40/child/Employees",
        "name" : "Employees",
        "kind" : "collection"
      } ]
    }, {
      "DepartmentId" : 20,
      "DepartmentName" : "Marketing",
      "links" : [ {
        "rel" : "self",
        "href" : "http://server/demo/rest/11.0/Departments/20",
        "name" : "Departments",
        "kind" : "item"
      }, {
        "rel" : "canonical",
        "href" : "http://server/demo/rest/11.0/Departments/20",
        "name" : "Departments",
        "kind" : "item"
      }, {
        "rel" : "child",
        "href" : "http://server/demo/rest/11.0/Departments/20/child/Employees",
        "name" : "Employees",
        "kind" : "collection"
      } ]
    }, {
      "DepartmentId" : 30,
      "DepartmentName" : "Purchasing",
      "links" : [ {
        "rel" : "self",
        "href" : "http://server/demo/rest/11.0/Departments/30",
        "name" : "Departments",
        "kind" : "item"
      }, {
```

```
      "rel" : "canonical",
      "href" : "http://server/demo/rest/11.0/Departments/30",
      "name" : "Departments",
      "kind" : "item"
    }, {
      "rel" : "child",
      "href" : "http://server/demo/rest/11.0/Departments/30/child/Employees",
      "name" : "Employees",
      "kind" : "collection"
    } ]
  }, {
    "DepartmentId" : 50,
    "DepartmentName" : "Shipping",
    "links" : [ {
      "rel" : "self",
      "href" : "http://server/demo/rest/11.0/Departments/50",
      "name" : "Departments",
      "kind" : "item"
    }, {
      "rel" : "canonical",
      "href" : "http://server/demo/rest/11.0/Departments/50",
      "name" : "Departments",
      "kind" : "item"
    }, {
      "rel" : "child",
      "href" : "http://server/demo/rest/11.0/Departments/50/child/Employees",
      "name" : "Employees",
      "kind" : "collection"
    } ]
  } ],
  "count" : 5,
  "hasMore" : false,
  "limit" : 25,
  "offset" : 0,
  "links" : [ {
    "rel" : "self",
    "href" : "http://server/demo/rest/11.0/Departments",
    "name" : "Departments",
    "kind" : "collection"
  } ]
}
```

- **Payload 2**

```
{
  "items" : [ {
    "EmployeeId" : 132,
    "FirstName" : "TJ",
    "LastName" : "Olson",
    "Email" : "TJOLSON",
    "JobId" : "ST_CLERK",
    "DepartmentId" : 50,
    "Salary" : 2100,
    "links" : [ {
      "rel" : "self",
      "href" : "http://server/demo/rest/11.0/Employees/132",
      "name" : "Employees",
      "kind" : "item"
    }, {
      "rel" : "canonical",
      "href" : "http://server/demo/rest/11.0/Employees/132",
      "name" : "Employees",
      "kind" : "item"
    } ]
  }, {
    "EmployeeId" : 136,
    "FirstName" : "Hazel",
    "LastName" : "Philtanker",
    "Email" : "HPHILTAN",
    "JobId" : "ST_CLERK",
    "DepartmentId" : 50,
    "Salary" : 3100,
    "links" : [ {
      "rel" : "self",
      "href" : "http://server/demo/rest/11.0/Employees/136",
      "name" : "Employees",
      "kind" : "item"
    }, {
      "rel" : "canonical",
      "href" : "http://server/demo/rest/11.0/Employees/136",
      "name" : "Employees",
      "kind" : "item"
    } ]
  } ],
  "count" : 2,
  "hasMore" : false,
  "limit" : 25,
  "offset" : 0,
  "links" : [ {
    "rel" : "self",
    "href" : "http://server/demo/rest/11.0/Employees",
    "name" : "Employees",
    "kind" : "collection"
  } ]
}
```

Fetching a Resource with Grouped Context Information

ADF REST runtime supports fetching a resource using a GET method and retrieving the payload with resource fields and resource item context information, like links, grouped separately when your request is enabled to use ADF REST framework version 6.

Starting with version 6 of the ADF REST framework, a `@context` element is introduced in the response payload for resource requests to help you to more easily differentiate between resource fields and the context information of individual items. This element helps to organize the response payload of the GET request by grouping information together for each item.

Compared to earlier framework versions (5 and earlier), the following changes to a GET request payload result with framework version 6 or later:

- A new `@context` section appears below the list of fields of each item of the resource and contains identifying information for the item, as well as links.
- A new `key` element appears within the `@context` section and contains the unique identifier of the specific resource item as a string.
- The `links` section within the `@context` section no longer list any properties, like `changeIndicator`.

- The `changeIndicator` value is moved to `ETag`, which is under a `headers` element within the `@context` section.

The following sample fetches the `Departments` resource collection and three items with context information for each item under the `@context` element.

Collection Request Example Made With Framework Version 6

- **URL**

  `http://server/sample/rest/12.0/Departments`
- **HTTP Method**

  GET
- **Content-Type**

  none
- **Payload**

  none

Collection Response Example Made With Framework Version 6

- **HTTP Code**

  `200`
- **Content-Type**

  `application/vnd.oracle.adf.resourcecollection+json`
- **Payload**

```
{
  "items" : [ {
    "DepartmentId" : 10,
    "DepartmentName" : "Administration",
    "@context" : {
      "key" : "1",
      "headers" : {
        "ETag" : "ACED00C78"
      },
      "links" : [ {
        "rel" : "self",
        "href" : "http://server/demo/rest/12.0/Departments/10",
        "name" : "Departments",
        "kind" : "item"
      }, {
        "rel" : "canonical",
        "href" : "http://server/demo/rest/12.0/Departments/10",
        "name" : "Departments",
        "kind" : "item"
      }, {
        "rel" : "child",
        "href" : "http://server/demo/rest/12.0/Departments/10/child/
Employees",
        "name" : "Employees",
        "kind" : "collection"
      } ]
    }
  }, {
    "DepartmentId" : 20,
    "DepartmentName" : "Marketing",
    "@context" : {
      "key" : "2",
      "headers" : {
        "ETag" : "ACED00B74"
      },
      "links" : [ {
        "rel" : "self",
        "href" : "http://server/demo/rest/12.0/Departments/20",
        "name" : "Departments",
        "kind" : "item"
      }, {
        "rel" : "canonical",
        "href" : "http://server/demo/rest/12.0/Departments/20",
        "name" : "Departments",
        "kind" : "item"
      }, {
        "rel" : "child",
        "href" : "http://server/demo/rest/12.0/Departments/20/child/
Employees",
        "name" : "Employees",
        "kind" : "collection"
      } ]
    }
  }, {
    "DepartmentId" : 30,
    "DepartmentName" : "Purchasing",
    "@context" : {
      "key" : "3",
      "headers" : {
        "ETag" : "ACED001E8"
      },
      "links" : [ {
        "rel" : "self",
        "href" : "http://server/demo/rest/12.0/Departments/30",
        "name" : "Departments",
        "kind" : "item"
      }, {
        "rel" : "canonical",
        "href" : "http://server/demo/rest/12.0/Departments/30",
        "name" : "Departments",
        "kind" : "item"
      }, {
        "rel" : "child",
        "href" : "http://server/demo/rest/12.0/Departments/30/child/
Employees",
        "name" : "Employees",
        "kind" : "collection"
      } ]
    }
  } ],
  "count" : 3,
  "hasMore" : true,
  "limit" : 25,
  "offset" : 0,
  "links" : [ {
    "rel" : "self",
    "href" : "http://server/demo/rest/12.0/Departments",
    "name" : "Departments",
    "kind" : "collection"
  } ]
}
```

The following sample fetches all fields of an instance of the `Departments` resource collection with context information for each item under the `@context` element.

Item Request Example Made With Framework Version 6

- **URL**

  `http://server/demo/rest/12.0/Departments/50`
- **HTTP Method**

  GET
- **Content-Type**

  none
- **Payload**

  none

Item Response Example Made With Framework Version 6

- **HTTP Code**

  `200`
- **Content-Type**

  `application/vnd.oracle.adf.resourceitem+json`
- **Payload**

```
{
  "DepartmentId" : 50,
  "DepartmentName" : "Shipping",
  "@context" : {
    "key" : "1",
    "headers" : {
      "ETag" : "ACED00G26"
    },
    "links" : [ {
      "rel" : "self",
      "href" : "http://server/demo/rest/12.0/Departments/50",
      "name" : "Departments",
      "kind" : "item"
    }, {
      "rel" : "canonical",
      "href" : "http://server/demo/rest/12.0/Departments/50",
      "name" : "Departments",
      "kind" : "item"
    }, {
      "rel" : "child",
      "href" : "http://server/demo/rest/12.0/Departments/50/child/Employees",
      "name" : "Employees",
      "kind" : "collection"
    } ]
  }
}
```

Creating a Resource Item

The ADF REST runtime supports the HTTP POST method to create a resource and child resource item in the REST resource collection.

The ADF REST runtime supports the following creation use cases:

- Creating a resource item in collection.
- Creating a child resource item.

Creating a Resource Item in a Collection

The ADF REST runtime supports creating resource items on an existing resource collection using a POST method.

Before making a request with the POST method, determine whether the resource item is defined by an LOV-enabled attribute. If an LOV-enabled attribute is present and that attribute is defined as mandatory, you must first access the LOV resource on the resource collection to display the selection list to the end user. For details about how to access an LOV resource, see Retrieving LOV-Enabled Attribute Values When Creating a Resource Item.

The following sample creates a new `Departments` resource collection with a single item.

Request

- **URL**

  `http://server/demo/rest/11.0/Departments`

- **HTTP Method**

  POST

- **Content-Type**

  `application/vnd.oracle.adf.resourceitem+json`

- **Payload**

```
{
  "DepartmentId" : 15,
  "DepartmentName" : "NewDept"
}
```

Response

- **HTTP Code**

  `201`

- **Content-Type**

  `application/vnd.oracle.adf.resourceitem+json`

- **Location**

  `http://server/demo/rest/Departments/15`

- **Payload**

```
{
  "DepartmentId" : 15,
  "DepartmentName" : "NewDept",
  "links" : [ {
    "rel" : "self",
    "href" : "http://server/demo/rest/11.0/Departments/15",
    "name" : "Departments",
    "kind" : "item"
  }, {
    "rel" : "canonical",
    "href" : "http://server/demo/rest/11.0/Departments/15",
    "name" : "Departments",
    "kind" : "item"
  }, {
    "rel" : "child",
    "href" : "http://server/demo/rest/11.0/Departments/15/child/Employees",
    "name" : "Employees",
    "kind" : "collection"
  } ]
}
```

Creating a Child Resource Item

The ADF REST runtime supports creating ADF REST child resource items in one roundtrip using a POST method. Create will only succeed when both the parent and child do not exist.

The following samples create nested resource items. The first request sample (URL1) creates a child resource item identified by employee `999` in an existing `Departments` resource. The second request (URL2) creates the parent and child resources.

Request

- **URL 1**

  `http://server/demo/rest/11.0/Departments/15/child/Employees`

- **URL 2**

  `http://server/demo/rest/11.0/Departments`

- **HTTP Method**

  POST

- **Content-Type**

  `application/vnd.oracle.adf.resourceitem+json`

- **Payload 1**

  ```
  {
       "EmployeeId": 999,
       "FirstName": "New",
       "LastName": "Guy",
       "Email": "NGUY",
       "JobId": "SA_REP",
       "DepartmentId": 15,
       "Salary": 9999
  }
  ```

- **Payload 2**

  ```
  {
       "DepartmentId": 17,
       "DepartmentName": "NewerDept",
       "Employees": [
           {
               "EmployeeId": 99999,
               "FirstName": "Newer",
               "LastName": "Guy",
  ```

```
                "Email": "NRGUY",
                "JobId": "SA_MAN",
                "DepartmentId": 17,
                "Salary": 10001
            }
        ]
    }
```

Response

- **HTTP Code**

  201

- **Content-Type**

  application/vnd.oracle.adf.resourceitem+json

- **Location**

  http://server/demo/rest/11.0/Departments/15/child/Employees/999

- **Payload 1**

```
{
  "EmployeeId" : 999,
  "FirstName" : "New",
  "LastName" : "Guy",
  "Email" : "NGUY",
  "JobId" : "SA_REP",
  "DepartmentId" : 15,
  "Salary" : 9999,
  "links" : [ {
    "rel" : "self",
    "href" : "http://server/demo/rest/11.0/Departments/15/child/Employees/
999",
    "name" : "Employees",
    "kind" : "item"
  }, {
    "rel" : "canonical",
    "href" : "http://server/demo/rest/11.0/Departments/15/child/Employees/
999",
    "name" : "Employees",
    "kind" : "item"
  }, {
    "rel" : "parent",
    "href" : "http://server/demo/rest/11.0/Departments/15",
    "name" : "Departments",
    "kind" : "item"
  }, {
    "rel" : "lov",
    "href" : "http://server/demo/rest/11.0/Departments/15/child/
Employees/999/lov/JobsView1",
    "name" : "JobsView1",
    "kind" : "collection"
  } ]
}
```

- **Location**

  http://server/demo/rest/11.0/Departments/17

- **Payload 2**

```
{
  "DepartmentId" : 17,
  "DepartmentName" : "NewerDept",
  "Employees" : [ {
    "EmployeeId" : 99999,
    "FirstName" : "Newer",
    "LastName" : "Guy",
    "Email" : "NRGUY",
    "JobId" : "SA_MAN",
    "DepartmentId" : 17,
    "Salary" : 10001,
    "links" : [ {
      "rel" : "self",
      "href" : "http://server/demo/rest/11.0/Departments/17/child/Employees/
99999",
      "name" : "Employees",
      "kind" : "item"
    }, {
      "rel" : "canonical",
      "href" : "http://server/demo/rest/11.0/Departments/17/child/Employees/
99999",
      "name" : "Employees",
      "kind" : "item"
    }, {
      "rel" : "parent",
      "href" : "http://server/demo/rest/11.0/Departments/17",
      "name" : "Departments",
      "kind" : "item"
    }, {
      "rel" : "lov",
      "href" : "http://server/demo/rest/11.0/Departments/17/child/Employees/
99999/lov/JobsLOV",
      "name" : "JobsLOV",
      "kind" : "collection"
    } ]
  } ],
  "links" : [ {
    "rel" : "self",
    "href" : "http://server/demo/rest/11.0/Departments/17",
    "name" : "Departments",
    "kind" : "item"
  }, {
    "rel" : "canonical",
    "href" : "http://server/demo/rest/11.0/Departments/17",
    "name" : "Departments",
    "kind" : "item"
  } ]
}
```

Updating a Resource Item

The ADF REST runtime supports the HTTP PATCH method to update a resource in the REST resource collection.
The ADF REST runtime supports updating resource items using a PATCH method. Update will only succeed when the row already exists.

The following sample updates `Departments` item `15`, where `DepartmentName` is changed in the request payload.

Request

- **URL**

  `http://server/demo/rest/11.0/Departments/15`

- **HTTP Method**

  PATCH

- **Content-Type**

  `application/vnd.oracle.adf.resourceitem+json`

- **Payload**

  ```
  {
    "DepartmentId" : 15,
    "DepartmentName" : "UpdatedDeptName"
  }
  ```

Response

- **HTTP Code**

  `200`

- **Content-Type**

  `application/vnd.oracle.adf.resourceitem+json`

- **Payload**

  ```
  {
    "DepartmentId" : 15,
    "DepartmentName" : "UpdatedDeptName",
    "links" : [ {
      "rel" : "self",
      "href" : "http://server/demo/rest/11.0/Departments/15",
      "name" : "Departments",
      "kind" : "item"
    }, {
      "rel" : "canonical",
      "href" : "http://server/demo/rest/11.0/Departments/15",
      "name" : "Departments",
      "kind" : "item"
    }, {
      "rel" : "child",
      "href" : "http://server/demo/rest/11.0/Departments/15/child/Employees",
      "name" : "Employees",
      "kind" : "collection"
    } ]
  }
  ```

Updating or Creating Resource Items (Upsert)

Using a POST method with Upsert mode enabled, the ADF REST runtime supports updating resource items that exist, and if not, creating the resource items.

You use a POST method with header variable `Upsert/true` to enable the Upsert functionality that creates an ADF REST resource if this resource does not exist or updates the resource if the resource exists. You select the Create and Update checkboxes on the user interface to enable this action (Upsert). If the header variable `Upsert/true` is not provided, or is set to `Upsert/false`, then the Upsert functionality is disabled. Also, if only the Create checkbox is selected or only the Update checkbox is selected, then POST creates or updates a resource if permissions for these actions are allowed. Note that Upsert will work as intended only if both Create and Update options are enabled on the resource.

The following sample creates (through Upsert) a new Department resource item with a child Employee resource item.

Request

- **URL**

  `http://server/demoapp/rest/1.0/Departments`

- **HTTP Method**

  `POST`

- **Content-Type**

  `application/vnd.oracle.adf.resourceitem+json`

- **HTTP Header**

  `Upsert-Mode/true`

- **Payload**

```
{
    "Deptno": 80,
    "Dname": "ENG80",
    "Emp": [{
        "Empno": 8080,
        "Ename": "Smith",
        "Mgr": 8080
    }]
}
```

Response

- **HTTP Code**

  `200`

- **Content-Type**

  `application/vnd.oracle.adf.resourceitem+json`

- **Content Encoding**

  `Null`

- **Payload**

```
{
    "Deptno": 80,
    "Dname": "ENG80",
    "Emp": [{
        "Empno": 8080,
        "Ename": "Smith",
        "Mgr": 8080
    }]
}
```

The following sample demonstrates the update process (through Upsert). In the response payload, a Department (Dept) resource item's AltKey namely Dname is utilized. Similarly, the Employee (Emp) child resource item's Altkey namely Email is utilized. The Dname Altkey (Dname=ENG80) is used to locate and load the required

Department (Dept 80). The request then updates this Department's Location (Loc). The Email Altkey is used to locate and load the Employee child resource item (Emp 8080) and the request payload updates this Employee child resource item's Sal and Job attributes. The request payload also contains a second Employee child resource item (Emp 9080). The Ename key is used to locate this second Employee child resource item. If found, this second Employee child resource item's Sal and Job attributes get updated. If this second Employee child resource item does not exist, the request payload creates this second Employee child resource item.

Request

- **URL**

  ```
  http://server/demoapp/rest/1.0/Departments
  ```

- **HTTP Method**

  ```
  POST
  ```

- **Content-Type**

  ```
  application/vnd.oracle.adf.resourceitem+json
  ```

- **HTTP Header**

  ```
  Upsert-Mode/true
  ```

- **Payload**

  ```
  {
      "Dname": "ENG80",
      "Loc": "HQ-altkey",
      "Emp": [{
              "Email": "smith@xyz.com",
              "Sal": 8000,
              "Job": "ENGINEER1"
          },
          {
              "Empno": 9080,
              "Ename": "John",
              "Sal": 9000,
              "Job": "SALES1"
          }
      ]
  }
  ```

Response

- **HTTP Code**

  ```
  200
  ```

- **Content Type**

  ```
  application/vnd.oracle.adf.resourceitem+json
  ```

- **Content Encoding**

  ```
  Null
  ```

- **Payload**

  ```
  {
      "Deptno": 80,
      "Dname": "ENG80",
      "Loc": "HQ-altkey",
  ```

```
        "TrEmpno": null,
        "Emp": [{
                "Empno": 8080,
                "Ename": "Smith",
                "Job": "ENGINEER1",
                "Email": "smith@xyz.com",
                "Hiredate": null,
                "Sal": 8000,
                "Comm": null,
                "Deptno": 80
            },
            {
                "Empno": 9080,
                "Ename": "John",
                "Job": "SALES1",
                "Mgr": 9080,
                "Hiredate": null,
                "Sal": 9000,
                "Comm": null,
                "Deptno": 80
            }
        ]
    }
```

Deleting a Resource Item

The ADF REST runtime supports the HTTP DELETE method to delete a resource in the REST resource collection.
The ADF REST runtime supports deleting resource items using a DELETE method. The framework does not currently support deleting a resource collection.

The following sample (URL1) deletes employee ID `99999` of the `Employees` resource collection as a child of the `Departments` resource item `17`. The second request URL deletes the `Departments` resource item `17`.

Request

- **URL 1**

  `http://server/demo/rest/11.0/Departments/17/child/Employees/99999`
- **URL 2**

  `http://server/demo/rest/11.0/Departments/17`
- **HTTP Method**

  DELETE
- **Content-Type**

  none
- **Payload**

  none

Response

- **HTTP Code**

  `204`
- **Content-Type**

none

- **Payload**

  none

Checking for Data Consistency

When updating or retrieving ADF REST resource items using the HTTP method, you can check data consistency by generating an entity tag with precondition headers so that the resource item matches the server side resource state before updating or retrieving.
The ADF REST runtime supports generating an entity tag (ETag) in the response header when the requested resource has data consistency check enabled. Data consistency checking is enabled by the ADF Business Components developer who must configure a change-indicator attribute on the entity object backing the resource.

When entity change indicators are configured on the entity object backing the resource, the ADF REST runtime will assign a unique value to indicate the state of each resource on the server side. At runtime, when the row underlying the server side resource changes, Oracle ADF assigns a new state value to the ETag. The following header shows the ETag returned with a request to retrieve a `Departments` resource item.

```
HTTP/1.1 200 OK
Cache-Control: no-cache, no-store, must-revalidate
Location:
Content-Length: 1069
Content-Type: application/json
ETag: "ACED0005737203720013626163 6C6520136261636C65237200136261636C652"
Content-Encoding:
Link: <http://server/demo/rest/11.0/Departments/
10>;rel="self";kind="item";name="Departments"
```

> **Note:**
>
> Note that the ETag and data consistency checking are not automatically enabled for REST resources. To support generating ETag values, the ADF Business Components developer must configure a change-indicator enabled attribute on the entity object backing the resource to be checked. For details about configuring change indicator attributes for entity objects, see How to Protect Against Losing Simultaneously Updated Data.

The service client can use the ETag value returned in the header response of each resource item to create subsequent requests that contain precondition headers (`If-Match`/`If-None-Match`). Based on the specified ETag and the precondition, the server will evaluate the current resource state and match against the provided ETag. If the precondition is satisfied, the requested operation is executed; otherwise, a `412` error is returned. The error payload will contain the current resource in the server side and the header will also reflect the current ETag value.

To support testing ETag values, the ADF REST framework provides the following precondition header fields. Usage of these precondition fields forces the framework to compare a supplied ETag value against the ETag values of previously requested items.

- Verify that the client is providing a state (obtained from a previous resource item response) that matches the current state on the server:

  `If-Match: "<ETag value from resource item response>"`

- Verify that the client is providing a state (obtained from a previous resource item response) that does **not** match the current state on the server.

  `If-None-Match: "<ETag value from resource item response>"`

The following are typical use cases when checking for data consistency:

- Check that the resource item matches the server side resource state before updating
- Retrieve the resource item using the server side resource state when none of the requested items match any previously requested items

While these use cases involve GET and PATCH methods, the precondition header and ETag value can be used to check that any HTTP method operation will be applied to the current state of the resource.

When retrieving a resource collection, an additional custom property `changeIndicator` will appear in the response payload of resources with data consistency enabled. This property contains the current ETag value of each resource item in the requested collection. The following sample illustrates the `changeIndicator` property in the `links` section of a `Departments` resource collection. The presence of ETag values in the resource collection payload is a convenience for the service client that can reduce the number of requests to obtain the ETag from individual resource items.

Note also the presence of the `RelState` attribute in the response. This is the name of the change-indicator attribute that the ADF Business Components developer configured on the entity object to support data consistency checks. The value of the attribute reflects the number of times the state has been updated.

```
{
  "items" : [ {
    "DepartmentId" : 10,
    "DepartmentName" : "Administration",
    "RelState" : 1,
    "links" : [ {
      "rel" : "self",
      "href" : "http://server/demo/rest/11.0/Departments/10",
      "name" : "Departments",
      "kind" : "item",
      "properties" : {
        "changeIndicator" :
"ACED0005737200136A6176612E7574696C2E41727261794C69737

47881D21D99C7619D03000149000473697A6578700000000177040000000173720018 6F721

636C652E6A626F2E646F6D61696E2E4E7564A362286F0200015B00046461746174000 25B45
        2787075720025B42ACF317F8060854E00200007870"
      }
    "links" : [ {
      "rel" : "self",
      "href" : "http://server/demo/rest/11.0/Departments/10",
      "name" : "Departments",
      "kind" : "item",
    }, {
      "rel" : "canonical",
      "href" : "http://server/demo/rest/11.0/Departments/10",
      "name" : "Departments",
      "kind" : "item",
      "properties" : {
        "changeIndicator" :
"ACED0005737200136A6176612E7574696C2E149000473697A6578
        7000000001770400000001737200 1B6F7261636C652E6A626F2E646F60000C78"
      }
    }, {
      "rel" : "child",
      "href" : "http://server/demo/rest/11.0/Departments/10/child/Employees",
      "name" : "Employees",
      "kind" : "collection",
    } ]
  }, {
    "DepartmentId" : 20,
    "DepartmentName" : "Marketing",
    "links" : [ {
      "rel" : "self",
      "href" : "http://server/demo/rest/11.0/Departments/20",
      "name" : "Departments",
      "kind" : "item",
      "properties" : {
        "changeIndicator" :
"ACED0005737200136A6176612E7574696C2E41727261794C6973747881D21D99C7619D0300014900
0473697A6578700000000177040000000173720 0186F7261636C652E6A626F2E646F6D61696E2E4E7
56D626572A5B1371914E0BFDA0200014900096D48617368436F6465787200116F7261636C652E7371
6C2E4E554D424552E90466EE632BE1D502000078720 0106F7261636C652E73716C2E446174756D407
8F514A362286F0200015B000464617461740 0025B427870757200025B42ACF317F8060854E0020000
787000000002C10A0000000078"
      }
    }, {
      "rel" : "canonical",
      "href" : "http://server/demo/rest/11.0/Departments/20",
      "name" : "Departments",
      "kind" : "item"
    }, {
      "rel" : "child",
      "href" : "http://server/demo/rest/11.0/Departments/20/child/Employees",
      "name" : "Employees",
      "kind" : "collection"
    } ]
  }, {
      ...
    } ]
  } ],
  "count" : 5,
  "hasMore" : true,
  "limit" : 25,
  "offset" : 0,
  "links" : [ {
    "rel" : "self",
    "href" : "http://server/demo/rest/11.0/Departments",
    "name" : "Departments",
    "kind" : "collection"
  } ]
}
```

Checking for Data Consistency When Updating ADF REST Resource Items

The ADF REST runtime support checking for data consistency when using a PATCH method to update resource items backed by an ADF Business Components entity object with a change-indicator attribute enabled.

To check for data consistency using the ETag header and conditional header fields:

1. Query one or more resource items and, for each returned resource item, obtain the ETag value from the `changeIndicator` property in the `properties` section of the response. When querying multiple resource items, there will not be a single ETag response header. Instead, the ETag for each of the items in the response will be in the `properties` section.

```
HTTP/1.1 200 OK
Cache-Control: no-cache, no-store, must-revalidate
Location:
Content-Length: 861
Content-Type: application/json
ETag: "responseETag123"
Content-Encoding:
Link: <http://server/demo/rest/11.0/Departments/
10>;rel="self";kind="item";name="Departments"
Set-Cookie: JSESSIONID=jXvsJ1GpdkFJV5Jh0yk7D72vPZ42t8tLYDg74NRKFQzXdnsjG9vv!
1113104013; path=/; HttpOnly
X-ORACLE-DMS-ECID:
51f1ff4535af720c:-7e156247:148ec9eeb3b:-8000-00000000000001ad
X-Powered-By: Servlet/2.5 JSP/2.1

{
  "DepartmentId" : 10,
  "DepartmentName" : "Administration",
  "links" : [ {
    "rel" : "self",
    "href" : "http://server/demo/rest/11.0/Departments/10",
    "name" : "Departments",
    "kind" : "item",
    "properties" : {
      "changeIndicator" : "responseETag123"
    }
  }, {
    "rel" : "canonical",
    "href" : "http://server/demo/rest/11.0/Departments/10",
    "name" : "Departments",
    "kind" : "item"
  }, {
    "rel" : "child",
    "href" : "http://server/demo/rest/11.0/Departments/10/child/Employees",
    "name" : "Employees",
    "kind" : "collection"
  } ]
}
```

2. Update the resource item, using a PATCH request and check for data consistency by supplying the following conditional header field:

- `If-Match: "<`*`ETag value from resource item response`*`>"` to verify that the state of a requested resource item is current with the previous resource item response.

The following sample updates the `DepartmentName` field of the Departments `10` resource item when the `If-Match` precondition test is satisfied. In the first request (Request 1), the ETag value `responseETag123` is identical to the ETag of the current `Departments 10` resource item on the server side, indicating that the state of the resource item is consistent with the server side. Consequently, the update to `DepartmentName` is allowed.

In the subsequent request (Request 2), however, the ETag supplied in the `If-Match` precondition is unchanged and no longer matches the new ETag value the server has for the `Departments 10` resource item. As a consequence of the stale ETag value used in the second request, the update fails with an HTTP code `412`, indicating the precondition test failed, and the current ETag value `responseETag567` is returned in the response header. This occurs in production web applications when multiple users simultaneously access the same resource item. For example, when user 1 and user 2 both query the same item, the item has, for example, ETag value 1. Then, if user 1 successfully updates the item with ETag value 1, and user 2 attempts to update the same item with ETag value 1, the attempt will fail.

Request 1

- **URL 1**

  ```
  http://server/demo/rest/11.0/Departments/10
  ```

- **HTTP Method**

  PATCH

- **Precondition 1**

  ```
  If-Match: "responseETag123"
  ```

- **Content-Type**

  ```
  application/vnd.oracle.adf.resourceitem+json
  ```

- **Payload 1**

  ```
  {
    "DepartmentName" : "FirstAttempt_NewDepartmentName"
  }
  ```

Response 1

- **HTTP Code**

  ```
  200
  ```

- **Content-Type**

  ```
  application/vnd.oracle.adf.resourceitem+json
  ```

- **ETag**

  ```
  responseETag567
  ```

- **Payload 1**

  ```
  {
      "DepartmentId" : 10,
      "DepartmentName" : "FirstAttempt_NewDepartmentName",
  ```

```
        "RelState" : null,
        "links" : [ {
          "rel" : "self",
          "href" : "http://server/demo/rest/11.0/Departments/10",
          "name" : "Departments",
          "kind" : "item",
          "properties" : {
            "changeIndicator" : "responseETag567"
          }
        }, {
          "rel" : "canonical",
          "href" : "http://server/demo/rest/11.0/Departments/10",
          "name" : "Departments",
          "kind" : "item"
        }, {
          "rel" : "child",
          "href" : "http://server/demo/rest/11.0/Departments/10/child/Employees",
          "name" : "Employees",
          "kind" : "collection"
        } ]
    }
```

Request 2

- **URL 2**

  `http://server/demo/rest/11.0/Departments/10`

- **HTTP Method**

  PATCH

- **Precondition 2**

  `If-Match: "staleETag789"`

- **Content-Type**

  `application/vnd.oracle.adf.resourceitem+json`

- **Payload 2**

  ```
  {
    "DepartmentName" : "SecondAttempt_NewDepartmentName"
  }
  ```

Response 2

- **HTTP Code**

  412 (Precondition failed)

- **Content-Type**

  `application/vnd.oracle.adf.resourceitem+json`

- **ETag**

  `responseETag567`

- **Payload 2**

  ```
  {
      "DepartmentId" : 10,
      "DepartmentName" : "FirstAttempt_NewDepartmentName",
      "RelState" : null,
      "links" : [ {
  ```

```
      "rel" : "self",
      "href" : "http://server/demo/rest/11.0/Departments/10",
      "name" : "Departments",
      "kind" : "item",
      "properties" : {
        "changeIndicator" : "responseETag567"
      }
    }, {
      "rel" : "canonical",
      "href" : "http://server/demo/rest/11.0/Departments/10",
      "name" : "Departments",
      "kind" : "item"
    }, {
      "rel" : "child",
      "href" : "http://server/demo/rest/11.0/Departments/10/child/Employees",
      "name" : "Employees",
      "kind" : "collection"
    } ]
}
```

Checking for Data Consistency When Retrieving ADF REST Resource Items

The ADF REST runtime supports checking for data consistency when using a GET method to retrieve resource items backed by an ADF Business Components entity object with a change-indicator attribute enabled.

To check for data consistency using the ETag header and conditional header fields:

1. Query one or more business object items and, for each returned resource item, obtain the ETag value from the `changeIndicator` property in the `properties` section of the response.

   When querying multiple business object items, there will not be a single ETag response header. In the case of ADF REST framework versions 5 and earlier, the ETag for each of the items in the response will be in the `properties` section. In the case of framework versions 6 and later, the ETag for each of the items will be in the `@context` element that serves to group all the information for an item in one section. The following sample shows the response with a framework version before version 6 enabled.

   ```
   HTTP/1.1 200 OK
   Cache-Control: no-cache, no-store, must-revalidate
   Location:
   Content-Length: 861
   Content-Type: application/json
   ETag: "responseETag123"
   Content-Encoding:
   Link: <http://server/demo/rest/11.0/Departments/
   10>;rel="self";kind="item";name="Departments"
   Set-Cookie: JSESSIONID=jXvsJ1GpdkFJV5Jh0yk7D72vPZ42t8tLYDg74NRKFQzXdnsjG9vv!
   1113104013; path=/; HttpOnly
   X-ORACLE-DMS-ECID:
   51f1ff4535af720c:-7e156247:148ec9eeb3b:-8000-00000000000001ad
   X-Powered-By: Servlet/2.5 JSP/2.1

   {
     "DepartmentId" : 10,
     "DepartmentName" : "Administration",
   ```

```
  "links" : [ {
    "rel" : "self",
    "href" : "http://server/demo/rest/11.0/Departments/10",
    "name" : "Departments",
    "kind" : "item",
    "properties" : {
      "changeIndicator" : "ACED0005737200136A6176612E7574696C2E4"
    }
  }, {
    "rel" : "canonical",
    "href" : "http://server/demo/rest/11.0/Departments/10",
    "name" : "Departments",
    "kind" : "item"
  }, {
    "rel" : "child",
    "href" : "http://server/demo/rest/11.0/Departments/10/child/Employees",
    "name" : "Employees",
    "kind" : "collection"
  } ]
}
```

2. Query one or more business object items and check for data consistency by supplying the following conditional header field

   :

   - `If-None-Match: "`*`<ETag value from resource item response>`*`"` to verify that the state of none of the previously requested resource items is current with the resource item request.

The following sample retrieves the Departments `10` resource item when the `If-None-Match` precondition test is satisfied. In the first request (Request 1), the ETag value `responseETag123` matches the ETag of the previously requested `Departments 10` resource item on the server side, indicating that the state of the resource item is consistent with the server side. Consequently, the precondition fails and there is no need to return a newer Departments `10` resource item. The request returns with an HTTP code `304`, indicating the state on the server has not been modified.

In the subsequent request (Request 2), however, the ETag `unmatchedETagXYZ` supplied in the `If-None-Match` precondition does not exist on the server. As a consequence, the precondition succeeds and the `Departments 10` resource item is retrieved. The request returns an HTTP code `200`, indicating the state had changed, and the current (unchanged) ETag value `responseETag123` is returned in the response header.

Request 1

- **URL 1**

  `http://server/demo/rest/11.0/Departments/10`

- **HTTP Method**

  GET

- **Precondition 1**

  `If-None-Match: "responseETag123"`

- **Content-Type**

  none

- **Payload**

none

Response 1

- **HTTP Code**

  `304` state not modified

- **Content-Type**

  none

- **Payload 1**

  none

Request 2

- **URL 2**

  `http://server/demo/rest/11.0/Departments/10`

- **HTTP Method**

  GET

- **Precondition 2**

  `If-None-Match: "unmatchedETagXYZ"`

- **Content-Type**

  none

- **Payload**

  none

Response 2

- **HTTP Code**

  `200`

- **Content-Type**

  `application/vnd.oracle.adf.resourceitem+json`

- **ETag**

  `responseETag123`

- **Payload 2 (Made With Framework Version 5 or Earlier)**

```
{
    "DepartmentId" : 10,
    "DepartmentName" : "Administration",
    "RelState" : null,
    "links" : [ {
      "rel" : "self",
      "href" : "http://server/demo/rest/11.0/Departments/10",
      "name" : "Departments",
      "kind" : "item",
      "properties" : {
        "changeIndicator" : "responseETag123"
      }
    }, {
      "rel" : "canonical",
      "href" : "http://server/demo/rest/11.0/Departments/10",
      "name" : "Departments",
      "kind" : "item"
    }, {
      "rel" : "child",
      "href" : "http://server/demo/rest/11.0/Departments/10/child/Employees",
      "name" : "Employees",
      "kind" : "collection"
    } ]
}
```

Payload 2 (Made With Framework Version 6 or Later)

```
{
    "DepartmentId" : 10,
    "DepartmentName" : "Administration",
    "RelState" : null,
    "@context" : {
       "key" : "AB8765BCD",
       "headers" : {
          "ETag" : "responseETag123."
       },
    "links" : [ {
      "rel" : "self",
      "href" : "http://server/demo/rest/11.0/Departments/10",
      "name" : "Departments",
      "kind" : "item",
    }, {
      "rel" : "canonical",
      "href" : "http://server/demo/rest/11.0/Departments/10",
      "name" : "Departments",
      "kind" : "item"
    }, {
      "rel" : "child",
      "href" : "http://server/demo/rest/11.0/Departments/10/child/Employees",
      "name" : "Employees",
      "kind" : "collection"
    } ]
}
```

Working with Attachments

The ADF REST runtime supports content streaming of custom content that is not contained in the payload by using a link to the content. You can create LOB content at the same time that you are creating or retrieving a resource item using the HTTP methods.
The ADF REST runtime supports content streaming for BLOB or CLOB attributes.

LOB attributes may be handled in the following ways.

- Linked to as custom content that is not contained in the payload itself.

  This is the default for BLOB and CLOB attributes and specified in the response payload using the `enclosure` link type to point to a resource that cannot be represented with the supported payload types (such as `image/png`).
- Encoded into `Base64` string format and contained in the request payload itself.

Currently, advanced features (like support for Chunked encoding) are not supported by the ADF REST runtime. Developers working on an enterprise architecture, may wish to investigate the use of specialized content management systems (such as Oracle WebCenter Content).

Instead of showing the LOB attribute in the `attribute` section, the response payload always contains an `enclosure` link to the content. The following describe for an `Employees` resource item defines the BLOB attribute `Picture` with a default `requestType` of `application/octet-stream`. The `enclosure` link for the resource item appears in the `items` section. The `actions` section identifies the operations that may be used to manipulate the content.

```
{
  "Resources" : {
    "Employees" : {
      "discrColumnType" : false,
      "attributes" : [ {
       ...
      }, {
        "name" : "Picture",
        "type" : "attachment",
        "updatable" : true,
        "mandatory" : false,
        "queryable" : false,
        "actions" : [ {
          "name" : "delete",
          "method" : "DELETE"
        }, {
          "name" : "get",
          "method" : "GET",
          "responseType" : [ "application/octet-stream" ]
        } ]
      } ],
      "item" : {
        "links" : [ {
         ...
        }, {
          "rel" : "enclosure",
          "href" : "http://server/demo/rest/11.0/Employees/101/enclosure/
Picture",
          "name" : "Picture",
          "kind" : "other"
        } ],
        "actions" : [ {
        } ]
      },
      "links" : [ {
        ...
      } ]
  }
}
```

The attribute content type of the resource item can be assigned by the ADF REST resource developer by configuring the `contentType` property as a custom property of the LOB-type attribute. For example, when working with PNG image files, the following content type can assigned in advance:

- **Custom Property**: `contentType`
- **Value**: `image/png`

In some use cases the attribute value itself is a link to some external content. When an attribute is configured this way, it is not only shown in the resource item payload but a link that points to the external content is also created. In the resource description, only the GET action will be available for the external link. You can modify the URL contained in the attribute value by making a request with an update on the resource item.

An attribute can be configured by the ADF REST resource developer to generate a link by adding the following attribute:

- **Name**: `inputHandler`
- **Value**: `oracle.adf.internal.model.rest.core.binding.inputHandler.LinkInputHandler`

Streaming Attachments Using a Resource Item Enclosure Link

The ADF REST runtime supports streaming content that cannot be contained in the payload of a resource item by using a link to the content. Supported methods on the linked content include GET and DELETE. Note that creating LOB content at the same time that you create a resource item (using a POST method) requires encoding the content as `base64`. A sample of how to embed content using `base64` can be found in Replacing LOB Content Using Base64.

To stream content from a link:

1. Retrieve the resource item and locate the `enclosure` link for the desired attribute. The `item` section defines the available links for the attribute.

   Alternatively, you can execute the resource describe to identify the enclosure link generated for resource items. In general, the resource describe contains other useful information not returned in the GET response payload, including the expected response type (`image/png`, for example) and supported actions for the attribute.

2. Execute an HTTP operation (GET or DELETE) using the link to the LOB content.

The following resource item for `Employees 101` shows the `enclosure` link for the `Picture` attribute in the `links` section.

> **Note:**
>
> The URL that you specify for `enclosure` should be the enclosure link for the image and not the URL of the image.

```
{
  "EmployeeId" : 101,
  "FirstName" : "Neena",
  "LastName" : "Smith",
  "Email" : "NSMITH",
  "JobId" : "AD_VP",
  "DepartmentId" : 90,
  "Salary" : 2000,
  "links" : [ {
    "rel" : "self",
    "href" : "http://server/demo/rest/11.0/Employees/101",
    "name" : "Employees",
    "kind" : "item"
  }, {
    "rel" : "canonical",
    "href" : "http://server/demo/rest/11.0/Employees/101",
    "name" : "Employees",
    "kind" : "item"
  }, {
    "rel" : "lov",
    "href" : "http://server/demo/rest/11.0/Employees/101/lov/JobsLOV",
    "name" : "JobsLOV",
    "kind" : "collection"
  }, {
    "rel" : "enclosure",
    "href" : "http://server/demo/rest/11.0/Employees/101/enclosure/Picture",
    "name" : "Picture",
    "kind" : "other"
  } ]
}
```

For example, the following sample streams the PNG image associated with employee 101.

Note that by default the request type is `application/octet-stream` to support a variety of media types. If the ADF REST resource developer defined the custom property `contentType`, the specified request type appears in the resource item describe.

Request

- **URL**

  `http://server/demo/rest/11.0/Employees/101/enclosure/Picture`

- **HTTP Method**

  GET

- **Content Type**

  none

- **Payload**

  none

Response

- **HTTP Code**

  `200`

- **Content-Type**

  `image/png`

- **Payload**

  content streamed

Replacing LOB Content Using Base64

The ADF REST runtime allows creating and updating LOB content using a JSON payload when the content is represented in `base64` string format.

For example, the following sample replaces a PNG image for employee 101. In this use case, the Picture attribute must be represented in the request payload as String encoded in base64. Because resource item 101 already exists and only the Picture attribute is being manipulated, no other attributes need to be specified.

Note that the response payload contains the enclosure link to the Picture attribute.

Request

- **URL**

  `http://server/demo/rest/11.0/Employees/101`

- **HTTP Method**

  PATCH

- **Content-Type**

  `application/vnd.oracle.adf.resourceitem+json`

- **Payload**

```
{
  "Picture" : "/9j/4AAQSkZJRgABAAEAYABgAAD//..."
}
```

Response

- **HTTP Code**

  `200`

- **Content-Type**

  `application/vnd.oracle.adf.resourceitem+json`

- **Payload**

```
{
  "EmployeeId" : 101,
  "FirstName" : "Neena",
  "LastName" : "Smith",
  "Email" : "NSMITH",
  "JobId" : "AD_VP",
  "DepartmentId" : 90,
  "Salary" : 2000,
  "links" : [ {
    "rel" : "lov",
    "href" : "http://server/demo/rest/11.0/Employees/101/lov/JobsLOV",
    "name" : "JobsLOV",
    "kind" : "collection"
  }, {
    "rel" : "self",
    "href" : "http://server/demo/rest/11.0/Employees/101",
    "name" : "Employees",
    "kind" : "item"
  }, {
    "rel" : "canonical",
    "href" : "http://server/demo/rest/11.0/Employees/101",
    "name" : "Employees",
    "kind" : "item"
  }, {
    "rel" : "enclosure",
    "href" : "http://server/demo/rest/11.0/Employees/101/enclosure/Picture",
        "name" : "Picture",
        "kind" : "other"
      } ]
    }
```

Working with LOV

In ADF, you can retrieve LOV-enabled attributes when you create or retrieve existing resource items using the HTTP methods. The ADF REST runtime supports retrieving LOV-enabled attribute values when the resource item does not exist in the resource collection.
The ADF REST runtime supports the following LOV-enabled attribute use cases.

- Retrieving non-dependent LOV-enabled attribute values for a standalone LOV.

  Refer to this use case when you need to allow users to update an LOV-enabled resource item in the context of a known containing resource item.
- Retrieving nested LOV-enabled attribute values for cascading LOVs.

  Refer to this use case when you need to allow users to update a nested LOV resource item based on a dependent LOV-enabled attribute.

Note that the use case procedure changed starting with ADF REST Framework version 5. In earlier Framework versions, the REST request depended on the use case. However, starting with Framework version 5, all LOV use cases, standalone or cascading, rely on row finders URLs that you enable for static LOV resources.

Retrieving Non-Dependent LOV-Enabled Attribute Values with Framework Versions 1 Through 4

In ADF REST Framework version 4 and earlier, the runtime supports retrieving the values from LOV-enabled attributes in the context of a known containing resource item using a GET method.

In ADF REST Framework versions 4 and earlier, the resource item for an LOV-enabled attribute nests a child LOV resource URL that you can use to obtain the list of values that are based on the context of the already known containing resource item.

> **Note:**
>
> Starting with ADF REST Framework version 5, the use of row finder URLs to populate a LOV resource item is recommended. See Retrieving LOV-Enabled Attribute Values with Framework Version 5 and Later.

When you need to populate cascading LOV attribute lists, in ADF REST Framework versions 4 and earlier, see Retrieving Dependent LOV-Enabled Attribute Values with Framework Versions 1 Through 4.

In Framework version 4 and earlier, to work with LOV-enabled attributes in non-cascading lists:

1. Execute the resource describe and locate the following details about the LOV:

Under the lov description of a resource attribute, locate the childRef property to identify the child resource collection that contains the LOV choices. Note that the resource describe of a LOV child resource does not show the lov description, as the operation to retrieve LOV-enabled attribute values is not supported on LOV child resources.

The lov description contains the view object attribute mapping from the resource item to the LOV child resource collection. The attribute mapping contains one or more source attributes from the child resource whose values will be copied to the resource item when an LOV value is selected. An LOV selection could derive more values in the resource item besides the attribute displaying the LOV. In such cases, the attribute mapping will identify it as derived = true. Finally, the lov description identifies which attributes from the child resource collection could be used for displaying to end users if the results are bound to a user interface.

2. Then, look for the resource item links element and locate the link with a rel of lov and the href that identifies the corresponding LOV child resource (as specified in the resource item childRef property).

   Tip: The resource describe displays the LOV child resource as a template that is completed by supplying a specific resource item name from the resource collection. You may execute a GET to retrieve the resource collection and locate the working link for the LOV child resource that corresponds to the specific resource item.

3. Execute a GET using the LOV link and, optionally, use additional filtering to narrow down the LOV results or to exclude not needed attributes from the payload. You should always fetch the display attributes listed in the lov description. The display attribute is the attribute value that allows the end user to make an LOV value selection from a list of values that they would recognize (such as the list of job titles). The LOV definition ensures that the correct source attribute is used to update the resource item when the LOV selection is applied.

For example, the Employees resource collection describe returns the following:

```
{
  "Resources" : {
    "Employees" : {
      "discrColumnType" : false,
      "attributes" : [ {
        "name" : "EmployeeId",
        "type" : "integer",
        "updatable" : true,
        "mandatory" : true,
        "queryable" : true,
        "precision" : 6
      }, {
        "name" : "FirstName",
        "type" : "string",
        "updatable" : true,
        "mandatory" : false,
        "queryable" : true,
        "precision" : 20
      }, {
        "name" : "LastName",
        "type" : "string",
        "updatable" : true,
        "mandatory" : true,
        "queryable" : true,
        "precision" : 25
      }, {
        "name" : "Email",
        "type" : "string",
        "updatable" : true,
        "mandatory" : true,
        "queryable" : true,
        "precision" : 25
      }, {
        "name" : "JobId",
        "type" : "string",
        "updatable" : true,
        "mandatory" : true,
        "queryable" : true,
        "precision" : 10,
        "controlType" : "choice",
        "maxLength" : "10",
        "lov" : {
          "childRef" : "JobsLOV",
          "attributeMap" : [ {
            "source" : "JobId",
            "target" : "JobId"
          } ],
          "displayAttributes" : [ "JobTitle" ]
        }
     ...

      "item" : {
        "links" : [ {
          "rel" : "lov",
          "href" :
                  "http://server/demo/rest/11.0/Employees/{id}/lov/JobsLOV",
          "name" : "JobsLOV",
          "kind" : "collection"
        }, {
...
```

The following sample fetches the values for the LOV display attribute `JobTitle` from the `JobsLOV` resource collection. The query parameters `onlyData` and `fields` ensure the representation is filtered to contain only data in the response payload, where `President` and `Administration Assistant` are examples of the values of the display attribute `JobTitle`.

Request Made With Framework Version 4 or Earlier

- **URL**

  `http://server/demo/rest/11.0/Employees/101/lov/JobsLOV?onlyData=true&fields=JobTitle`

- **HTTP Method**

  GET

- **Content-Type**

  none

- **Payload**

Response Returned By Framework Version 4 or Earlier

- **HTTP Code**

```
200
```

- **Content-Type**

  ```
  application/vnd.oracle.adf.resourcecollection+json
  ```

- **Payload**

  ```
  {
    "items" : [ {
      "JobTitle" : "President"
    }, {
      "JobTitle" : "Administration Assistant"
    }, {
      "JobTitle" : "Finance Manager"
    }, {
      "JobTitle" : "Accountant"
    }, {
      "JobTitle" : "Accounting Manager"
    }, {
      "JobTitle" : "Public Accountant"
    } ],
    "count" : 6,
    "hasMore" : false,
    "limit" : 25,
    "offset" : 0,
    "links" : [ {
      "rel" : "self",
      "href" : "http://server/demo/rest/11.0/Employees/101/lov/JobsLOV",
      "name" : "JobsLOV",
      "kind" : "collection"
    } ]
  }
  ```

Retrieving Dependent LOV-Enabled Attribute Values with Framework Versions 1 Through 4

In ADF REST Framework version 4 and earlier, the runtime supports retrieving LOV-enabled attribute values using a GET method when it is necessary to populate a resource item without a row context, such as when supporting dependent LOV-enabled attribute lists, also known as cascading LOVs.

In ADF REST Framework versions 1 through 4 the describe provides a static LOV URL to access the dependent list without the context of a known row, where the selection the end user makes in one LOV determines the list to display in the nested LOV. Representing two lists of States and Cities is an example of cascading LOVs, where the list to display for the Cities LOV child attribute is not know until the State LOV attribute selection has been made. In this case, the ADF REST resource developer working in the Model project defines static LOV resources that do not require a row context to enable access to the nested LOV attribute list in the resource describe. For details about creating a static LOV resource in the Model project, see How to Support Create Operations on ADF REST Resources with LOV Attributes.

When you need to populate a non-dependent LOV attribute list (a single LOV) in ADF REST Framework versions 1 through 4, see Retrieving LOV-Enabled Attribute Values for Existing Resource Items.

> **Note:**
>
> Starting with ADF REST Framework version 5, the use of row finder URLs to populate a LOV resource item is recommended. With version 5 enabled, only top-level LOV resources are supported, where the resource describe provides no URL to access nested LOV child resources. See Retrieving LOV-Enabled Attribute Values with Framework Version 5 and Later.

In Framework version 4 and earlier, to work with dependent LOV-enabled attributes:

1. Execute the resource describe and locate the following details about the LOV:

   Under the `lov` description of an attribute, locate the `lovResourcePath` property to identify the LOV resource that contains the LOV choices. Note that the resource describe of a LOV child resource does not show the `lov` description, as the operation to retrieve LOV-enabled attribute values is not supported on LOV child resources.

   The `lov` description contains the attribute mapping from the resource item to the LOV resource collection. The attribute mapping contains one or more source attributes from the LOV resource whose values will be copied to the resource item when an LOV value is selected. An LOV selection could derive more values in the resource item besides the attribute displaying the LOV. In such cases, the attribute mapping will identify it as `derived = true`. Finally, the `lov` description identifies which attributes from the LOV resource collection could be used for displaying to end users if the results are bound to a user interface.

2. Then, look for the resource collection describe `links` element and locate the link with a `rel` of `lov` and the `href` that identifies the LOV resource (as specified in the `lovResourcePath` property).

   Note the `lovResourcePath` is an array that may specify more than one LOV resource. The array supports the use case of dependent LOV-enabled attributes (also called cascading LOVs). For example, the array `"lovResourcePath" : [ "Countries", "States" ]` identifies LOV resources `Countries` and `States`, where the LOV list to display the states is dependent on the LOV selection for countries. To get the values to display for the dependent LOV attribute, you use a child link from the parent LOV resource collection as follows:

   a. Execute a GET using the LOV resource link (where `rel: lov` and `name: Countries`).

   b. The end user selects a resource item from the LOV resource collection (for example, `United States`).

   c. Execute a GET using the child link (`rel: child` and `name: States`) in the selected resource item. This will retrieve a resource collection with the list of values (states) based on the parent LOV resource (countries).

3. Execute a GET using the LOV resource link and, optionally, use additional filtering to narrow down the LOV results or to exclude not needed attributes from the payload. You should always fetch the display attributes listed in the `lov` description. The display attribute is the attribute value that allows the end user to make an LOV value selection from a list of values that they would recognize (such as the list of job titles).

4. With the cached LOV attribute values from the LOV resource, it is then possible to display the LOV display attribute in the client user interface where the end

user is expected to enter values to create a new resource item. When enters the desired values, chooses a value from the displayed LOV for the LOV-enabled attribute, and then clicks Submit, the client can execute the create request. The ADF Business Components view object attribute's LOV definition ensures that the correct source attribute is used to update the resource item when the LOV selection is applied. For details about submitting a POST request with the payload that creates a new resource item, see Creating a Resource Item in Collection.

For example, the `Employees` resource collection describe returns the following:

```
{
  "Resources" : {
    "Employees" : {
      "discrColumnType" : false,
      "attributes" : [ {
        "name" : "EmployeeId",
        "type" : "integer",
        "updatable" : true,
        "mandatory" : true,
        "queryable" : true,
        "precision" : 6
      }, {
        "name" : "FirstName",
        "type" : "string",
        "updatable" : true,
        "mandatory" : false,
        "queryable" : true,
        "precision" : 20
      }, {
        "name" : "LastName",
        "type" : "string",
        "updatable" : true,
        "mandatory" : true,
        "queryable" : true,
        "precision" : 25
      }, {
        "name" : "Email",
        "type" : "string",
        "updatable" : true,
        "mandatory" : true,
        "queryable" : true,
        "precision" : 25
      }, {
        "name" : "JobId",
        "type" : "string",
        "updatable" : true,
        "mandatory" : true,
        "queryable" : true,
        "precision" : 10,
        "controlType" : "choice",
        "maxLength" : "10",
        "lov" : {
          "childRef" : "JobsLOV",
          "attributeMap" : [ {
            "source" : "JobId",
            "target" : "JobId"
          } ],
          "displayAttributes" : [ "JobTitle" ],
          "lovResourcePath" : [ "Jobs" ]
        }
      ...
```

```
        },
        "links" : [ {
          ...
            "rel" : "lov",
            "href" : "http://server/demo/rest/11.0/Jobs",
            "name" : "Jobs",
            "kind" : "collection"
          }, {
...
```

The following sample fetches the values for the LOV display attribute `JobTitle` from the `Jobs` resource collection. The query parameters `onlyData` and `fields` ensure the representation is filtered to contain only data in the response payload, where `President` and `Administration Assistant` are examples of the values of the display attribute `JobTitle`.

Request Made With Framework Version 4 or Earlier

- **URL**

  `http://server/demo/rest/11.0/Jobs?onlyData=true&fields=JobTitle`

- **HTTP Method**

  GET

- **Content-Type**

  none

- **Payload**

Response Returned By Framework Version 4 or Earlier

- **HTTP Code**

  `200`

- **Content-Type**

  `application/vnd.oracle.adf.resourcecollection+json`

- **Payload**

```
{
  "items" : [ {
    "JobTitle" : "President"
  }, {
    "JobTitle" : "Administration Assistant"
  }, {
    "JobTitle" : "Finance Manager"
  }, {
    "JobTitle" : "Accountant"
  }, {
    "JobTitle" : "Accounting Manager"
  }, {
    "JobTitle" : "Public Accountant"
  } ],
  "count" : 6,
  "hasMore" : false,
  "limit" : 25,
  "offset" : 0,
  "links" : [ {
    "rel" : "self",
    "href" : "http://server/demo/rest/11.0/Jobs",
        "name" : "Jobs",
        "kind" : "collection"
      } ]
    }
```

Retrieving LOV-Enabled Attribute Values with Framework Version 5 and Later

In ADF REST Framework version 5 and later, the runtime supports retrieving values from LOV-enabled attributes using a GET method by passing in the containing resource item which may be either already known or based on a selection made at runtime in a dependent LOV attribute list.

The row finder URL supports obtaining the LOV list of a single LOV or of cascading LOVs, where the selection the end user makes in one LOV determines the list to display in the nested LOV. Representing two lists of States and Cities is an example of cascading LOVs, where the list to display for the Cities LOV child attribute is not know until the State LOV attribute selection has been made. In this case, the ADF REST resource developer working in the Model project defines static LOV resources that do not require having a row context to enable access to the nested LOV-enabled attribute list in the resource describe. For details about creating a static LOV resource in the Model project, see How to Support Create Operations on ADF REST Resources with LOV Attributes.

The row finder URL that you obtain from the resource describe and pass on a GET request allows you to submit a bind parameter value to specify the containing resource and give the LOV-enable attribute its row context. In the case of a cascading LOVs, once you have retrieved the dependent LOV attribute values, you can populate the nested LOV list by using a POST request with the retrieved values as the payload.

> **Note:**
>
> Starting with ADF REST Framework version 5, the use of row finder URLs to populate a LOV resource item is recommended. Unlike earlier versions of the framework, the use of row finder URLs in the describe supports both dependent and non-dependent LOVs. For details about working with LOVs with earlier framework versions, see instead Retrieving Non-Dependent LOV-Enabled Attribute Values with Framework Versions 1 Through 4 and Retrieving Dependent LOV-Enabled Attribute Values with Framework Versions 1 Through 4.

In Framework version 5 and later, to work with LOV row finder URLs:

1. Execute the resource describe and locate the following details about the LOV:

   Under the `lov` description of an attribute, locate the `childRefForCreate` property to identify the LOV resource that contains the LOV choices.

   The `lov` description contains the attribute mapping from the resource item to the LOV resource collection. The attribute mapping contains one or more source attributes from the LOV resource whose values will be copied to the resource item when an LOV value is selected. An LOV selection could derive more values in the resource item besides the attribute displaying the LOV. In such cases, the attribute mapping will identify it as `derived = true`. Finally, the `lov` description identifies which attributes from the LOV resource collection could be used for displaying to end users if the results are bound to a user interface.

2. Then, look for the resource item describe `links` element and locate the link with a `rel` of `lov` and the `href` for the parent LOV list.

3. Execute a GET using the LOV resource link and, optionally, use additional filtering to narrow down the LOV results or to exclude not needed attributes from the payload. You should always fetch the display attributes listed in the `lov` description. The display attribute is the attribute value that allows the end user to make an LOV value selection from a list of values that they would recognize (such as the list of states).

4. With the cached LOV attribute values from the LOV resource, it is then possible to display the LOV display attribute in the client user interface where the end user is expected to enter values to create a new resource item. When enters the desired values, chooses a value from the displayed LOV for the LOV-enabled attribute, and then clicks Submit, the client can execute the create request. The ADF Business Components view object attribute's LOV definition ensures that the correct source attribute is used to update the resource item when the LOV selection is applied. For details about submitting a POST request with the payload that creates a new resource item, see Creating a Resource Item in Collection.

5. To get the values to display for a dependent LOV attribute, use a LOV resource link (as specified in the `childRefForCreate` property) with parametrized row finder for the selected resource item. Execute a GET using the LOV resource link (`rel: lov` and `name: CityLOVForCreate`) in the selected resource item. This will retrieve a resource collection with the list of values (cities) based on the parent LOV resource (states).

For example, the `Address` resource collection describe returns the following:

```
{
  "Resources" : {
    "Address" : {
      "discrColumnType" : false,
      "attributes" : [ {
       ...
       }, {
         "name" : "City",
         "type" : "string",
         "updatable" : true,
         "mandatory" : true,
         "queryable" : true,
         "precision" : 32,
         "controlType" : "choice",
         "maxLength" : "32",
         "lov" : {
           "childRef" : "CityLOV",
           "childRefForCreate" : "CityLOVForCreate",
           "attributeMap": [ {
             "source" : "CityName",
             "target" : "City"
           }],
           "displayAttributes" : [ "CityName" ],
         }
       }, {
          "name" : "State",
          "type" : "string",
          "updatable" : true,
          "mandatory" : true,
          "queryable" : true,
          "precision" : 2,
          "controlType" : "choice",
          "maxLength" : "2",
          "lov" : {
            "childRef" : "StateLOV",
            "childRefForCreate" : "StateLOVForCreate",
            "attributeMap": [ {
              "source" : "StateCode",
              "target" : "State"
            }],
            "displayAttributes" : [ "StateCode" ],
          }
        }, {
      ...
      },
      "item" : {
        "links" : [ {
          ...
        }, {
          "rel": "lov",
          "href": "http://server/demo/rest/11.0/Address/{id}/lov/
CityLOV",
          "name": "CityLOV",
          "kind": "collection"},
        }, {
          "rel" : "lov",
          "href" : "http://server/demo/rest/11.0/Cities?
finder=ByStateFinder%3BstateVar%3D{State}",
          "name" : "CityLOVForCreate",
          "kind" : "collection"
        }, {
        }, {
          "rel": "lov",
          "href": "http://server/demo/rest/11.0/Address/{id}/lov/
StateLOV",
          "name": "StateLOV",
          "kind": "collection"},
        }, {
          "rel" : "lov",
          "href" : "http://server/demo/rest/11.0/States",
          "name" : "StateLOVForCreate",
          "kind" : "collection"
        }, {
...
```

The following sample fetches the values for the parent LOV display attribute `StateCode` from the `States` LOV resource collection. The query parameters `onlyData` and `fields` ensure the representation is filtered to contain only data in the response payload, where `CA` and `TX` are examples of the values of the display attribute `StateCode`.

LOV Request Made With Framework Version 5

- **URL**

  `http://server/demo/rest/11.0/States?onlyData=true&fields=StateCode`
- **HTTP Method**

  GET
- **Content-Type**

  none
- **Payload**

LOV Response Returned By Framework Version 5

- **HTTP Code**

  `200`
- **Content-Type**

  `application/vnd.oracle.adf.resourcecollection+json`
- **Payload**

```
{
  "items" : [ {
     "StateCode" : "CA"
  }, {
     "StateCode" : "TX"
  }, {
     "StateCode" : "MS"
  }, {
  ...
  }
 ]
}
```

Where the usage involves a cascading LOV, this sample fetches the values for the dependent LOV with the LOV display attribute `CityName` from the `Cities` LOV resource collection. The row finder takes a parameter for the state name `CA` returned in the client for the parent LOV.

Dependent LOV Request Made With Framework Version 5

- **URL**

  ```
  http://server/demo/rest/11.0/Cities?
  finder=ByStateFinder%3BstateVar%3D{State}"
  ```
- **HTTP Method**

  GET
- **Content-Type**

  none
- **Payload**

Dependent LOV Response Returned By Framework Version 5

- **HTTP Code**

  `200`

- **Content-Type**

  `application/vnd.oracle.adf.resourcecollection+json`

- **Payload**

```
{
  "items": [{
     "Statecode": "CA",
     "Cityname": "Tracy"
  }, {
     "Statecode": "CA",
     "Cityname": "Pleasanton"
  }, {
     "Statecode": "CA",
     "Cityname": "Jackson"
  }, {
     "Statecode": "CA",
     "Cityname": "San Jose"
  }, {
  ...
  }
 ]
}
```

Working with ADF REST Framework Versions

Service clients may need to pass a different payload format to utilize a new feature or enhancement, such as the advanced query capabilities offered in version 2 of the ADF REST runtime framework. A framework version refers to a specific version of the ADF REST framework available starting in a particular Oracle JDeveloper release.

Being able to specify a framework version to process requests, allows clients to opt into those features when they are ready.

The ADF REST runtime supports the following use cases for executing requests according to specific ADF REST framework versions:

- Ability for service clients to specify a framework version that affects the processing of the payload.
- Ability for service clients to indicate the default framework version (as configured by the server) to be used.
- Ability for ADF REST resource developers to declare the default framework version for all URIs under a release version:
  - When a request does not specify a framework version, all payloads for URIs starting with `/context/<release-version>/` will assume the default framework version as declared in the `adf-config.xml` file.
  - When a request specifies a framework version, the specified framework version will be honored.

Service clients may pass the custom header REST-Framework-Version on the REST resource request to specify the framework version to use to execute the request. The ADF REST framework version passed in the version header overrides the default framework declaration defined by the application in the adf-config.xml file by the REST service developer.

When the service client passes no version header in the request, the ADF REST runtime uses the default, as defined in the adf-config.xml file. When a default framework version is not defined and no version header is passed, then the base version (version 1) of the ADF REST framework is assumed.

The framework version declaration is made by ADF REST resource developers in the adf-config.xml file for each application-specific resource version name. For example, the sample following shows resource versions 11.2, 11.1, and 11.0 with the framework versions 3, 2, and 1 declared.

```
<versions>
    <version name="11.2" displayName="11.2" restFrameworkVersion="2"/>
    <version name="11.1" displayName="11.1" lifecycle="deprecated"
restFrameworkVersion="1"/>
    <version name="11.0" displayName="11.0" lifecycle="deprecated"
restFrameworkVersion="1"/>
</versions>
```

In the above example, when the request does not pass a framework version header, resources 11.0 and 11.1 will use the old payload format, while 11.2 will use the new payload format. This ensures, in cases where a framework version is not specified in the request, existing service clients which access resources for a particular release will not be affected when the ADF REST framework introduces a new payload format; yet new service clients which access the latest resources may pick up the new payload format.

For the root resource /context, the default ADF REST framework version for the latest release will be used.

The service client may want to find out the default framework version for a particular release. To support this use case, ADF REST will return the default framework version in the resource version describe, as the following sample shows. Note that service clients may override the default framework version with another framework version identifier by specifying the value in the REST-Framework-Version header. The allowedFrameworkVersions property lists the values of the available framework versions.

```
{
    "items" : [
        {
            "version" : "11.2",
            "isLatest" : true,
            "adf:extension" : {
                    "defaultFrameworkVersion" : "2",
                    "allowedFrameworkVersions" : [ "1","2","3","4","5","6","7" ]
            },
            "links" : [
             ...
```

For details about the ADF REST framework functionality supported in each framework version, see What You May Need to Know About Versioning the ADF REST Runtime Framework.

Retrieving ADF REST Framework Versions for Resource Versions

The ADF REST runtime supports retrieving the version of the ADF REST framework associated with each application-specific release version name that the service end point defines using a GET method.

While the release versions named in the REST application are specific to the application's REST resources, framework versions designate a change in functionality of the ADF REST framework and correspond to a particular JDeveloper release. To ensure the service endpoint exposes the desired level of functionality to service clients, REST resource developers may optionally assign a framework version to each REST resource release version name in the application's `adf-config.xml` file.

> **Note:**
>
> For details about the ADF REST framework functionality supported in each framework version, see What You May Need to Know About Versioning the ADF REST Runtime Framework.

To examine the framework version assigned to the release version names that the application defines:

1. Execute the service end point describe and locate the available release version names in the describe.
2. Locate the `version` property to identify the release version.
3. Examine the associated `defaultFrameworkVersion` property to understand the version of the ADF REST framework that will be used to execute requests for the specific release version:
   - `defaultFrameworkVersion` property specifies the default ADF REST framework version that has been associated with a particular release version, as optionally defined by the ADF REST resource developer in the `adf-config.xml` file. Note that a new ADF REST framework version may introduce new functionality. Thus, associating a specific framework version with each release version ensures that service clients interact with the appropriate level of functionality. See Working with ADF REST Framework Versions.
4. If you want to override the default framework version, examine the `allowedFrameworkVersions` property to understand what versions that you may use:
   - `allowedFrameworkVersions` property identifies the list of ADF REST framework versions that are supported for a particular release version. Service clients may override the default framework version with any value in the list by specifying the value in the `REST-Framework-Version` header.

For example, the describe for a service end point with two release versions returns the following objects, where each release version has been associated with a specific ADF REST framework version:

```
{
    "items" : [
        {
            "version" : "version_identifier_latest",
            "isLatest" : true,
            "adf:extension" : {
                    "defaultFrameworkVersion" : "framework_identifier",
                    "allowedFrameworkVersions" : [ "framework_identifier1", 
"framework_identifier2", ... ]
            },
            "links" : [
                {
                    "rel" : "self",
                    ...

                },
                {
                    "rel" : "canonical",
                    ...

                },
                {
                    "rel" : "predecessor-version",
                    ...

                },
                {
                    "rel" : "describe",
                    ...

                }
            ]
        },
        {
            "version" : "version_identifier_previous",
            "adf:extension" : {
                    "defaultFrameworkVersion" : "framework_identifier",
                    "allowedFrameworkVersions" : [ "framework_identifier1", 
"framework_identifier2", ... ]
            },
            "links" : [
                {
                    "rel" : "self",
                    ...
                },
                {
                    "rel" : "canonical",
                    ...
                },
                {
                    "rel" : "successor-version",
                    ...
                },
                {
                    "rel" : "describe",
                    ...
                }
            ]
        }
    ],
    "links" : [
     ...
        {
            "rel" : "current",
                    ...
            }
        ]
}
```

The following sample retrieves all available release versions defined in the resource catalog of the `demo` application. In the sample, the three release versions are `11.0`, `11.1`, and `11.2`, where `11.2` is the current (or most recent) release version. In the sample, versions `11.0` and `11.1` are explicitly associated with ADF REST framework version `1` and release version `11.2` is associated with ADF REST framework version 2. Note that a framework version refers to a specific version of the ADF REST framework, available starting in a particular Oracle JDeveloper release.

Request

- **URL**

  `http://server/demo/rest`

- **HTTP Method**

  GET

- **Content-Type**

  none

- **Payload**

  none

Response

- **HTTP Code**

  `200`

- **Content-Type**

  `application/vnd.oracle.adf.description+json`

- **Payload**

```
{
    "items" : [
        {
            "version" : "11.2",
            "isLatest" : true,
            "adf:extension" : {
                    "defaultFrameworkVersion" : "3",
                    "allowedFrameworkVersions" : [ "1", "2", "3",
"4" ]
            },
            "links" : [
                {
                    "rel" : "self",
                    "href" : "http://server/demo/rest/11.2",
                    "name" : "self",
                    "kind" : "item"
                },
                {
                    "rel" : "canonical",
                    "href" : "http://server/demo/rest/11.2",
                    "name" : "canonical",
                    "kind" : "item"
                },
                {
                    "rel" : "predecessor-version",
                    "href" : "http://server/demo/rest/11.1",
                    "name" : "predecessor-version",
                    "kind" : "item"
                },
                {
                    "rel" : "describe",
                    "href" : "http://server/demo/rest/11.2/describe",
                    "name" : "describe",
                    "kind" : "describe"
                }
            ]
        },
        {
            "version" : "11.1",
            "adf:extension" : {
                    "defaultFrameworkVersion" : "1",
                    "allowedFrameworkVersions" : [ "1", "2", "3", "4" ]
            },
            "links" : [
                {
                    "rel" : "self",
                    "href" : "http://server/demo/rest/11.1",
                    "name" : "self",
                    "kind" : "item"
                },
                {
                    "rel" : "canonical",
                    "href" : "http://server/demo/rest/11.1",
                    "name" : "canonical",
                    "kind" : "item"
                },
                {
                    "rel" : "predecessor-version",
                    "href" : "http://server/demo/rest/11.0",
                    "name" : "predecessor-version",
                    "kind" : "item"
                },
                {
                    "rel" : "successor-version",
                    "href" : "http://server/demo/rest/11.2",
                    "name" : "successor-version",
                    "kind" : "item"
                },
                {
                    "rel" : "describe",
                    "href" : "http://server/demo/rest/11.1/describe",
                    "name" : "describe",
                    "kind" : "describe"
                }
            ]
        },
        {
            "version" : "11.0",
            "adf:extension" : {
                    "defaultFrameworkVersion" : "1",
                    "allowedFrameworkVersions" : [ "1", "2", "3", "4" ]
            },
            "links" : [
                {
                    "rel" : "self",
                    "href" : "http://server/demo/rest/11.0",
                    "name" : "self",
                    "kind" : "item"
                },
                {
                    "rel" : "canonical",
                    "href" : "http://server/demo/rest/11.0",
                    "name" : "canonical",
                    "kind" : "item"
                },
                {
                    "rel" : "successor-version",
                    "href" : "http://server/demo/rest/11.1",
                    "name" : "successor-version",
                    "kind" : "item"
                },
                {
                    "rel" : "describe",
                    "href" : "http://server/demo/rest/11.0/describe",
                    "name" : "describe",
                    "kind" : "describe"
                }
            ]
        }
    ],
    "links" : [
        {
            "rel" : "self",
            "href" : "http://server/demo/rest",
            "name" : "self",
            "kind" : "collection"
        },
        {
            "rel" : "canonical",
            "href" : "http://server/demo/rest",
            "name" : "canonical",
            "kind" : "collection"
        },
        {
            "rel" : "current",
            "href" : "http://server/demo/rest/11.2",
            "name" : "current",
            "kind" : "item"
        }
    ]
}
```

Executing a Request Using the Header to Specify the Framework Version

The ADF REST runtime supports executing individual requests on the service endpoint using a custom header to affect the processing of the payload with the functionality specific to a particular ADF REST framework version. The framework version specified by the custom header overrides the default framework version declaration that may exist in the client application.

Note:

A framework version refers to a specific version of the ADF REST framework (corresponding to a particular Oracle JDeveloper release). For details about the ADF REST framework functionality supported in each framework version, see What You May Need to Know About Versioning the ADF REST Runtime Framework.

To ensure the service endpoint exposes the desired level of functionality to service clients, REST resource developers may optionally assign a default ADF REST framework version to each REST resource version named in the application’s `adf-config.xml` file. However, the service client may override the default framework version and process individual requests using a specified framework version that gets passed in a custom header. In this case, the ADF REST runtime will ignore the declared, default framework version.

To process a request using a specific ADF REST framework version, the request must pass the custom header `REST-Framework-Version` with the framework version number specified. For example, the following header specifies framework version 2 will be used to process the request that passes this version header.

```
REST-Framework-Version: 2
```

If the custom header is omitted on the request, then the ADF REST runtime uses the application’s default framework version, as defined in the `adf-config.xml` file. When the application does not define a default framework version and the request on the service client omits the version header, then the base version (version 1) of the ADF REST framework is assumed.

Executing Requests Using the Declared Default Framework Version

The ADF REST runtime supports executing all requests on the service endpoint to affect the processing of the payload with functionality specific to the declared, default version of the ADF REST framework.

Note:

A framework version refers to a specific version of the ADF REST framework (corresponding to a particular Oracle JDeveloper release). For details about the ADF REST framework functionality supported in each framework version, see What You May Need to Know About Versioning the ADF REST Runtime Framework.

To ensure the service endpoint exposes the desired level of functionality to service clients, REST resource developers may optionally assign a default ADF REST framework version to each REST resource version named in the application’s `adf-config.xml` file. Thus, when the request is made, the ADF REST runtime will process the request for the particular resource version according to the declaration in the `adf-config.xml` file.

For example, the following URL specifies resource version 11.2, where the application's adf-config.xml file associates this resource version with version 2 of the ADF REST framework. Because version 2 of the ADF REST framework supports rowmatch filter expressions, as a result, the ADF REST runtime will process the request with the appropriate functionality and fetch the departments with names that begin with SA (for example, SALES) or have a Location of BOSTON.

```
    http://server/demo/rest/11.2/Departments?q=DepartmentName like 'SA*' or
Location = 'BOSTON'
```

Note that a service client may override the default framework version and process individual requests using a specified framework version that gets passed in a custom header. In this case, the ADF REST runtime will ignore the application default. If the custom header is omitted on the request, then the ADF REST runtime always uses the application's default framework version, as defined in the adf-config.xml file. When the application does not define a default framework version and the request on the service client omits the version header, then the base version (version 1) of the ADF REST framework is assumed.

Working with Warning and Error Responses

Error responses can be obtained in the form of a JSON payload with exception details when ADF REST framework version 4 or later is enabled. Alternatively, with framework version 3 or earlier, error responses are in the form of a simple message string.

Warning responses can be obtained in the form of a JSON payload with details when ADF REST framework version 6 or later is enabled.

In addition to HTTP status codes and error messages, it is possible to obtain exception details in the response when your request is enabled to use ADF REST framework version 4 and the request is made for either application/vnd.oracle.adf.error+json or application/json media types. With framework version 4, the response will be in the form an exception detail payload which provides the following benefits to the service client:

- If multiple errors occur in a single request, the details of each error are presented in a hierarchical structure.
- An application-specific error code may be present that identifies the ADF exception corresponding to each error.
- An error path may be present that identifies the location of each error in the request payload structure.

Note:

The exception detail may or may not present certain details, such as the application-specific error code and the request payload's error path.

For example, compare the error response for a POST submitted with a payload that contains the following incorrectly formatted date field when framework version 3 (or earlier) is enable and when framework version 4 (or later) is enabled.

```
{    "EmpNum" : 5027,
     "EmpName" : "John",
     "EmpHireDate" : "not a date"
}
```

Standard Error Response, Version 3 and earlier

Without framework version 4, no response payload is generated and instead only a single error message that does not reference the request payload will be returned in the response.

```
"An instance of type oracle.jbo.domain.Date cannot be created from
string not a date. The string value must be in format YYYY-MM-
DDTHH:MI:SS.sss+hh:mm."
```

Exception Payload Error Response, Version 4 and later

With framework version 4 enabled, the following exception detail payload is generated for the response. The payload includes the usual HTTP status code and formats the details of one or more exceptions in an array structure.

```
{    "title" : "Bad Request",
    "status" : "400",
    "o:errorDetails" : [ {
     "detail" : "An instance of type oracle.jbo.domain.Date cannot be
created from string not a date.
               The string value must be in format YYYY-MM-
DDTHH:MI:SS.sss+hh:mm.",
       "o:errorCode" : "26099",
       "o:errorPath" : "/EmpHireDate"
    } ]
 }
```

Exception Payload Warning Response, Version 6 and later

With framework version 6 enabled, a warning detail payload is generated for the response similar to the following. The payload includes the usual HTTP status code and formats the details of the warning in the `@context` section for each resource item.

```
"@context" : {
    ...
    } ],
    "warnings": [ {
      "detail": "Warning from overridden validateEntity method in
DeptImpl : DeptName = ABC or DEF"
    }, {
      "detail": "Attribute set with value 50 for DeptNum in Dept
failed",
      "o:errorCode": "27011",
      "o:errorPath": "/DeptNum"
    }, {
      "detail": "Warning from overridden create method in
DeptViewDefRowImpl"
    }, {
      "detail": "Warning from overridden afterCommit method in
DeptViewImpl"
    } ]
  }
}
```

Understanding the Exception Payload Error Response

The exception detail payload will be generated for an ADF REST error response when the following conditions exist:

1. ADF REST framework version is version 4.
2. Either `application/vnd.oracle.adf.error+json` or `application/json` is an acceptable media type for the response.

The exception detail payload is a JSON object with the following structure as determined by the ADF REST framework:

```
{   "title" : "Message as per HTTP status code",
    "status" : "HTTP error code",
    "o:errorDetails" : [
     ...
        {
            "detail" : "Message of detail error",
            "o:errorCode" : "error code"
            "o:errorPath" : "JSON pointer to the location of the error
in the request payload"
        },
        ...
    ]
}
```

You opt into the exception payload as the error responses by using framework version 4 and making a request for either the `application/vnd.oracle.adf.error+json` media type or `application/json` media type.

Note that within the exception payload, `o:errorDetails` can vary as per the number and the types of errors encountered. Additionally, the error code and error path are not guaranteed to be present in the response payload and should not be relied upon by service clients.

Obtaining the Standard Error Message Response

The ADF REST runtime generates an error message that describes the validation or system error when the request is made with ADF REST framework versions 1 through 3 enabled.

Before version 4 of the ADF REST framework, the error response returns a single error message and HTTP status code. Version 4 and later allows service clients to obtain an error response with a detailed exception payload.

The following sample attempts to create the Departments object with a new department record. However, for this example the request fails because the record for the department instance already exists. The response is an error message because ADF REST framework version 4 (or later) is not enabled.

Request Example Made With Framework Version 3

- **URL**

  `http://server/demo/rest/11.2/Departments`

- **HTTP Method**

  POST

- **Accept Header**

  `application/vnd.oracle.adf.resourceitem+json,application/json`

- **Payload**

  ```
  {
      "DeptNum" : 50,
      "DeptName" : "SALES",
  }
  ```

Response Example From Framework Version 3

- **HTTP Code**

  `400`

- **Error Response**

  ```
  A department with the same name already exists. Please provide a
  different name.
  ```

Obtaining an Exception Payload Error Response

The ADF REST runtime supports obtaining exception details in the response when your request is enabled to use ADF REST framework version 4 and the request is made with an appropriate media type.

Starting with version 4 of the REST API framework, web applications may obtain an error response with a detailed exception payload.

Notice in the exception payload the `o:errorDetails` array provides the error path for where the error occurred in the request object; however, these particular details may not always be available to web applications.

The following sample attempts to create the department object with a new department item. However, for this example the request fails because the item for the department already exists.

Request Example 1 Made With Framework Version 4

- **URL**

  `http://server/demo/rest/11.2/Departments`

- **HTTP Method**

  POST

- **Accept Header**

```
application/vnd.oracle.adf.resourceitem+json,application/json
```

- **Payload**

```
{
    "DeptNum" : 50,
    "DeptName" : "SALES",
}
```

Response Example 1 From Framework Version 4

- **HTTP Code**

```
400
```

- **Content-Type**

```
application/json
```

- **Payload**

```
{
   "title" : "Bad Request",
   "status" : "400",
   "o:errorDetails" : [ {
      "detail" : "A department with the same name already exists. Please 
provide a different name.",
      "o:errorCode" : "Dept_Rule_0"
  } ]
}
```

The following sample attempts to create the department object with a new department item. However, for this example the request fails because the employee names entered exceed the number of characters allowed by the validation rule defined for the `EmpName` field.

Request Example 2 Made With Framework Version 4

- **URL**

```
http://server/demo/rest/11.1/Departments
```

- **HTTP Method**

  POST

- **Accept Header**

```
application/vnd.oracle.adf.resourceitem+json,application/
vnd.oracle.adf.error+json
```

- **Payload**

```
{
   "DeptNum" : 52,
   "DeptName" : "newDept522",
   "Employees" : [ {
        "EmpNum" : 501,
        "EmpName" : "MILLERSxxxxxxxxxxxxxxxxx"
    }, {
        "EmpNum" : 502,
        "EmpName" : "JONESPxxxxxxxxxxxxxxxxx"
      } ]
}
```

Response Example 2 From Framework Version 4

- **HTTP Code**

  `400`

- **Content-Type**

  `application/vnd.oracle.adf.error+json`

- **Payload**

```
{
   "title" : "Bad Request",
   "status" : "400",
   "o:errorDetails" : [ {
      "detail" : "Value MILLERSxxxxxxxxxxxxxxxxx for field EmpName exceeds 
the maximum length allowed.",
      "o:errorCode" : "27040",
      "o:errorPath" : "/Employees/0/EmpName"
   }, {
      "detail" : "Value JONESPxxxxxxxxxxxxxxxxx for field EmpName exceeds 
the maximum length allowed.",
      "o:errorCode" : "27040",
      "o:errorPath" : "/Employees/1/EmpName"
   } ]
}
```

The following sample attempts to perform a batch operation. However, for this example the batch operation fails for the reasons shown in the exception detail payload of the error response.

Request Example 3 Made With Framework Version 4

- **URL**

  `http://server/demo/rest/11.1`

- **HTTP Method**

  POST

- **Accept Header**

  `application/vnd.oracle.adf.batch+json,application/vnd.oracle.adf.error+json`

- **Payload**

```
{
    "parts": [
        {
            "id": "part1",
            "path": "/Employees",
            "operation": "create",
        "payload" : {
        "EmpNum" : 1299,
        "EmpJob" : "CLERK",
        "EmpMgr" : 7566,
        "EmpHireDate" : null,
        "EmpSal" : 245,
        "EmpComm" : 0,
        "EmpDeptNum" : 30
        }
        },
        {
            "id": "part2",
            "path": "/Employees",
            "operation": "create",
            "payload": {
                "EmpNum" : 7589,
        "EmpName" : "SampleEmpxxxxxxxxxxxxxxxxxx",
        "EmpJob" : "CLERK",
        "EmpMgr" : 7566,
        "EmpHireDate" : null,
        "EmpSal" : 245,
        "EmpComm" : 0,
        "EmpDeptNum" : 30
            }
        },
        {
            "id": "part3",
            "path": "/Departments",
            "operation": "create",
            "payload": {
               "DeptNum" : 52,
               "DeptName" : "newDept522",
               "Employees" : [
                   {
                       "EmpNum" : 7588,
                       "EmpName" : "SampleEmpxxxxxxxxxxxxxxxxxx",
                       "EmpJob" : "CLERK",
                       "EmpMgr" : 7566,
                       "EmpHireDate" : null,
                       "EmpSal" : 245,
                       "EmpComm" : 0,
                       "EmpDeptNum" : 30
                   }
               ]
           }
       },
    {
            "id": "part4",
            "path": "/Departments/10/child/Loc",
            "operation": "get"
        },
    {
            "id": "part5",
            "path": "/Departments?invQP=invVal",
            "operation": "get"
        },
    {
            "id": "part6",
            "path": "/Departments/54",
            "operation": "delete"
        },
    {
            "id": "part7",
            "path": "/1.0/Departments/54",
            "operation": "get"
        }
    ]
}
```

Response Example 3 From Framework Version 4

- **HTTP Code**

  400

- **Content-Type**

  `application/vnd.oracle.adf.error+json`

- **Payload**

```
{
    "title" : "Bad Request",
    "status" : "400",
    "o:errorDetails" : [ {
        "detail" : "URL request parameter invQP cannot be used in this
context.",
        "o:errorCode" : "27520"
    }, {
        "detail" : "Attribute EmpName in Emp is required.",
        "o:errorCode" : "27014",
        "o:errorPath" : "/parts/0"
    }, {
        "detail" : "Value SampleEmpxxxxxxxxxxxxxxxxxx for field EmpName
exceeds the maximum length allowed.",
        "o:errorCode" : "27040",
        "o:errorPath" : "/parts/1/payload/EmpName"
    }, {
        "detail" : "Attribute EmpName in Emp is required.",
        "o:errorCode" : "27014",
        "o:errorPath" : "/parts/1"
    }, {
        "detail" : "Value SampleEmpxxxxxxxxxxxxxxxxxx for field EmpName
exceeds the maximum length allowed.",
        "o:errorCode" : "27040",
        "o:errorPath" : "/parts/2/payload/Employees/0/EmpName"
    }, {
        "detail" : "Attribute EmpName in
AM.Dept_empWorksIn_deptToEmpQA_EmpViewDef is required.",
        "o:errorCode" : "27014",
        "o:errorPath" : "/parts/2"
    }, {
        "detail" : "Not Found",
        "o:errorCode" : "11404",
        "o:errorPath" : "/parts/3"
    } ]
}
```

Obtaining Warning Messages in the Payload Response

The ADF REST runtime supports obtaining warning details in the response when your request is enabled to use ADF REST framework version 6 and the request is made with an appropriate media type.

Starting with version 6 of the REST API framework, web applications may obtain a warning response with a response payload.

Notice in the exception payload the `o:errorPaths` element provides the error path for where the error occurred in the request object; however, these particular details depend upon validation rules set on the entity object backing the resource.

The following sample attempts to create the department object with a new department item. However, for this example the request fails because the `DeptName` for the department is specified as not allowed in the validation method of the `Departments` entity object.

Request Example 1 Made With Framework Version 6

- **URL**

  `http://server/demo/rest/12.0/Departments`

- **HTTP Method**

  POST

- **Accept Header**

  `application/vnd.oracle.adf.resourceitem+json,application/json`

- **Payload**

```
{
    "DeptNum" : 50,
    "DeptName" : "ABC",
}
```

Response Example 1 From Framework Version 6

- **HTTP Code**

  `201`

- **Content-Type**

  `application/json`

- **Payload**

```
...
  "@context" : {
    ...
  } ],
  "warnings": [ {
    "detail": "Warning from overridden validateEntity method in DeptImpl : 
DeptName = ABC or DEF"
  }, {
    "detail": "Attribute set with value 50 for DeptNum in Dept failed",
    "o:errorCode": "27011",
    "o:errorPath": "/DeptNum"
  }, {
    "detail": "Warning from overridden create method in DeptViewDefRowImpl"
  }, {
    "detail": "Warning from overridden afterCommit method in DeptViewImpl"
  } ]
 }
}
```

The following sample attempts to create the department object with a new department item. However, for this example the request fails because the employee salary entered does not meet the salary allowed by the validation rule defined for the `EmpSalary` field.

Request Example 2 Made With Framework Version 6

- **URL**

  `http://server/demo/rest/11.1/Departments`

- **HTTP Method**

  POST

- **Accept Header**

```
application/vnd.oracle.adf.resourceitem+json
```

- **Payload**

```
{
   "DeptNum" : 52,
   "DeptName" : "newDept522",
   "Employees" : [ {
        "EmpNum" : 501,
        "EmpName" : "MILLER
        "EmpSalary" : "85"
      }
   ]
}
```

Response Example 2 From Framework Version 6

- **HTTP Code**

```
200
```

- **Content-Type**

```
application/vnd.oracle.adf.error+json
```

- **Payload**

```
...
  "@context" : {
    ...
  } ],
  "warnings": [ {
    "detail": "Warning from Emp entity expressionValidator : Salary value is
< 100.",
    "o:errorCode": "ExcTooLow",
    "o:errorPath": "/Employees/10"
  }, ]
 }
}
```

Advanced Operations

You can use the HTTP methods at ADF REST runtime to perform advanced operations such as fetching a resource collection with a row finder, obtaining a count of resource items in a resource collection, querying a resource with a filter, and retrieving LOV attribute values for an existing resource item.
The ADF REST runtime supports the following advanced use cases:

- Retrieving LOV attribute values in the context of an existing resource item.
- Querying a resource with a partial get using filtering to restrict attributes.
- Fetching the resource collection with a row finder.
- Returning just the data of the resource item or resource collection.
- Returning the estimated count of resource items in a resource collection.
- Overriding the HTTP method to perform an update
- Making batch requests.

Querying With Filtering Attributes (Partial Get)

The ADF REST runtime supports retrieving a subset of fields from resource collections using a GET method.

The payload structure of nested child resource differs depending on the ADF REST framework version that has been registered for the ADF REST service client. For details about the ADF REST framework versions, see What You May Need to Know About Versioning the ADF REST Runtime Framework.

The following samples are based on two different versions of the `Employees` resource and `Departments` resource. The URL samples showing resource `11.0` reflect a response payload structure supported by ADF REST framework versions 1 and 2. While the URL samples showing resource `11.1`, reflect the response payload structure supported in ADF REST framework version 3 (and later). In both framework scenarios, the samples fetch the `Employees` resource as a child of the `Departments` resource.

> **Note:**
>
> SQL SELECT statements executed by the ADF REST resource's backing view object are based on how the view object was created. Only view objects that the ADF Business Components developer creates with declarative SQL mode enabled support optimized SQL SELECT statements formed exclusively by the list of attributes named by the query parameter `fields`. The SELECT statement executed by non-declarative view objects will contain all attributes of the view object definition. To gain this runtime optimization, it is therefore recommended that ADF Business Components developers create view objects for ADF REST resources using only declarative SQL mode.

ADF REST Framework Version 3 (and later)

Starting with version 3 of the ADF REST framework, the ADF REST runtime returns a nested child resource in the response payload as a resource collection, instead of as an array of resource items. This functionality, available in framework version 3 (and later), allows service clients to make a request for additional records after determining how many items were left unfetched in the initial request. The attributes `hasMore` and `count` on the child resource indicate whether more items may be returned from the resource collection. For details about using the pagination attributes from the response payload when you opt into ADF REST framework version 3, see Paging a Resource Collection.

The following sample illustrates functionality for ADF REST framework version 3 (and later). The response payload represents the nested child resource as a resource collection, where the collection object includes the `hasMore` and `count` attributes. A link is provided should it be necessary to query the child resource for additional resource items. In this sample, the response payload shows the `hasMore` attribute is `false`, suggesting that no items remain unfetched on the Employees child resource for either department 10 or department 20.

Request Made With Framework Version 3

- **URL**

```
http://server/demo/rest/11.1/Departments?
fields=DepartmentId;Employees:FirstName&onlyData=true
```

- **HTTP Method**

  GET

- **Content-Type**

  none

- **Payload**

  none

Response Made With Framework Version 3

- **HTTP Code**

  `200`

- **Content-Type**

  `application/vnd.oracle.adf.resourcecollection+json`

- **Payload**

```
{
    "items" : [ {
        "DepartmentId" : 10,
        "Employees" : {
            "items" : [ {
               {
                   "FirstName" : "Jennifer"
                  }
         }  ],
         "count" : 1,
         "hasMore" : false,
         "limit" : 25,
         "offset" : 0,
         "links" : [ {
             "rel" : "self",
             "href" : "http://server/demo/rest/11.1/Departments/10/child/
Employees",
             "name" : "Employees",
             "kind" : "collection"
        }, ]
     }, {
        "DepartmentId" : 20,
            "Employees" : {
                "items" : [ {
                   {
                       "FirstName" : "Michael"
                   },
                   {
                       "FirstName" : "Pat"
                   }
         }  ],
         "count" : 2,
         "hasMore" : false,
         "limit" : 25,
         "offset" : 0,
         "links" : [ {
             "rel" : "self",
             "href" : "http://server/demo/rest/11.1/Departments/20/child/
Employees",
              "name" : "Employees",
              "kind" : "collection"
        }, ]
     }, {
      ...
    ],
    "count" : 25,
    "hasMore" : true,
    "limit" : 25,
    "offset" : 0,
    "links" : [
        {
            "rel" : "self",
            "href" : "http://server/demo/rest/11.1/Departments",
            "name" : "Departments",
            "kind" : "collection"
        }
    ]
```

ADF REST Framework Version 1 or Version 2

Version 1 and version 2 of the ADF REST framework return the nested child resource expanded in the response payload as an array of resource items. If the resource collection being fetched is large, several requests will be required to fetch all items.

The following samples fetch the attribute values for instances of the `Departments` and `Employees` collections. The query parameter `fields` ensures the response payload contains only the specified attributes. Note that a GET request may return no values for any resource in the URL that does not specify an attribute value.

The first request (URL 1) fetches the values for an instance of the `Employees` collection. The query parameter `fields` ensures the response payload contains only the specified attributes: `FirstName`, `LastName`, and `Email`.

The second request (URL 2) fetches the `DepartmentId` values for instances of the `Departments` collection and the `FirstName` value for employees of each department. The query parameter `onlyData` filters the response to hide child links.

The third request (URL 3) fetches the `DepartmentId` values for instances of the `Departments` collection, the `FirstName` value for employees of each department, and the `JobId` history for each employee. The query parameter `onlyData` again filters the response to hide child links.

Request 1 Made With Framework Version 1 or 2

- **URL 1**

  `http://server/demo/rest/11.0/Employees/101?`
  `fields=FirstName,LastName,Email`

- **HTTP Method**

  GET

- **Content-Type**

  none

- **Payload**

  none

Response 1 From Framework Version 1 or 2

- **HTTP Code**

  `200`

- **Content-Type**

  `application/vnd.oracle.adf.resourceitem+json`

- **Payload 1**

```
{
  "FirstName" : "Neena",
  "LastName" : "Smith",
  "Email" : "NSMITH",
  "links" : [ {
    "rel" : "self",
    "href" : "http://server/demo/rest/11.0/Employees/101",
    "name" : "Employees",
    "kind" : "item"
  }, {
    "rel" : "canonical",
    "href" : "http://server/demo/rest/11.0/Employees/101",
    "name" : "Employees",
    "kind" : "item"
  }, {
    "rel" : "lov",
    "href" : "http://server/demo/rest/11.0/Employees/101/lov/JobsLOV",
    "name" : "JobsLOV",
    "kind" : "collection"
  } ]
}
```

Request 2 Made With Framework Version 1 or 2

- **URL 2**

  `http://server/demo/rest/11.0/Departments?fields=DepartmentId;Employees:FirstName&onlyData=true`

- **HTTP Method**

  GET

- **Content-Type**

  none

- **Payload**

  none

Response 2 From Framework Version 1 or 2

- **HTTP Code**

  `200`

- **Content-Type**

  `application/vnd.oracle.adf.resourcecollection+json`

- **Payload 2**

```
{
    "items" : [
        {
            "DepartmentId" : 10,
            "Employees" : [
                {
                    "FirstName" : "Jennifer"
                }
            ]
        },
        {
            "DepartmentId" : 20,
            "Employees" : [
                {
                    "FirstName" : "Michael"
                },
                {
                    "FirstName" : "Pat"
                }
            ]
        },
        {
            "DepartmentId" : 30,
            "Employees" : [
                {
                    "FirstName" : "Den"
                },
                {
                    "FirstName" : "Alexander"
                },
                {
                    "FirstName" : "Shelli"
                },
                {
                    "FirstName" : "Sigal"
                },
                {
                    "FirstName" : "Guy"
                },
                {
                    "FirstName" : "Karen"
                }
            ]
        },
        {
            "DepartmentId" : 40,
            "Employees" : [
                {
                    "FirstName" : "Susan"
                }
            ]
        },
      ...
    ],
    "count" : 25,
    "hasMore" : true,
    "limit" : 25,
    "offset" : 0,
    "links" : [
        {
            "rel" : "self",
            "href" : "http://server/demo/rest/11.0/Departments",
            "name" : "Departments",
                "kind" : "collection"
            }
        ]
```

Request 3 Made With Framework Version 1 or 2

- **URL 3**

  `http://server/demo/rest/11.0/Departments?fields=DepartmentId;Employees:FirstName;Employees.JobHistory:JobId&onlyData=true`

- **HTTP Method**

  GET

- **Content-Type**

  none

- **Payload**

  none

Response 3 From Framework Version 1 or 2

- **HTTP Code**

  `200`

- **Content-Type**

  `application/vnd.oracle.adf.resourcecollection+json`

- **Payload 3**

```
{
    "items" : [

        {
            "DepartmentId" : 10,
            "Employees" : [
                {
                    "FirstName" : "Jennifer",
                    "JobHistory" : [
                        {
                            "JobId" : "AD_ASST"
                        },
                        {
                            "JobId" : "AC_ACCOUNT"
                        }
                    ]
                }
            ]
        },
        {
            "DepartmentId" : 20,
            "Employees" : [
                {
                    "FirstName" : "Michael",
                    "JobHistory" : [
                        {
                            "JobId" : "MK_REP"
                        }
                    ]
                },
                {
                    "FirstName" : "Pat",
                    "JobHistory" : [
                        {
                            "JobId" : "AD_ASST"
                        },
                        {
                            "JobId" : "AC_ACCOUNT"
                        }
                    ]
                }
            ]
        },
        {
            "DepartmentId" : 30,
            "Employees" : [
                {
                    "FirstName" : "Den",
                    "JobHistory" : [
                        {
                            "JobId" : "ST_CLERK"
                        }
                    ]
                },
                {
                    "FirstName" : "Alexander",
                    "JobHistory" : [
                        {
                            "JobId" : "AD_ASST"
                        },
                        {
                            "JobId" : "AC_ACCOUNT"
                        }
                    ]
                },
                {
                    "FirstName" : "Shelli",
                    "JobHistory" : [
                        {
                            "JobId" : "AD_ASST"
                        },
                        {
                            "JobId" : "AC_ACCOUNT"
                        }
                    ]

            ]
          ...
    ],
    "count" : 25,
    "hasMore" : false,
    "limit" : 25,
    "offset" : 0,
    "links" : [
        {
            "rel" : "self",
            "href" : "http://server/demo/rest/11.0/Departments",
            "name" : "Departments",
            "kind" : "collection"
        }
        ]
    }
```

Filtering a Resource Collection with a Row Finder

The ADF REST runtime supports applying a row finder to fetch a resource collection using a GET method. The ADF REST resource developer defines seeded filters called row finders. The ADF REST runtime supports passing parameters into these filters and supports using the seeded filters to reduce (or filter) the collection.

Filtering with a row finder is performed using the **finder** query string to specify one or more row finder parameter values. The **finder** with row finder query string parameter format is:

```
finder=<rowfinderName>;<attr1>=<value1>,<attr2>=<value2>,...
```

Example: `finder=DeptByName;Dname=ACCOUNTING`

The resource collection describe explains the shape of a row finder. To work with the row finder:

1. Execute the resource describe and locate the `finders` attribute in the collection element. The `name` attribute identifies the row finder definition name. Also locate the name of the row finder parameter under `attributes`.
2. Execute a GET with the query parameter `finder` and pass the row finder name and parameters.

For example, the Departments resources describe returns the following:

```
"collection" : {
     "rangeSize" : 25,
     "finders" : [ {
       "name" : "EmpByEmailFinder",
       "title" : "EmployeesByEmailVC",
       "attributes" : [ {
         "name" : "Email",
         "type" : "string",
         "updatable" : true,
         "required" : "Optional",
         "queryable" : false
} ]
```

The following sample fetches the `Departments` collection specified by a row finder `EmpByEmailFinder` where the `Email` attribute value `NSMITH` is passed.

Request

- **URL**

  ```
  http://server/demo/rest/11.1/Employees?
  finder=EmpByNameFinder;Email=NSMITH
  ```

- **HTTP Method**

  GET

- **Content-Type**

  none

- **Payload**

none

Response

- **HTTP Code**

  `200`

- **Content-Type**

  `application/vnd.oracle.adf.resourceitem+json`

- **Payload**

```
{
  "items" : [ {
    "EmployeeId" : 101,
    "FirstName" : "Neena",
    "LastName" : "Smith",
    "Email" : "NSMITH",
    "JobId" : "AD_VP",
    "DepartmentId" : 90,
    "Salary" : 2000,
    "links" : [ {
      "rel" : "self",
      "href" : "http://server/demo/rest/11.1/Employees/101",
      "name" : "Employees",
      "kind" : "item"
    }, {
      "rel" : "canonical",
      "href" : "http://server/demo/rest/11.1/Employees/101",
      "name" : "Employees",
      "kind" : "item"
    }, {
      "rel" : "lov",
      "href" : "http://server/demo/rest/11.1/Employees/101/lov/JobsLOV",
      "name" : "JobsLOV",
      "kind" : "collection"
    } ]
  } ],
  "count" : 1,
  "hasMore" : false,
  "limit" : 25,
  "offset" : 0,
  "links" : [ {
    "rel" : "self",
    "href" : "http://server/demo/rest/11.1/Employees",
    "name" : "Employees",
    "kind" : "collection"
  } ]
}
```

Returning Only Resource Data in a Payload

The ADF REST runtime supports retrieving only the data of resource items using a GET method of a resource collection or a resource item.

The following sample fetches the values of the `Employees` collection attributes. The query parameter `onlyData` ensures the representation is filtered to contain only data in the response payload and no links.

Request

- **URL**

  `http://server/demo/rest/11.1/Employees?onlyData=true`

- **HTTP Method**

  GET

- **Content-Type**

  none

- **Payload**

  none

Response

- **HTTP Code**

  `200`

- **Content-Type**

  `application/vnd.oracle.adf.resourceitem+json`

- **Payload**

```
{
  "items" : [ {
    "EmployeeId" : 101,
    "FirstName" : "Neena",
    "LastName" : "Smith",
    "Email" : "NSMITH",
    "JobId" : "AD_VP",
    "DepartmentId" : 90,
    "Salary" : 2000
  }, {
    "EmployeeId" : 102,
    "FirstName" : "Lex",
    "LastName" : "De Haan",
    "Email" : "LDEHAAN",
    "JobId" : "AD_VP",
    "DepartmentId" : 90,
    "Salary" : 3000
  }, {
    "EmployeeId" : 103,
    "FirstName" : "Alexander",
    "LastName" : "Hunold",
    "Email" : "AHUNOLD",
    "JobId" : "IT_PROG",
    "DepartmentId" : 60,
    "Salary" : 4000
  }, {
    "EmployeeId" : 104,
    "FirstName" : "Bruce",
    "LastName" : "Ernst",
    "Email" : "BERNST",
    "JobId" : "IT_PROG",
    "DepartmentId" : 60,
    "Salary" : 5000
  }, {
    "EmployeeId" : 105,
    "FirstName" : "David",
    "LastName" : "Austin",
      "Email" : "DAUSTIN",
      "JobId" : "IT_PROG",
      "DepartmentId" : 60,
      "Salary" : 6000
    } ],
    "count" : 5,
    "hasMore" : true,
    "limit" : 25,
    "offset" : 0,
    "links" : [ {
      "rel" : "self",
      "href" : "http://server/demo/rest/11.1/Employees",
      "name" : "Employees",
      "kind" : "collection"
    } ]
  }
```

Returning the Estimated Count of Resource Items

The ADF REST runtime supports retrieving the estimated item count in the resource collection.

The following sample estimates the total records and queries the first two items in the `Employee` collection. The query parameter `totalResults` ensures the response payload contains the `totalResults` attribute.

Request

- **URL**

  `http://server/demo/rest/11.1/Employees?totalResults=true&limit=2`

- **HTTP Method**

  GET

- **Content-Type**

  none

- **Payload**

  none

Response

- **HTTP Code**

  `200`

- **Content-Type**

  `application/vnd.oracle.adf.resourceitem+json`

- **Payload**

```
{
  "items" : [ {
    "EmployeeId" : 101,
    "FirstName" : "Neena",
    "LastName" : "Smith",
    "Email" : "NSMITH",
    "JobId" : "AD_VP",
    "DepartmentId" : 90,
    "Salary" : 2000,
    "links" : [ {
      "rel" : "self",
      "href" : "http://server/demo/rest/11.1/Employees/NSMITH",
      "name" : "Employees",
      "kind" : "item"
    }, {
      "rel" : "canonical",
      "href" : "http://server/demo/rest/11.1/Employees/NSMITH",
      "name" : "Employees",
      "kind" : "item"
    }, {
      "rel" : "lov",
      "href" : "http://server/demo/rest/11.1/Employees/NSMITH/lov/JobsLOV",
      "name" : "JobsLOV",
      "kind" : "collection"
    } ]
  }, {
    "EmployeeId" : 102,
    "FirstName" : "Lex",
    "LastName" : "De Haan",
    "Email" : "LDEHAAN",
    "JobId" : "AD_VP",
    "DepartmentId" : 90,
    "Salary" : 3000,
    "links" : [ {
      "rel" : "self",
      "href" : "http://server/demo/rest/11.1/Employees/LDEHAAN",
      "name" : "Employees",
      "kind" : "item"
    }, {
      "rel" : "canonical",
      "href" : "http://server/demo/rest/11.1/Employees/LDEHAAN",
      "name" : "Employees",
      "kind" : "item"
    }, {
      "rel" : "lov",
      "href" : "http://server/demo/rest/11.1/Employees/LDEHAAN/lov/JobsLOV",
      "name" : "JobsLOV",
      "kind" : "collection"
    } ]
  } ],
  "totalResults" : 5,
  "count" : 2,
  "hasMore" : true,
  "limit" : 2,
  "offset" : 0,
  "links" : [ {
    "rel" : "self",
    "href" : "http://server/demo/rest/11.1/Employees",
    "name" : "Employees",
    "kind" : "collection"
  } ]
}
```

Overriding the HTTP Method and Performing an Update

Some HTTP servers or clients do not support exposing all the methods in the HTTP specification. The ADF REST runtime allows you to use the popular POST method to carry out operations for other methods.

The following sample uses the POST method to update a `Departments` resource item by overriding the PATCH method.

Request

- **URL**

  `http://server/demo/rest/11.1/Departments/15`

- **HTTP Method**

  POST

- **X-HTTP-Method-Override**

  PATCH

- **Content-Type**

  `application/vnd.oracle.adf.resourceitem+json`

- **Payload**

```
{
  "DepartmentName": "UpdatedDept",
}
```

Response

- **HTTP Code**

  `200`

- **Content-Type**

  `application/vnd.oracle.adf.resourceitem+json`

- **Payload**

```
{
  "DepartmentId" : 15,
  "DepartmentName" : "UpdatedDept",
  "links" : [ {
    "rel" : "self",
    "href" : "http://server/demo/rest/11.1/Departments/15",
    "name" : "Departments",
    "kind" : "item"
    }
  }, {
    "rel" : "canonical",
    "href" : "http://server/demo/rest/11.1/Departments/15",
    "name" : "Departments",
    "kind" : "item"
  }, {
    "rel" : "child",
    "href" : "http://server/demo/rest/11.1/Departments/15/child/Employees",
    "name" : "Employees",
    "kind" : "collection"
  } ]
}
```

Making Batch Requests

The ADF REST runtime supports executing multiple operations in a single roundtrip using a batch request. The data is committed at the end of the request. However, if one request part in a batch request fails, then all changes are rolled back and an error response is returned.

A batch request can consist of a combination of create, update, delete, upsert, and get requests. The path parameter and the payload needs to be the same as what you use to invoke the request directly. The get method supports the same URL parameters in the batch request as a separate HTTP request.

The following sample illustrates a successful batch operation that executes operations in four parts: 1) update employee 101, 2) update employee 102, 3) update employee 103, 4) query employee 101.

Request

- **URL**

  `http://server/demo/rest/11.1`

- **HTTP Method**

  POST

- **Content-Type**

  `application/vnd.oracle.adf.batch+json`

- **Payload**

```
{
    "parts": [{
        "id": "part1",
        "path": "/Employees/101",
        "operation": "update",
        "payload": {
            "Salary": 10000
        }
    }, {
        "id": "part2",
        "path": "/Employees/102",
        "operation": "update",
        "payload": {
            "Salary": 10000
        }
    }, {
        "id": "part3",
        "path": "/Employees/103",
        "operation": "update",
        "payload": {
            "Salary": 10000
        }
    }, {
        "id": "part4",
        "path": "/Employees?q=EmployeeId%3D101",
        "operation": "get"
    }]
}
```

Response

- **HTTP Code**

  `200`

- **Content-Type**

```
application/vnd.oracle.adf.batch+json
```

- **Payload**

```
{
    "parts": [{
        "id": "part1",
        "path": "http://server/demo/rest/11.1/Employees/101",
        "operation": "update",
        "payload": {
            "EmployeeId": 101,
            "FirstName": "Neena",
            "LastName": "Smith",
            "Email": "NSMITH",
            "JobId": "AD_VP",
            "DepartmentId": 90,
            "Salary": 10000,
            "links": [{
                "rel": "self",
                "href": "http://server/demo/rest/11.1/Employees/101",
                "name": "Employees",
                "kind": "item"
            }, {
                "rel": "canonical",
                "href": "http://server/demo/rest/11.1/Employees/101",
                "name": "Employees",
                "kind": "item"
            }, {
                "rel": "lov",
                "href": "http://server/demo/rest/11.1/Employees/101/lov/
JobsLOV",
                "name": "JobsLOV",
                "kind": "collection"
            }]
        }
    }, {
        "id": "part2",
        "path": "http://server/demo/rest/11.1/Employees/102",
        "operation": "update",
        "payload": {
            "EmployeeId": 102,
            "FirstName": "Lex",
            "LastName": "De Haan",
            "Email": "LDEHAAN",
            "JobId": "AD_VP",
            "DepartmentId": 90,
            "Salary": 10000,
            "links": [{
                "rel": "self",
                "href": "http://server/demo/rest/11.1/Employees/102",
                "name": "Employees",
                "kind": "item"
            }, {
                "rel": "canonical",
                "href": "http://server/demo/rest/11.1/Employees/102",
                "name": "Employees",
                "kind": "item"
            }, {
                "rel": "lov",
                "href": "http://server/demo/rest/11.1/Employees/102/lov/
JobsLOV",
                "name": "JobsLOV",
                "kind": "collection"
            }]
        }
    }, {
        "id": "part3",
        "path": "http://server/demo/rest/11.1/Employees/103",
        "operation": "update",
        "payload": {
            "EmployeeId": 103,
            "FirstName": "Alexander",
            "LastName": "Hunold",
            "Email": "AHUNOLD",
            "JobId": "IT_PROG",
            "DepartmentId": 60,
            "Salary": 10000,
            "links": [{
                "rel": "self",
                "href": "http://server/demo/rest/11.1/Employees/103",
                "name": "Employees",
                "kind": "item"
            }, {
                "rel": "canonical",
                "href": "http://server/demo/rest/11.1/Employees/103",
                "name": "Employees",
                "kind": "item"
            }, {
                "rel": "lov",
                "href": "http://server/demo/rest/11.1/Employees/103/lov/
JobsLOV",
                "name": "JobsLOV",
                "kind": "collection"
            }]
        }
    }, {
        "id": "part4",
        "path": "http://server/demo/rest/11.1/Employees",
        "operation": "get",
        "payload": {
            "EmployeeId": 101,
            "FirstName": "Neena",
            "LastName": "Smith",
            "Email": "NSMITH",
            "JobId": "AD_VP",
            "DepartmentId": 90,
            "Salary": 10000,
            "links": [{
                "rel": "self",
                "href": "http://server/demo/rest/11.1/Employees/101",
                "name": "Employees",
                "kind": "item"
            }, {
                "rel": "canonical",
                "href": "http://server/demo/rest/11.1/Employees/101",
                "name": "Employees",
                "kind": "item"
            }, {
                "rel": "lov",
                "href": "http://server/demo/rest/11.1/Employees/101/lov/
JobsLOV",
                "name": "JobsLOV",
                "kind": "collection"
                }]
            }
        }]
    }
```

ADF REST Framework Reference

ADF REST framework supports encoding of HTTP payloads, different media types, HTTP headers, and methods, so that the communication between the server and client is seamless and secure.
The ADF REST framework supports HTTP methods, HTTP headers, request URL parameters, media types, and other concepts to enable making REST API calls on resources.

ADF REST Describe links Object Structure

`links` is a JSON object where the value is always a URL link and the link name is defined according to the `rel` of the link. The `links` object is generated for each resource collection, item, and for the resource itself.

Note that URL links in the resource describe will be generated using a template placeholder value (`{id}`) when there is not enough information to determine all parts of the URL. For example, the following child link provides a URL with the placeholder for the value of the specific Department resource:

```
"item" : {
     "links" : [ {
        "rel" : "child",
        "href" : "http://server/demo/rest/11.0/Departments/{id}/child/Employees",
        "name" : "Employees",
        "kind" : "collection",
        "cardinality" : {
          "value" : "1 to *",
          "sourceAttributes" : "DepartmentId",
          "destinationAttributes" : "DepartmentId"
        }
```

rel Attribute Values

The `rel` attribute defines the type of link relationship between the current resource and the resource which the link points to. Relationships may be specified by any of the values shown in Table 22-3.

Table 22-3 Link Relationship in ADF REST Resource Describe

Link Relationship	Description
`self`	Always generated for a resource. The `href` points to the resource itself or to the resource describe. In the `links` object, the link name is `self` for this `rel`.
`canonical`	Always generated. The `href` points to the canonical resource or to the canonical resource describe. If a canonical resource is not defined, it will have the same `href` that `self` has. In the `links` object, the link name is `canonical` for this `rel`.

Table 22-3 (Cont.) Link Relationship in ADF REST Resource Describe

Link Relationship	Description
`parent`	Always generated for a nested resource. The `href` points to the `self` link of the parent resource. In the `links` object, the link name is `parent` for this `rel`.
`child`	Generated when the resource has nested children. The `href` points to the nested collection. In the `links` object, the link name is the accessor name for this `rel`.
`lov`	Generated on a resource item for LOV-enabled (list of values) attributes. There are two types of `href`: 1.) on the `links` object of the resource item, the `href` identifies an LOV child resource (name specified in the `childRef` property of the resource item), and 2.) on the `links` object of the resource describe, the `href` identifies a LOV resource (name specified in the `lovResourcePath` property on the resource item). The first type is defined in the model project by an LOV view accessor and an LOV-enabled attribute backing the resource item. The second type is defined in the model project by an LOV resource on the LOV view accessor and the resource is associated with the LOV-enabled attribute backing the resource item.
`enclosure`	Generated for BLOB and CLOB attributes by default. This relationship indicates that the link points to a resource that cannot be represented with the supported payload types. An image, that cannot be represented in JSON, is an example of this relationship. Use the `enclosure` link (can be obtained from resource item `describe`, `get`, or `post/patch` response) to read/update/delete content.
`external`	Generated for a resource that exists outside of the framework domain.
`current`	Generated in the resource version describe when multiple resource version identifiers exist. The `href` points to the most recent version identifier, as defined by the `adf-config.xml` version definition.
`predecessor-version`	Generated in the resource version describe when multiple resource version identifiers exist. The `href` points to the previous version identifier, as defined by the `adf-config.xml` version definition.
`successor-version`	Generated in the resource version describe when multiple resource version identifiers exist. The `href` points to the next most recent version identifier, as defined by the `adf-config.xml` version definition.
`describe`	Generated in the resource version describe. The `href` points to the resource catalog describe for all resources of the same version.

href Attribute Value

The `href` attribute defines the URL to the linked resource or resource describe.

cardinality Attribute Values

The `cardinality` attribute is an optional attribute that defines the cardinality between the source resource and the destination resource. This attribute will be available only when the `rel` attribute value is `child` and the resource type is `vnd.oracle.adf.description+json`. This cardinality attribute has the following attributes.

- `value`: The value of the cardinality. Example: "`1 to *`"
- `sourceAttributes`: The attribute in the source resource used to link to the destination resource.
- `destinationAttributes`: The attribute in the destination resource used to link to the source resource.

ADF REST API Framework Versions

A framework version refers to a specific version of the ADF REST framework available starting in a particular Oracle JDeveloper release.

The ADF REST runtime supports clients to specify a framework version that affects the processing of the payload or indicate the default framework version (as configured by the server) to be used. When you specify a framework version to process requests, it allows clients to opt into those features when they are ready. For example, to support both `rowmatch` expressions (offered in framework version 2 and later) and the query-by-example syntax (version 1 only), you would need to associate framework version 2 (or later) with a new release version identifier that you define in the web application. For example, in the URL samples given below, you can preserve the original functionality for release version 11.0 and expose the new functionality for version 11.1. This assumes that web application has declared framework version 2 the default on release version 11.1. Alternatively, if your web application will no longer require the functionality of your current framework version, you may associate the new framework version with your existing release version identifier. Therefore, you are not required to increment the release version to make use of a new framework version. For details about the ADF REST framework functionality supported in each framework version, see What You May Need to Know About Versioning the ADF REST Framework.

When a request does not specify a framework version, all payloads for URIs starting with `/context/<release-version>/` will assume the default framework version as declared in the `adf-config.xml` file.

> **Note:**
>
> Each ADF REST framework version after version 1 introduces functionality that the previous framework versions does not support. Thus, when you choose to opt into a later framework version, the REST API of your application may introduce backward incompatible changes on the service client consuming the REST API. In the table below, see the Does Not Support column for backward compatibility issues.

The following table explains the changes for each framework version.

Table 22-4 ADF REST API Framework Versions

REST API Framework Version	Supports	Does Not Support	Examples
1 - Default version. Use to process requests for web applications when no other version is specified	Supports query-by-example resource query syntax Filtering resource collections using the q query parameter is limited to a query-by-example.		`GET /rest/19.0/Departments?q=Dname SA*;Loc BOSTON`
2 - You must specify the version for the request. Only then the REST API support the use of expanded expression syntax to process the request.	Supports more advanced query syntax for making REST API calls. Interprets `q` query parameter value differently than Framework version 1. Supports filtering resource collections using `rowmatch` query expressions.	Query-by-example resource query syntax is not compatible. Introduces a backward incompatible change to web application that rely on Framework version 1.	See What You May Need to Know About the Advanced Query Syntax in ADF REST Framework Version 2
3 - The payload structure represents nested child resource as a resource collection, instead of an array of items as in version 1 and 2	Supports retrieving nested child resources with payload attributes that may be used by the web application to determine whether more resource items would be returned in a subsequent REST API request. Supports pagination of nested child resource that would otherwise require more than one request to fetch. Exposes functionality that allows `GET` operations to use the `?expand` and `?fields` query parameter to return a nested child resource as a resource collection with the `hasMore` attribute	Introduces a backward incompatible change to web application that rely on Framework version 1 or 2.	See Querying With Filtering Attributes (Partial Get). Also see Obtaining the Standard Error Message Response for error response.

Table 22-4 (Cont.) ADF REST API Framework Versions

REST API Framework Version	Supports	Does Not Support	Examples
4 – Possible to obtain exception details in the response when your request is enabled to use REST API framework version 4 and the request is made for either `application/vnd.oracle.adf.error+json` or `application/json` media types.	Supports the response in the form an exception detail payload that provides the following benefits to the web application: • Presents the details of each error in a hierarchical structure if multiple errors occur in a single request. • Identifies the exception corresponding to each error by including an application specific error code. • Presents an error path that identifies the location of each error in the request payload structure.	The exception detail may or may not present certain details, such as the application-specific error code and the request payload's error path.	See Obtaining an Exception Payload Error Response.
5 - Possible to share the LOVs that may not depend on the row context to filter out records. Also possible to share the LOVs with multiple resources. LOV resources will be represented by its own top-level resource and the row context LOV URLs are no longer returned in the payload and in the describe.	Removes the dependency of LOVs to be nested inside a resource parameters in the LOV resource URL. Enhanced support for request and response parameters beyond the current supported types of string, number, date, and boolean.	Nested levels of LOV resource URL.	`GET /rest/v1/Cities?finder=ByStateFinder;stateName={State}",` See What You May Need to Know About Configuring LOV Resources for Row Finders. The supported Java types include `java.util.List` and `java.util.Map`.

Table 22-4 (Cont.) ADF REST API Framework Versions

REST API Framework Version	Supports	Does Not Support	Examples

6 - Supports differentiation between the resource fields and the item context information like links and headers. Also displays warnings in the response payload in the context section for create/upsert and update actions.	Non-attribute fields like links and headers appear within @context field in resource item response object
	links field in @context will no longer have the properties field.
	headers -> ETag has the changeIndicator value.
	key field under @context contains the unique identifier of the specific resource item as a string.
	warnings field under @context contains the validation error details when a validation rule has been enabled on a backing entity object attribute.
			...
			"@context" : {
			"key" : "AB8765BCD",
			"headers" : {
			"ETag" : "ACED..."
			...
			},
			"links": [{
			"rel": "self",
			"href":
			...
			}]
			"warnings": [{
			"detail": "Warning from overridden validateEntity method in DeptImpl : DeptName = ABC"
			}, {
			"detail": "Attribute set with value 92 for DeptNum in Dept failed",
			"o:errorCode": "27011",
			"o:errorPath": "/DeptNum"
			}, {
			"detail": "Warning from overridden afterCommit method in DeptViewImpl"
			}, {
			"detail": "Warning from overridden afterCommit method in DeptViewImpl"
			}]
			}
			}

7 - Provides an option to remove row-level LOV resource descriptions in describe and from the resource item payload links section.	Supports the use of top-level LOV resource links	Row context LOVs in the describe and payloads	

ADF REST Payload Compression Support

The HTTP payloads that are exchanged between the server and the client can be encoded. This feature is enabled when the client specifies an `Accept-Encoding: gzip` header or `Content-Encoding` header in the service request.

To disable encoding support for resources, you may set the custom property `adf.rest.enablecompression` to `false`. In JDeveloper, the property is set in the ADF configuration file (`adf-config.xml`) of the Fusion web application.

Table 22-5 describes the content-encoding tokens supported by the ADF REST framework.

Table 22-5 Supported Content-Encoding Tokens

Content-Encoding	Description
`identity`	Does not compress the payload. The behavior is the same as when the encoding is omitted.
`x-gzip` and `gzip`	Compresses the payload using the format produced by the file compression program `gzip` (GNU zip), as described in RFC 1952 [25]. This format is a `Lempel-Ziv` coding (LZ77) with a 32 bit CRC.
`deflate`	Compresses the payload with a combination of the `zlib` format defined in FRC 1950 [31] and the `deflate` compression mechanism described in RFC 1951 [29].

ADF REST Media Types

Media types, also called MIME types or content types, define the allowed resource structure of the payload exchanged between the client and server. All ADF REST media types are based on JSON. Resources accessed in client applications fall under the `application` type and `json` subtype.

The service client invoking the REST API will interact with the RESTful web service using one of the media types listed in Table 22-6. The types are defined such that the media type does not vary with the view object definition backing the resource. Note that the value of the accept header depends on the context of the invocation. Links to the JSON token structure of the ADF REST framework media types are provided in the following table.

> **Note:**
>
> As an alternative to specifying the supported media types, the service client request accept header can specify `application/json` when a superset of all supported media types may be accepted in the response.

Table 22-6 Media Types Supported by the ADF REST Framework

Media Type	Invocation Context	Description
`application/vnd.oracle.adf.resourcecollection+json`	GET method	Represents the format for all resource collections returned by the ADF REST runtime. All attributes are automatically generated by the framework. Only the content of the `items` attribute is based on the resource definition. For an example, see Describing a Resource Collection.

Table 22-6 (Cont.) Media Types Supported by the ADF REST Framework

Media Type	Invocation Context	Description
`application/vnd.oracle.adf.resourceitem+json`	GET method POST method PATCH method	Represents the format for all resource items returned by the ADF REST runtime. Also represents the format for a resource item in a POST or PATCH request payload. Only the attribute `links` is automatically generated by the framework. All the other attributes are based on the resource definition. For an example, see Describing a Resource Item.
`application/vnd.oracle.adf.actionresult+json`	POST method	Describes the result of an action execution.
`application/vnd.oracle.adf.description+json`	GET method	Describes the resource and its elements. For an example, see Describing All Available Resources. .
`application/vnd.oracle.adf.batch+json`	POST method	Describes a set of operations to be performed, where the operation consists of a set of parts and each part represents a request. The batch request is executed in one single transaction. For an example, see Making Batch Requests.
`application/vnd.oracle.adf.version+json`	GET method	Describes the result of a request to get all versions of a resource. For an example, see Retrieving All Available Version Release Names.
`application/vnd.oracle.adf.error+json`	any	Describes the exception payload error response for a request made with an error. To use this media type and obtain the exception details in an error response payload, the request must be made with ADF REST framework version 4 (or later) enabled. For an example, see Obtaining an Exception Payload Error Response.

ADF REST Data Types

ADF REST data types are mapped by the ADF REST framework between ADF REST resource items and their backing ADF Business Components entity object attributes. At runtime, the framework exposes the data type of fetched ADF REST resource items as the describe attribute `type`.

Table 22-7 shows the relationship between the ADF Business Components data types supported on entity object attributes backing the view object of ADF REST resources and the corresponding ADF REST data types that the ADF REST framework defines. In general, the framework defines the data type of an ADF REST resource item based on the SQL type of the attribute backing the resource item, with these two exceptions:

- When the backing attribute is defined as a boolean type map, then the ADF REST type will always be a boolean.
- When the backing attribute’s Java type is blob or clob, then the ADF REST type will be an attachment.

Table 22-7 Data Types Supported by the ADF REST Framework

ADF Backing Attribute	ADF REST Data Type

Where an attribute is configured with one of the following type maps:
`oracle.jbo.valuemaps.Boolean10PropertySet`
`oracle.jbo.valuemaps.BooleanTFPropertySet`
`oracle.jbo.valuemaps.BooleanYNPropertySet`
	`boolean`

Java Class: `oracle.jbo.domain.ClobDomain` or `oracle.jbo.domain.BlobDomain`	`attachment`
Java Class: `java.util.List` supported starting in ADF REST framework version 5	`array`
Java Class: `java.util.Map` supported starting in ADF REST framework version 5	`object`
SQL type: `Char`	`string`
SQL type: `Number`	`integer`
SQL type: `Number` (precision = *, scale > 0)	`number`
SQL type: `Number` (precision < 10, scale = 0)	`integer`
SQL type: `Number` (precision >= 10, scale = 0)	`integer`

SQL type: `Date`	`date` if Java type is `java.sql.Date`
	`time` if Java type is `java.sql.Time`
	`datetime` if Java type is other (not `java.sql.Date` and not `java.sql.Time`)
	The REST service accepts the YYYY-`MM-DDThh:mm:ss.s+hh:mm` or YYYY-`MM-DDThh:mm:ss.s-hh:mm` format. If you specify the time without the ":" in the second part, `s+hhmm` or `s-hhmm`, it fails.

ADF REST HTTP Codes

The ADF REST framework supports the HTTP codes listed in the following table. The specific code that is returned depends on the HTTP method invoked on the web service.

Table 22-8 HTTP Codes Supported by the ADF REST Framework

HTTP Code	Description
`200 OK`	Request successfully executed and the response has content.
`201 Created`	Resource successfully created. The response contains the created resource.
`204 No Content`	Request successfully executed and the response doesn't have content.
`304 Not Modified`	According to the provided ETag, the resource was not modified.
`400 Bad Request`	The request could not be understood by the server due to malformed syntax.

Table 22-8 (Cont.) HTTP Codes Supported by the ADF REST Framework

HTTP Code	Description
401 Unauthorized	The server is refusing to service the request because the resource of the request is secured and authentication has not yet been provided.
404 Not Found	The requested resource was not found.
406 Not Acceptable	The resource identified by the request is only capable of generating response entities which have content characteristics not acceptable according to the accept headers sent in the request.
412 Precondition failed	The resource state in the server side doesn't match the provided ETag.
415 Unsupported Media Type	The server is refusing to service the request because the entity of the request is in a format not supported by the requested resource for the requested method.
500 Internal Server Error	The server encountered an unexpected condition which prevented it from fulfilling the request.

ADF REST HTTP Headers Support

The ADF REST framework supports the HTTP headers listed in the following table.

Table 22-9 HTTP Headers Supported by the ADF REST Framework

HTTP Header Name	Description
Content-Type	Use to specify the content-type of the request/response payload. The ADF REST runtime is able to interpret (request/response) the media types, as described in ADF REST Media Types.
Content-Encoding	Use to specify the content-encoding of the request/response payload. The ADF REST runtime is able to parse a compressed request that uses the encodings, as described in ADF REST Payload Compression Support.
Accept	Use to specify the expected content-type of the response payload. The ADF REST runtime is able to interpret (request/response) the media types, as described in ADF REST Media Types.
Accept-Encoding	Use to specify the list of acceptable encoded responses. The ADF REST framework is able to generate a response using the encodings described in ADF REST Payload Compression Support.
REST-Framework-Version	Use to specify the version of the ADF REST framework to use at runtime to process the request. The ADF REST framework version passed in the version header overrides the default framework declaration defined by the application in the adf-config.xml file, as described in Working with ADF REST Framework Versions.
Location	Use to identify the URI of a newly created resource. The ADF REST framework includes the Location header in the response of a POST to create a new resource. For an example, see Creating a Resource Item in Collection.

Table 22-9 (Cont.) HTTP Headers Supported by the ADF REST Framework

HTTP Header Name	Description
`ETag`	Use to compare the state of the resource in a request with the state of resource on the server. The ADF REST framework supports the `ETag` generation for resources backed by an ADF Business Components entity object that has been configured to use an change-indicator attribute. See Checking for Data Consistency.
`If-Match`	Use to determine whether the state of the resource in a request is current with the resource on the server. This header is supported in order to execute conditional requests. See Checking for Data Consistency.
`If-None-Match`	Use to determine whether the state of the resource in a request does not match the current state on the server. This header is supported in order to execute conditional requests. See Checking for Data Consistency.
`X-HTTP-Method-Override`	Use to execute an action that is otherwise not supported by the server. This is a custom header (not defined by the HTTP specification) that contains the name of an HTTP method as its value. This value (if valid) will be used to define the HTTP method that will be used. This header will only be considered in a POST request. See Overriding the HTTP Method and Performing an Update.
`X-Requested-By`	The presence of this header is enforced for every request when the anti-CSRF mechanism is turned on. The anti-CSRF mechanism can be turned on by setting the value of the `enableAntiCSRF` ResourceServlet init parameter to `True`. This header is not enforced if the GET, OPTIONS, and HEAD methods are used. If the header cannot be found, then 400 (Bad Request) is returned.

Table 22-9 (Cont.) HTTP Headers Supported by the ADF REST Framework

<table>
<tr><th>HTTP Header Name</th><th>Description</th></tr>
<tr><td>Cache-Control</td><td>Use to avoid intermediate proxies to cache/store framework payloads. This header is configured for every HTTP response. The value, by default, turns off caching, and the framework sends the response header: no-cache, no-store, must-revalidate.
However, an application developer may override the default behavior by configuring the resource definition. The value can be set on the resource tree and the header will be set accordingly. Currently, Cache-Control can be configured only as <MaxAge>
&lt;<i>seconds</i>&gt;&lt;/MaxAge>, if set in the resource tree. The following sample of a resource definition file shows how to set the value.<pre>
<pageDefinition ...>
  <parameters/>
  <executables/>
  <bindings>
    <tree ....>
      <ServiceConfiguration>
        <CacheControl>
          <MaxAge>30</MaxAge>
        </CacheControl>
      </ServiceConfiguration>
      <nodeDefinition ../>
    </tree>
  </bindings>
</pageDefinition>
</pre>The integer value supplied within <MaxAge> is used to create the Cache-Control string like max-age=30 to specify the resource will be considered fresh for thirty seconds. This value is also set in the Cache-Control header in the response for an operation involving any resource belonging to the resource tree.
If configured on the resource definition, the Cache-Control value is also shown in the describe, per resource.</td></tr>
<tr><td>Upsert-Mode</td><td>Use Upsert-Mode:true in a request that uses POST to create a resource item if the resource item does not exist, or update a resource item if the resource item exists. Note that a POST request with Upsert-Mode:false behaves as a POST without the custom header and performs the CREATE operation exclusively.</td></tr>
<tr><td>Prefer</td><td>You can use this header to receive only those fields that are included in the compact view of the response payload. You can specify the value return=minimal to receive only those fields that are marked includeInCompactView : true in the response payload. See How to Control the ADF REST Response Payload Fields.</td></tr>
<tr><td>REST-Pretty-Print</td><td>There is a reasonable formatting in REST response resulting in a lot of white space. You can use gzip to transport a zipped up response content. See ADF REST Payload Compression Support. However, the extracted content on the client would still contain the white spaces resulting in processing time on the clients that can be reduced considerably by using this header. See How to Control the Format of the ADF REST Response.</td></tr>
</table>

Table 22-9 (Cont.) HTTP Headers Supported by the ADF REST Framework

HTTP Header Name	Description

`Metadata-Context`	A metadata revision that indicates the MDS label to be used at runtime. A metadata revision is created at the end of an event that updates mainline metadata; for example, MDS MAR deployment or unified sandbox publishing.
	If your environment is unified sandbox enabled, you must use the sandbox property in the value, which is treated as the unified sandbox ID. The Metadata-Context header is honored only if the ApplSessionFilter is the first filter that initializes ADFContext. The mode for ADFSessionOptions is always set to EDIT when this header is present.
	If you specify a revision, ADFSessionOptions will be configured according to the metadata revision.
	If you want to execute the REST service in the context of a specific metadata revision, use the following syntax. This will ensure that the execution of REST service is isolated from other metadata changes happening in the application until the REST client is ready to consume those changes.
	`Metadata-Context: revision="<metadata revision ID>"`
	If you wan to test the REST services with changes in a sandbox, use the following syntax.
	`Metadata-Context: sandbox="<unified sandbox ID>"`

ADF REST HTTP Method and Payload Support

The ADF REST framework supports operations on the following HTTP methods.

- `GET`
- `POST`
- `PATCH`
- `DELETE`

GET Method Operations

The ADF REST framework supports the following operations using a GET method with the URI as shown.

- Describe the resource collection, resource item, or resource catalog (when resource collection and resource item are omitted).

  ```
  http://server/demo/rest/{version}/[{resourceCollectionPath}|
  {resourceItemPath}]/describe
  ```

- Retrieve the resource collection representation with or without a query string parameter.

  ```
  http://server/demo/rest/{version}/{resourceCollectionPath}[?
  {queryStringParam}[&{queryStringParam}]]
  ```

- Retrieve the resource item representation with or without a query string parameter.

  ```
  http://server/demo/rest/{version}/{resourceItemPath}[?{queryStringParam}
  [&{queryStringParam}]]
  ```

- Retrieve a specific version of the resource collection or all available resources (when version identifier and resource collection are omitted).

  ```
  http://server/demo/rest/[{version}/{resourceCollectionPath}]
  ```

Request Parameters

- The GET method supports query string parameters to query, filter, page, and sort the resource representation. The supported parameters are listed in the following tables. All GET method URI parameters can be combined with any other parameter in the table, except where noted on the `expand` and `field` parameters. Note that query string parameters can only be used on resource media types. They cannot, for example, be used when describing the resource.

> **Note:**
>
> The results of the GET method or the query syntax may vary depending on the ADF REST framework version used for a client request. The following tables specify where the framework version is important to note when using query string parameters. For additional information about framework versions, see What You May Need to Know About Versioning the ADF REST Runtime Framework.

Table 22-10 Supported GET Method Query String Parameters Used Only in Resource Collections

GET URI Parameter	Value	Description
q Starting in **ADF REST framework version 2**, q supports complex rowmatch expressions.	In framework version 1, the query parameter is used in the WHERE clause and contains one or more query by example-type expressions, separated by a semi-colon. Format: `<exp1>;<exp2>` Example: `?q=Deptno>=10 and <= 30;Loc!=NY` Starting in framework 2, for richer querying support, the query parameter accepts a rowmatch expression format that identifies the specific rows to retrieve from the resource. The filter can be as simple as a single expression, or you can create more complex filters by combining expressions using the `and` and `or` conjunctions with matching sets of parentheses for grouping. For example, the following expression uses conjunction to query the resource using three different fields: `(AssignedToId is null) or ( (Priority <= 2) and (RecordName like 'TT-99%'))` If a query parameter value has a special character (like ‘;’, ‘,’, ‘=’ or similar), then the value (in the expression) should be enclosed in quotes to define a literal value. If the literal value contains quotes, then, in addition, the quotes need to be escaped, as defined by the framework version, to be viable: • Version 1 syntax encloses the literal value in double quotes (`"`) and requires a backslash (`\`) to escape double quote	The resource collection will be queried using the provided expressions. Supported operators in framework version 1 and later: • = (Equal to) • > (Greater than) • < (Less than) • >= (Greater than or equal to) • <= (Less than or equal to) • `!=` (Not equal to) framework version 1 only • `AND` (And) • `OR` (Or) • `NOT` (Not) • `LIKE` (Like) Supported operators in framework version 2 and later also include: • <> (Not equal to) • `BETWEEN` (Between) • `NOT BETWEEN` (Not between) • `IN` (In) • `NOT IN` (Not in) • `IS NULL` (Null) • `IS NOT NULL` (Not Null) Allowed special characters in framework version 1: • `"` (double quotation mark) to define a literal value for use where special characters appear in the value • `'` (single quotation mark) to define a literal value for use where reserved words appear in the value • `\` (backslash) to define an escape character to escape a double quotation mark (`"`) or single quotation mark/ apostrophe (`'`) character used within a literal value • `*` (asterisk) to define a wildcard character Allowed special characters in framework version 2 and later: • `'` (single quotation mark) to define a literal value or to define an escape character to escape a single quotation mark/apostrophe (`'`) character used within a literal value • `%` (percent) to define a wildcard character For an example of the usage supported in ADF REST framework version 1 and version 2, see Filtering a Resource Collection with a Query Parameter. For examples of rowmatch expressions, see What You May Need to Know About the Advanced Query Syntax in ADF REST Framework Version 2.

Table 22-10 (Cont.) Supported GET Method Query String Parameters Used Only in Resource Collections

GET URI Parameter	Value	Description
	characters contained in the value: `?q=NickName="\"Billy the Kid\""` - Version 2, or later, syntax encloses the literal value in single quotes (`'`) and requires a single quote to escape a single quote character contained in the value (used as an apostrophe): `?q=ListName='Bill''s list'` Additionally, in version 1, if the query parameter value contains a reserved word (such as `AND` or `OR`), then the value needs to be enclosed in single quotes (`'`) and there must be a space before and after the quoted string. For example, the string matching filter `’Accounting and Finance’` in the following query parameter expression contains the reserved word `AND`, and therefore requires a space before and after the single quotation marks to be viable in version 1 syntax: `?q=DepartmentName= 'Accounting and Finance' &fields=DepartmentName` Note that starting in framework version 2, the use of a space character is no longer required to delimit a string matching filter that contains a reserved word. For version 1, if the value contains both a special character and a reserved word, then the value needs to be enclosed in single quotes (`'`), and then enclosed again in double quotes (`"`) , along with escaping the double quotes	

Table 22-10 (Cont.) Supported GET Method Query String Parameters Used Only in Resource Collections

GET URI Parameter	Value	Description
	contained within the value with a backslash (\), and there needs to be a space before and after the twice-quoted string. For example: `?q=DepartmentName= "'\"Accounting\" and Finance'" &fields=DepartmentName` For examples of the rowmatch expression format, see What You May Need to Know About the Advanced Query Syntax in ADF REST Framework Version 2.	

Table 22-10 (Cont.) Supported GET Method Query String Parameters Used Only in Resource Collections

GET URI Parameter	Value	Description
`finder`	An expression containing the primary key attribute value definition. Format: `PrimaryKey;<PKattr1>=<PKval1>,<PKattr2>=<PKval2>,...` Example: ?`finder=PrimaryKey;EmployeeId=101` Or: An expression containing information about the row finder and its finder attributes. Format: `<rowfinderName>;<attr1>=<val1>,<attr2>=<value2>` Example: ?`finder=DeptByName;Dname=ACCOUNTING` If the finder attribute value contains any special character like ';', ',', '=' (or similar), the value must be defined as a literal and enclosed in double quotes. For example: ?`finder=EmpByName;EmpName="Jones,Bill"` If the finder attribute value is a literal value (enclosed in double quotes) and the value contains double quotation marks `"`, the contained quotation marks must be preceded by the backslash escape `\` character to be viable as a finder attribute value. For example: ?`finder=EmpByName;NickName="\"Billy the Kid\""`	The resource collection will be queried using the provided primary key and attribute values definitions. Alternatively, the resource collection will be queried using the provided row finder and attribute definitions. Finders with parameters whose type does not belong to `char`, `date`, or `number`, will be filtered out. The list of available finders (primary key finder and row finder) is provided in the describe. Allowed special characters in framework version 1 and later: • `"` (double quotation mark) to define a literal value • `\` (backslash) to define an escape character to escape a double quotation mark (`"`) character used within a literal value For a primary key finder example, see Filtering a Resource Collection with Primary Key Values. For a row finder example, see Filtering a Resource Collection with a Row Finder.

Table 22-10 (Cont.) Supported GET Method Query String Parameters Used Only in Resource Collections

GET URI Parameter	Value	Description
`totalResults`	boolean Default: `false`	The resource collection representation will include the estimated row count when `totalResults=true`. For an example, see Returning the Estimated Count of Resource Items.
`limit`	integer Default: Iterator binding `RangeSize` property in the ADF REST resource definition file.	This parameter restricts the number of resources returned inside the resource collection. If the limit exceeds the resource total results, then the framework will return the available resources. For an example, see Paging a Resource Collection.
`offset`	integer Default: 0 (the first position)	Used to define the starting position of the resource collection. If offset exceeds the resource count, then no resources are returned. For an example, see Paging a Resource Collection.

Table 22-11 Supported GET Method Query String Parameters Used in Resource Collections or Resource Items

GET URI Parameter	Value	Description
`fields` Starting in **ADF REST framework version 3 and later**, `fields` will return children resource items as a resource collection to support pagination of the collection. For more information, see What Happens at Runtime: Invoking an ADF REST Framework Version.	A simple comma separate list of resource item attributes. Format: `<attr1>,<attr2>` Example: ?`fields=Dname,DLoc` May be used on child resources. Format: *`<accessor>:<att1>,<att2>`* Example: ?`fields=Employees:FirstName,LastName` May be used on nested resources using accessor dot notation. Format: *`<accessor1>.<accessor2>:<Attr1>,<Attr2>`* Example: ?`fields=Employees.JobHistory:JobId` Or on both resources in the nested resource. Format: *`<accessor1>:<Attr1>,<Attr2>;<accessor1>.<accessor2>:<Attr1>,<Attr2>`* Example: ?`fields=Name,Location;Employees:FirstName,LastName`	This parameter filters the resource item attributes. Only the specified attributes are returned. Note that if a nested resource is queried using the accessor dot notation (`Employees.JobHistory`), then attributes may be specified on both resources. A resource in the accessor dot notation without a specified attribute will return no resource item attributes. Note this parameter cannot be combined with the `expand` parameter. If both parameters are provided, only `fields` will be considered. For examples that uses version 1 or 2 of the ADF REST framework as compared to version 3 (and later), see Querying With Filtering Attributes (Partial Get).
`onlyData`	boolean Default: `false`	The representation will be filtered in order to contain only data (no `links` objects, for example). For an example, see Returning Only Resource Data in a Payload.

Table 22-11 (Cont.) Supported GET Method Query String Parameters Used in Resource Collections or Resource Items

GET URI Parameter	Value	Description
`expand` Starting in **ADF REST framework version 3 and later**, `expand` will return children resource items as a resource collection to support pagination of the collection. For more information, see What Happens at Runtime: Invoking an ADF REST Framework Version.	Display all children. Format: `all` Display one or more child resource using a comma-separated list of accessors. Format: `<accessor1>,<accessor2>` Example: `?expand=Employees,Localizations` Display nested resources using the accessor dot notation. Format: `<accessor1>.<accessor2>` Example: `?expand=Employees.JobHistory`	When this parameter is provided, the specified children are included in the resource payload (instead of the link). Note the `expand` parameter cannot be combined with the `fields` parameter. If both parameters are provided, only `fields` will be considered. Note that if a nested resource is queried using the accessor dot notation (`Employees.JobHistory`), then the missing children will be processed implicitly. For example, `?expand=Employees.JobHistory` is the same as `?expand=Employees,Employees.JobHistory` (which will expand `Employees` and `JobHistory`). For examples that uses version 1 or 2 of the ADF REST framework as compared to version 3 (and later), see Fetching Nested Child Resources.
`dependency`	A set of dependency attributes. Format: `<attr1>=<val1>,<attr2>=<value2>` Example: `dependency=ProductId=2`	The dependencies are attributes that are set before and rolled back after generating the response. Generally, they are used to preview the effects of an attribute change as it applies to cascading LOV-enabled attributes. The dependencies attributes are always set in the resource item in question. When a child resource collection is requested and the `dependency` parameter is set, the attributes will be set in the parent resource item before generating the resource collection payload.

Table 22-11 (Cont.) Supported GET Method Query String Parameters Used in Resource Collections or Resource Items

GET URI Parameter	Value	Description

`orderBy`	A comma separated list of order-by attributes with a sort flag to specify ascending or descending order.
	Format: `<orderBy_attr1_name>[:<(asc/desc)>],<orderBy_attr2_name>[:<(asc/desc)>]`
	Example: ?`orderBy=DName:desc,DLoc`
	Default: ORDERBY attributes defined on the view object query backing the resource will be applied.
	To perform case-insensitive sorting on a resource collection, the attribute list must follow this format, using either the `upper` or `lower` flag:
	`upper(<orderBy_attr1_name>):<(asc)>`
	or
	`lower(<orderBy_attr2_name>)`
		Sorts a resource collection based on its attributes. If the `asc/desc` are not provided (or an invalid value is provided), `asc` will be used as default.
		By default, the fetched collection will be sorted in a case sensitive way unless either the `upper` or `lower` flag is specified in the order-by query request.
		For an example, see Sorting a Resource Collection.

`links`	A comma separated list of `<rel_name>`, where `<rel_name>` is a string representing the relation type of a link. Example: `self,canonical`	When a resource item or a resource collection is requested and the `links` query parameter is used, then only those links with relation types ("`rel`") matching the values in the comma-separated parameter value will be shown in the response. Note the `links` parameter cannot be combined with `onlyData` when `onlyData` has a value of `true`, as there will be no links section displayed in the payload.

Table 22-12 Supported GET Method Query String Parameters Used in Resource Catalog Describe

GET URI Parameter	Value	Description

`metadataMode`	`verbose` (default)
	`minimal`
	`list`
		Use to retrieve the description of resources with or without all details. By default, the full catalog is retrieved, performing an exhaustive describe that returns the complete set of information for all resources, including nested children resources. The full catalog describe therefore includes the following sections of information for each resource:
		title, attributes, collection, item, annotations, children, and links
		Note that annotations must be defined by the ADF REST resource developer in the application and may not be present on the resource.
		You can improve the readability of a large catalog and retrieve a shallow catalog limited to the titles and links of parent resources (does not include children resources) by appending the URL parameter `?metadataMode=minimal` to the describe request.
		If you do not want any metadata in the response but only self links, you can append `?metadataMode=list` to the describe request.
		Optionally, additional parameters may be appended to the minimal describe request to include children resources and / or resource annotations, as explained for the query parameters `showAnnotations` and `includeChildren`. For example, you can append `?metadataMode=minimal&includeChildren=true` to retrieve a minimal catalog describe with all children resources included.
		For an example, see Retrieving a Minimal Catalog Describe.

`showAnnotations`	`true`, `false` (default depends on whether full or minimal catalog is retrieved)	Use to either exclude or include resource annotations, as specified in the application. The default depends on the `metadataMode` parameter value: • By default, a full catalog (`metadataMode=verbose`) describe includes annotations. To exclude resource annotations from the full catalog describe, you can append `?showAnnotations=false` on the describe request. • By default, a minimal catalog (`metadataMode=minimal`) describe excludes annotations. To include resource annotations in the minimal catalog describe, you can append `?showAnnotations=true` on the describe request. Note that annotations must be defined by the ADF REST resource developer in the application and may not be present on the resource. You cannot use this parameter with `?metadataMode=list`.

Table 22-12 (Cont.) Supported GET Method Query String Parameters Used in Resource Catalog Describe

GET URI Parameter	Value	Description
`include`	`public` (default), `unlisted`, `all`	Use to retrieve the description of resources of a certain visibility, as declared in the application. This URL parameter values specify including only public resources (`public`), only private resources (`unlisted`), or both types (`all`). By default, only resources that are public are shown in the describe. To view private resources, you must append either `?include=all` or `?include=unlisted` on the describe request. For an example, see Retrieving the Catalog Describe Based on Resource Visibility Declaration. For information about how the application developer declares visibility in the resource definition, see How to Hide a Resource from the Catalog Describe.
`resources`	A comma separated list of resource collection names. Format:`<resName1>,<resName2>` Example: `?resources=Departments,Employees`	Use to retrieve the description of the named resources to filter resources at the catalog level.

Table 22-13 Supported GET Method Query String Parameters Used in Resource Catalog Describe or Resource Item Describe

GET URI Parameter	Value	Description
`includeChildren`	`true`, `false` (default depends on whether full or minimal catalog is retrieved)	Use to either exclude or include all available children resources nested within a parent resource. The default depends on the `metadataMode` parameter value: • By default, a full catalog (`metadataMode=verbose`) describe includes all children resources. To exclude children resources from the full catalog describe, you can append `?includeChildren=false` on the describe request. • By default, a minimal catalog (`metadataMode=minimal`) describe excludes all children resources. To include children resources in the minimal catalog describe, you can append `?includeChildren=true` on the describe request. You cannot use this parameter with `?metadataMode=list`. For a resource catalog describe example, see Retrieving a Minimal Catalog Describe. When a resource item describe is requested and `?includeChildren=true` is provided, all children will be recursively included in the describe. For an example, see Describing a Resource Item.

Query String Operators Supported by ADF REST Data Types

The following table shows the ADF REST data types and the valid operators that may be used in query parameter strings. Note that the operators BETWEEN, NOT BETWEEN, IN, NOT IN, and the wildcard character % are available only starting in ADF REST framework version 2.

Table 22-14 Operators Supported by Data Types in Query (q) Parameter Strings

ADF REST Data Type	Supported Operator

integer	• = (Equal to)
	`.../Departments?q=Deptno = 20`
	• <> (Not equal to)
	`.../Departments?q=Deptno <> 20`
	• < (Less than)
	`.../Departments?q=Deptno < 20`
	• <= (Less than or equal to)
	`.../Departments?q=Deptno <= 20`
	• > (Greater than)
	`.../Departments?q=Deptno > 30`
	• >= (Greater than or equal to)
	`.../Departments?q=Deptno >= 30`
	• `BETWEEN` (Between)
	`.../Departments?q=Deptno BETWEEN 10 AND 30`
	• `NOT BETWEEN` (Not between)
	`.../Departments?q=Deptno NOT BETWEEN 10 and 30`
	• `IN` (In)
	`.../Departments?q=Deptno IN (10, 30)`
	• `NOT IN` (Not in)
	`.../Departments?q=Deptno NOT IN (10, 30)`
	• `IS NULL` (Is null)
	`.../Departments?q=Deptno IS NULL`
	• `NOT NULL` (Not null)
	`.../Departments?q=Deptno NOT NULL`

Table 22-14 (Cont.) Operators Supported by Data Types in Query (q) Parameter Strings

ADF REST Data Type	Supported Operator

number	• = (Equal to)
	`.../Departments?q=Salary = 3120.99`
	• <> (Not equal to)
	`.../Departments?q=Salary <> 3120.99`
	• < (Less than)
	`.../Departments?q=Salary < 3120.99`
	• <= (Less than or equal to)
	`.../Departments?q=Salary <= 3120.99`
	• > (Greater than)
	`.../Departments?q=Salary > 3120.99`
	• >= (Greater than or equal to)
	`.../Departments?q=Salary >= 3120.99`
	• `BETWEEN` (Between)
	`.../Departments?q=Salary BETWEEN 2000 AND 3120.99`
	• `NOT BETWEEN` (Not between)
	`.../Departments?q=Salary NOT BETWEEN 2000 and 3120.99`
	• `IN` (In)
	`.../Departments?q=Salary IN (800, 3120.99)`
	• `NOT IN` (Not in)
	`.../Departments?q=Salary NOT IN (800, 3120.99)`
	• `IS NULL` (Is null)
	`.../Departments?q=Salary IS NULL`
	• `NOT NULL` (Not null)
	`.../Departments?q=Salary NOT NULL`

Table 22-14 (Cont.) Operators Supported by Data Types in Query (q) Parameter Strings

ADF REST Data Type	Supported Operator

string	• = (Equal to)
	`.../Departments?q=DeptName = ’SALES’`
	• <> (Not equal to)
	`.../Departments?q=DeptName <> ’SALES’`
	• `LIKE` (Like)
	`.../Departments?q=DeptName LIKE ’SA%’`
	`.../Departments?q=DeptName LIKE ’%ES’`
	`.../Departments?q=UPPER(DeptName) LIKE UPPER('%e%')`
	• `NOT LIKE` (Not like)
	`.../Departments?q=UPPER(DeptName) NOT LIKE UPPER('%c%')`
	• `IN` (In)
	`.../Departments?q=DeptName IN ('SALES', 'RESEARCH')`
	• `NOT IN` (Not in)
	`.../Departments?q=DeptName NOT IN ('SALES', 'RESEARCH')`
	• `IS NULL` (Is null)
	`.../Departments?q=DeptName IS NULL`
	• `IS NOT NULL` (Is not null)
	`.../Departments?q=DeptName IS NOT NULL`

date	• = (Equal to)
	`.../Employees?q=HireDate = ’1999-01-01’`
	• <> (Not equal to)
	`.../Employees?q=HireDate <> ’1999-01-01’`
	• < (Less than)
	`.../Employees?q=HireDate < ’1999-01-01’`
	• <= (Less than or equal to)
	`.../Employees?q=HireDate <= ’1999-01-01’`
	• > (Greater than)
	`.../Employees?q=HireDate > ’1999-01-01’`
	• >= (Greater than or equal to)
	`.../Employees?q=HireDate >= ’1999-01-01’`
	• `BETWEEN` (Between)
	`.../Employees?q=HireDate BETWEEN ’1999-01-01’ AND ’2010-01-01’`
	• `NOT BETWEEN` (Not between)
	`.../Employees?q=HireDate NOT BETWEEN ’1999-01-01’ AND ’2010-01-01’`
	• `IS NULL` (Is null)
	`.../Employees?q=HireDate IS NULL`
	• `NOT NULL` (Not null)
	`.../Employees?q=HireDate NOT NULL`

Table 22-14 (Cont.) Operators Supported by Data Types in Query (q) Parameter Strings

ADF REST Data Type	Supported Operator

time	• = (Equal to)
	`.../Employees?q=HireTime = '08:30:40'`
	• <> (Not equal to)
	`.../Employees?q=HireTime <> ’08:30:40’`
	• < (Less than)
	`.../Employees?q=HireTime < ’08:30:40’`
	• <= (Less than or equal to)
	`.../Employees?q=HireTime <= ’08:30:40’`
	• > (Greater than)
	`.../Employees?q=HireTime > ’08:30:40’`
	• >= (Greater than or equal to)
	`.../Employees?q=HireTime >= ’08:30:40’`
	• `BETWEEN` (Between)
	`.../Employees?q=HireTime BETWEEN ’04:30:00’ AND '08:30:40'`
	• `NOT BETWEEN` (Not between)
	`.../Employees?q=HireTime NOT BETWEEN ’04:30:00’ AND '08:30:40'`
	• `IS NULL` (Is null)
	`.../Employees?q=HireTime IS NULL`
	• `NOT NULL` (Not null)
	`.../Employees?q=HireTime NOT NULL`

Table 22-14 (Cont.) Operators Supported by Data Types in Query (q) Parameter Strings

ADF REST Data Type	Supported Operator

datetime
Note: Both UTC and local datetime formats are supported. The value returned is determined by the time zone configured for the VM.
	• = (Equal to)
	`.../Employees?q=HireDateTime = '1999-01-01T08:30:40Z'`
	• <> (Not equal to)
	`.../Employees?q=HireDateTime <> ’1999-01-01T08:30:40Z’`
	• < (Less than)
	`.../Employees?q=HireDateTime < ’1999-01-01T08:30:40Z’`
	• <= (Less than or equal to)
	`.../Employees?q=HireDateTime <= ’1999-01-01T08:30:40Z’`
	• > (Greater than)
	`.../Employees?q=HireDateTime > ’1999-01-01T08:30:40Z’`
	• >= (Greater than or equal to)
	`.../Employees?q=HireDateTime >= ’1999-01-01T08:30:40Z’`
	• `BETWEEN` (Between)
	`.../Employees?q=HireDateTime BETWEEN ’1999-01-01T08:30:40Z’ AND ’1999-12-01T08:30:40Z’`
	• `NOT BETWEEN` (Not between)
	`.../Employees?q=HireDateTime NOT BETWEEN ’1999-01-01T08:30:40Z’ AND ’1999-12-01T08:30:40Z’`
	• `IS NULL` (Is null)
	`.../Employees?q=HireDateTime IS NULL`
	• `NOT NULL` (Not null)
	`.../Employees?q=HireDateTime NOT NULL`

boolean	• `= ’true’` (true)
	`.../Employees?q=Active = ’true’`
	• `= ’false’` (false)
	`.../Employees?q=Active = ’false’`
	• `IS NULL` (Is null)
	`.../Employees?q=Active IS NULL`
	• `NOT NULL` (Not null)
	`.../Employees?q=Active NOT NULL`

Media Types Supported

- **Request**
  - None
- **Response**

  - `application/vnd.oracle.adf.resourcecollection+json`: When retrieving a resource collection.
  - `application/vnd.oracle.adf.resourceitem+json`: When retrieving a resource item.
  - `application/vnd.oracle.adf.description+json`: When describing a resource.
  - `application/vnd.oracle.adf.version+json`: When retrieving all available resource versions.

Use Case Samples

- Retrieving the ADF REST Resource Describe
- Querying With Filtering Attributes (Partial Get)
- Filtering a Resource Collection with a Query Parameter
- Filtering a Resource Collection with Primary Key Values
- Filtering a Resource Collection with a Row Finder
- Checking for Data Consistency When Retrieving ADF REST Resource Items
- Retrieving All Available Version Release Names
- Retrieving a Resource By a Specific Version
- Streaming Attachments Using a Resource Item Enclosure Link
- Retrieving LOV-Enabled Attribute Values for Existing Resource Items
- Returning Only Resource Data in a Payload
- Returning the Estimated Count of Resource Items

ADF REST Framework Version Overview

- What You May Need to Know About Versioning the ADF REST Framework
- What You May Need to Know About the Advanced Query Syntax in ADF REST Framework Version 2
- What Happens At Runtime: Invoking an ADF REST Framework Version

POST Method Operations

The ADF REST framework supports the following operations using a POST method with the URI as shown.

- Create a new resource.

  `http://server/demo/rest/{version}/{resourceCollectionPath}`

- Create a parent resource item and create the nested child resource collection in one roundtrip.

  `http://server/demo/rest/{version}/{resourceCollectionPath}`

- Updating or creating resource items using Upsert-Mode Header

  `http://server/demoapp/rest/{version}/{resourceCollectionPath}`

- Execute an action on a resource collection or resource item.

`http://server/demo/rest/{version}/{resourceCollectionPath}|{resourceItemPath}`

- Execute an a batch request.

  `http://server/demo/rest/{version}/{resourceCollectionPath}`

Request Parameters

- none

Media Types Supported

- **Request**
  - `application/vnd.oracle.adf.resourceitem+json`: When creating a resource or overriding the HTTP method to perform an update.
  - `application/vnd.oracle.adf.resourceitem+json`: When updating or creating a resource item using Upsert.
  - `application/vnd.oracle.adf.action+json`: When executing an action.
  - `application/vnd.oracle.adf.batch+json`: When executing a batch request.
- **Response**
  - `application/vnd.oracle.adf.resourceitem+json`: When executing an action or creating a resource.
  - `application/vnd.oracle.adf.resourceitem+json`: When updating or creating a resource item using Upsert.
  - `application/vnd.oracle.adf.actionresult+json`: When executing an action that has an object to return.
  - `application/vnd.oracle.adf.batch+json`: When executing an batch request.

Use Case Samples

- Creating a Resource Item in Collection
- Creating Child Resource Item
- Streaming Attachments Using a Resource Item Enclosure Link
- Overriding the HTTP Method and Performing an Update
- Making Batch Requests
- Updating or Creating Resource Items Using Upsert

PATCH Method Operations

The ADF REST framework supports the following operation using a PATCH method with the URI as shown.

- Updating a resource item.

  `http://server/demo/rest/{version}/{resourceItemPath}`

Request Parameters

- none

Media Types Supported

- **Request**
  - `application/vnd.oracle.adf.resourceitem+json`: The resource item to be updated.
- **Response**
  - `application/vnd.oracle.adf.resourceitem+json`: The updated resource item.

Use Case Samples

- Updating a Resource Item
- Checking for Data Consistency When Updating ADF REST Resource Items
- Replacing LOB Content Using Base64
- Overriding the HTTP Method and Performing an Update

DELETE Method

The ADF REST framework supports the following operation using a DELETE method with the URI as shown.

- Deleting a resource item.

  `http://server/demo/rest/{version}/{resourceItemPath}`

Request Parameters

- none

Media Types Supported

- **Request**
  - none
- **Response**
  - none

Use Case Samples

- Deleting a Resource Item

Part IV
Creating ADF Task Flows

This part describes the tasks developers can perform when creating ADF task flows in the Fusion web application. ADF task flows provide a modular approach for defining control flow in a Fusion web application.

Part IV contains the following chapters:

- Getting Started with ADF Task Flows
- Working with Task Flow Activities
- Using Parameters in Task Flows
- Using Task Flows as Regions
- Creating Complex Task Flows
- Using Dialogs in Your Application

23 Getting Started with ADF Task Flows

This chapter describes getting started with ADF tasks flows that you can create in your Fusion web application. In addition to describing how to create ADF task flows, key ADF task flow features such as activities, control flows, control flow rules, memory scopes, and bounded and unbounded task flows are described. The chapter also describes how to refactor, test, and run a finished task flow.
This chapter includes the following sections:

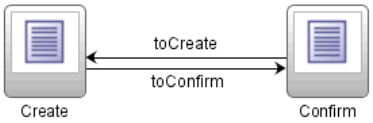
About ADF Task Flows

An ADF task flow is a graphical representation of the connectivity between the activity nodes. This method allows you to create a modular set of reusable task flows and join them through control flow cases to represent an application. The two types of task flows are, Unbounded task flow and Bounded task flow.

ADF task flows provide a modular approach for defining control flow in a Fusion web application. Instead of representing an application as a single large JSF page flow, you can break it up into a collection of reusable task flows. Each task flow contains a portion of the application's navigational graph. The nodes in the task flows are activities. An **activity node** represents a simple logical operation such as displaying a page, executing application logic, or calling another task flow. The transitions between the activities are called **control flow cases**.

Figure 23-1 shows two view activities called `Create` and `Confirm`. These view activities are similar to page nodes within a JSF page flow.

Figure 23-1 ADF Task Flow



The are two types of ADF task flow:

- Unbounded task flow: A set of activities, control flow rules, and managed beans that interact to allow a user to complete a task. The unbounded task flow consists of all activities and control flows in an application that are not included within a bounded task flow.
- Bounded task flow: A specialized form of task flow that, in contrast to the unbounded task flow, has a single entry point (an entry point is a view activity that can be directly requested by a browser) and zero or more exit points. It contains its own set of private control flow rules, activities, and managed beans. A bounded task flow allows reuse, parameters, transaction management, reentry, and can render within an ADF region in a JSF page.

  For a description of the activity types that you can add to an unbounded or bounded task flow see Working with Task Flow Activities .

  By default, JDeveloper proposes the following file name for the source file of a bounded task flow:

  ```
  task-flow-definitionN.xml
  ```

  where `N` is a number that increments each time that you create a new bounded task flow. You can choose a file name for the bounded task flow. For example, you might use `checkout-task-flow.xml` if the purpose of the bounded task flow is to allow customers to check out from a shopping application.

  The source file contains the metadata for the bounded task flow. Multiple bounded task flows can be included within the same source file using the following metadata:

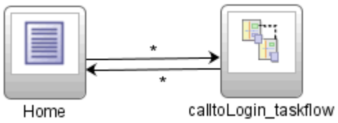
  ```
  <task-flow-definition id="TaskFlowID1">
          ...
        </task-flow-definition>
  <task-flow-definition id="TaskFlowID2">
         ...
  </task-flow-definition>
  ```

  However, as JDeveloper's visual editors show one bounded task flow per file, it makes the modification of bounded task flows easier if you do not include more than one bounded task flow within the same source file.

A typical application is a combination of the unbounded task flow and one or more bounded task flows. For example, JDeveloper, by default, creates an empty unbounded task flow (source file name is `adfc-config.xml`) when you create an application using the Fusion Web Application template. At runtime, the Fusion web application can call bounded task flows from activities that you added to the unbounded task flow.

As shown in Figure 23-2, the first activity to execute in an application is often a view activity within the unbounded task flow. A view activity represents a JSF page that displays as part of the application. The activity shown in Figure 23-2 starts with the `Home` view activity and then calls a bounded task flow. The `calltoLogin_taskFlow` activity calls a bounded task flow that enables a user to log into the application.

Figure 23-2 Unbounded Task Flow Calling a Bounded Task Flow



You can also design an application in which all application activities reside within the unbounded flow. This mimics a JSF application, but does not take advantage of bounded task flow functionality. To take full advantage of task flow functionality, use bounded task flows.

Table 23-1 describes the advantages that the control flow provided by ADF task flows offers over the control flow offered by JSF page flow. In certain scenarios you may need to use JSF page flow. For example, using phase listeners, as described in Customizing the ADF Page Lifecycle , involves using the JSF page flow's `faces-config.xml` configuration file. In general, it is not recommended that you mix JSF page flow and ADF task flows in your application.

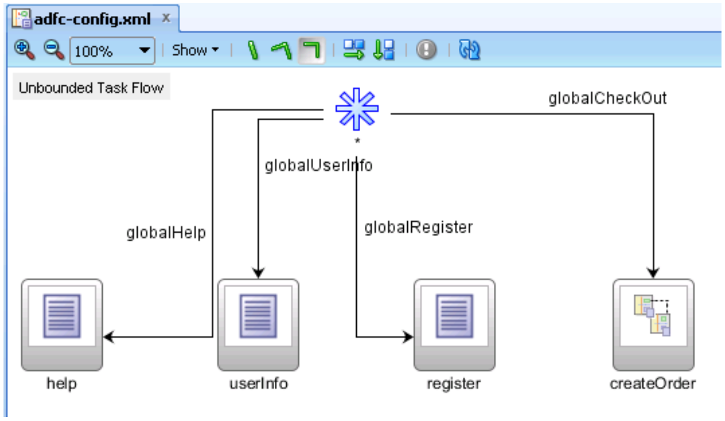
Table 23-1 ADF Task Flow Advantages

JSF Page Flow	ADF Task Flow
The entire application must be represented in a single page navigation file (`faces-config.xml`). Although you can have multiple copies of `faces-config.xml` in a project, the application loads these files as one at runtime.	The application can be broken up into a series of modular flows that call one another.
All nodes within a JSF page flow must be JSF pages. No other types of objects can exist within the JSF page flow.	You can add to the task flow diagram nodes such as views, method calls, and calls to other task flows.
Navigation is only between pages.	Navigation is between pages as well as other activities, including routers. See, see Using Router Activities.
Application fragments cannot be reused.	ADF task flows are reusable within the same or an entirely different application. After you break up your application into task flows, you may decide to reuse task flows containing common functionality. See, see Reusing Application Components .
There is no shared memory scope between multiple requests except for session scope.	Shared memory scope (for example, page flow scope) enables data to be passed between activities within the task flow. Page flow scope defines a unique storage area for each instance of a bounded task flow.

About Unbounded Task Flows

A Fusion web application always contains an **ADF unbounded task flow**, which contains the entry point or points to the application. An entry point is a view activity that can be directly requested by a browser. A Fusion web application only has one unbounded task flow. By default, the source file for the unbounded task flow is the `adfc-config.xml` file. Although you can create additional source files for unbounded task flows, the application combines all source files at runtime into the `adfc-config.xml` file. Figure 23-3 displays the diagram for an unbounded task flow from an application. This task flow contains a number of view activities that are all entry points to the application.

Figure 23-3 Unbounded Task Flow



You typically use the unbounded task flow instead of a bounded task flow if:

- You want to take advantage of ADF Controller features not offered by bounded task flows, such as bookmarkable view activities. For more information, see Bookmarking View Activities.
- The task flow will not be called by another task flow.
- The application has multiple points of entry. In Figure 23-3, the task flow can be entered through any of the pages represented by the view activity icons on the unbounded task flow.

The unbounded task flow cannot declaratively specify parameters. In addition, it cannot contain a **default activity**, an activity designated as the first to run in the unbounded task flow. This is because the unbounded task flow does not have a single point of entry. To perform any of these requires a bounded task flow.

In order to take advantage of completely declarative ADF Controller transaction and reentry support, use a bounded task flow rather than the unbounded task flow.

About Bounded Task Flows

An **ADF bounded task flow** is used to encapsulate a reusable portion of an application. A bounded task flow is similar to a Java method in that it:

- Has a single entry point
- May accept input parameters
- May generate return values
- Has its own collection of activities and control flow rules

- Has its own memory scope and managed bean lifespan (a page flow scope instance)

  For more information about memory scopes, see What You May Need to Know About Memory Scope for Task Flows.

- Can determine whether or not it has an isolated data control frame

  For more information about access to data control frames, see Sharing Data Controls Between Task Flows .

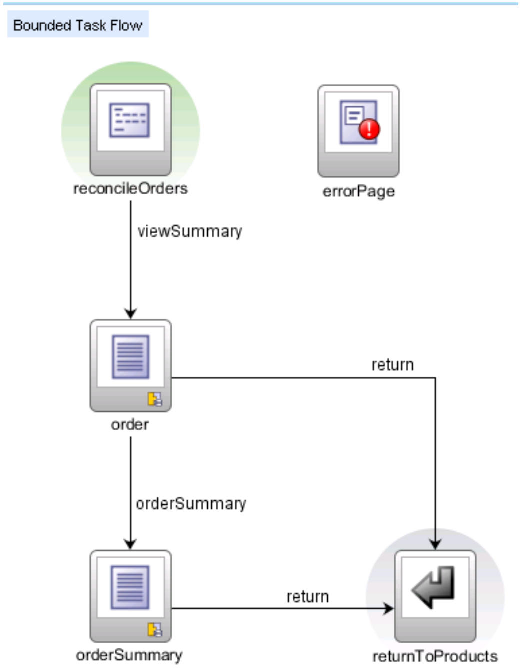
- Can define an activity as an exception handling activity

  For more information about exception handling in bounded task flows, see Handling Exceptions in Task Flows.

The `createOrder` activity in Figure 23-3 is a call to a bounded task flow. The unbounded task flow can call a bounded task flow, but cannot be called by another task flow. A bounded task flow can call another bounded task flow, which can call another and so on. There is no limit to the depth of the calls.

Figure 23-4 shows a bounded task flow.

Figure 23-4 Bounded Task Flow



The reasons for creating a bounded task flow are:

- The bounded task flow always specifies a default activity, a single point of entry that must execute immediately upon entry of the bounded task flow.

  In Figure 23-4, this is the method call activity labeled **reconcileOrders**. Because it is the default activity, the method call activity is always invoked before the bounded task flow renders the page referenced by the **order** view activity.

- It is reusable. For example, it can be included in other applications requiring a process to manage or check out orders. The bounded task flow can also be reused within the same application.
- Any managed beans you use within a bounded can be specified in page flow scope, so are isolated from the rest of the application. These managed beans (with page flow scope) are automatically released when the task flow completes.

Table 23-2 summarizes the main features of a bounded task flows.

Table 23-2 Bounded Task Flow Features

Feature	Description
Well-defined boundary	A bounded task flow consists of its own set of private control flow rules, activities, and managed beans. A caller requires no internal knowledge of page names, method calls, child bounded task flows, managed beans, and control flow rules within the bounded task flow boundary. Input parameters can be passed into the bounded task flow, and return values can be passed out on exit of the bounded task flow. Data controls can be shared between task flows.
Single point of entry	A bounded task flow has a single point of entry, a default activity that executes before all other activities in the task flow. For more information, see What You May Need to Know About the Default Activity in a Bounded Task Flow.
Page flow memory scope	You can specify page flow scope as the memory scope for passing data between activities within the bounded task flow. Page flow scope defines a unique storage area for each instance of a bounded task flow. Its lifespan is the bounded task flow, which is longer than request scope and shorter than session scope. For more information, see What You May Need to Know About Memory Scope for Task Flows.
Addressable	You can access a bounded task flow by specifying its unique identifier within the XML source file for the bounded task flow and the file name of the XML source file. For more information, see What Happens When You Add a Task Flow Call Activity .
Reuse	You can identify an entire group of activities as a single entity, a bounded task flow, and reuse the bounded task flow in another application within an ADF region. You can also reuse an existing bounded task flow simply by calling it. For example, one task flow can call another bounded task flow using a task flow call activity or a URL. In addition, you can use task flow templates to capture common behaviors for reuse across different bounded task flows. For more information, see Creating Task Flow Templates .
Parameters and return values	A caller can pass input parameters to a bounded task flow and accept return values from it. For more information, see Passing Parameters to a Bounded Task Flow. In addition, you can share data controls between bounded task flows. For more information, see Sharing Data Controls Between Task Flows .

Table 23-2 (Cont.) Bounded Task Flow Features

Feature	Description
Transaction management	A bounded task flow can represent a transactional unit of work. You can declaratively specify options on the bounded task flow that determine whether, when entering the task flow, the task flow creates a new transaction, joins an existing one or is not part of the existing transaction. For more information, see Managing Transactions in Task Flows.
Reentry	You can specify options on the bounded task flow that determine whether or not it can be reentered. For more information, see Reentering Bounded Task Flows.
On-demand loading of metadata	Bounded task flow metadata is loaded on demand when entering a bounded task flow.
Security	You can secure a bounded task flow by defining the privileges that are required for someone to use it. For more information, see Enabling ADF Security in a Fusion Web Application.

Understanding the assumptions and constraints listed in Table 23-3 will help you successfully use ADF task flows, activities and other associated ADF Controller features.

Table 23-3 ADF Controller Features Assumptions and Constraints

Feature Area	Assumption/ Constraint	Description
ADF Controller objects and diagram UI	JSF view layer	ADF Controller operates in a JSF environment. Oracle's web-based Fusion web application strategy focuses on JSF as the sole view layer technology.
ADF Controller objects and diagram UI	Dependent on Oracle ADF Faces	ADF Controller extensions are implemented on top of Oracle ADF Faces. They are dependent on the ADF Faces libraries, but ADF Controller can run against any JSF implementation, providing these libraries are present.
ADF Controller objects and diagram UI	Navigation and state management encapsulated	ADF Controller encapsulates both navigation and, to some extent, state management. JSF and the Servlet API are still available for the basic management of state at the application, session, and request levels.
ADF Controller objects and diagram UI	Model layer	ADF Model layer is used to implement the application's model layer.
ADF Controller objects and diagram UI	Dependent on MDS	ADF Controller metadata is stored in MDS. However, MDS is currently not capable of loading `faces-config.xml`. If the customization features that MDS provides are required, you should use ADF task flows exclusively in order to define managed beans and control flow rules.

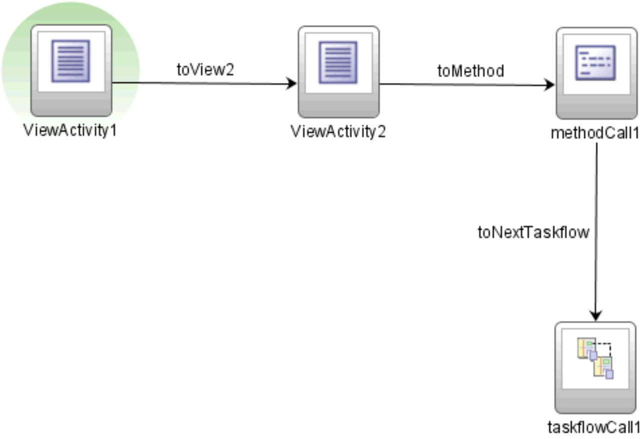
Table 23-3 (Cont.) ADF Controller Features Assumptions and Constraints

Feature Area	Assumption/ Constraint	Description
ADF Controller objects and diagram UI	No supported migration path from struts or model 1	There is no support for a migration from Struts or Model 1 to the ADF Controller. However, you can create a new bounded task flow based on selected pages in a JSF page flow. For more information, see How to Create a Task Flow from JSF Pages.
Bounded task flow	Exposed as page flow-scoped state	ADF Controller manages implementation of a page flow scoped-state. Any auto-management functions provided by the framework, such as back button support and state cleanup function, assume page flow-scoped data. In order for an application to fully implement such functions for all of its pages, the entire application should be exposed as an ADF bounded task flow, using nested bounded task flows as needed. The application should store any state requiring versioning within the page flow scope.
Bounded task flow	Transactional boundaries	The developer will use ADF bounded task flows to manage transaction boundaries.
Page flow scope	Access availability within ADF lifecycle	An application cannot attempt to access the page flow scope early in the ADF lifecycle before ADF Controller is ready to provide it. Page flow scope is not guaranteed to be available for access until after Before and After listeners have executed on the Restore View phase. ADF Controller uses before and after listeners on the Restore View phase to synchronize the server side state with the request. This is where things such as browser back-button detection and bookmark dereference are handled.
Navigation	Navigation	When using ADF Controller task flows, perform all application navigation should be performed using ADF Controller control flow rules instead of using navigation rules in `faces-config.xml`. Although ADF Controller delegates navigation handling when no matching control flow cases are found in ADF Controller metadata, not all ADF Controller functionality is guaranteed to work correctly if navigation is performed by a non-ADF Controller `NavigationHandler`.

About Control Flows

A task flow consists of activities and control flow cases that define the transitions between activities. Figure 23-5 shows a control flow rule, labeled `toView2`, that defines the transition between the `ViewActivity1` and `ViewActivity2` view activities. The `ViewActivity1` view activity displays before the `ViewActivity2` view activity when the task flow in Figure 23-5 executes.

Figure 23-5 Task Flow with Activities and Control Flow Cases

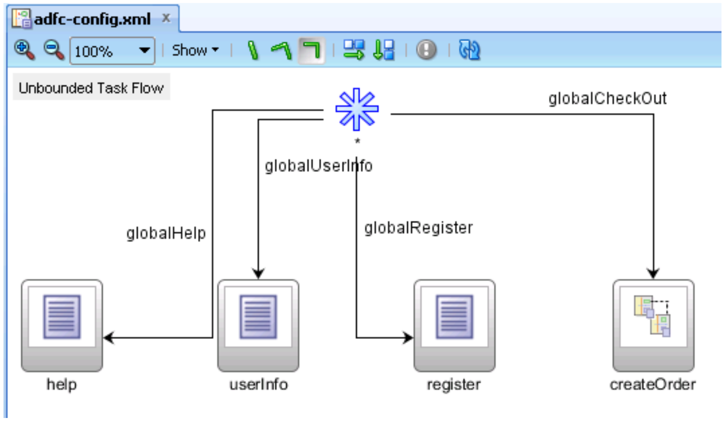


The task flow in Figure 23-5 also contains a method call activity (`methodCall1`) that invokes after the `ViewActivity2` view activity and before the `taskflowCall1` task flow call activity. In a task flow, you invoke an activity such as a method call activity before or after a page renders. Invoking a method call activity outside of a particular page can facilitate reuse because you can reuse the page in other contexts that do not require the method (for example, a different task flow). For more information about control flow rules, see Adding Control Flow Rules to Task Flows.

A wildcard control flow rule represents a control flow `from-activity-id` that contains a trailing wildcard (`foo*`) or a single wildcard character (`*`). Use the single wildcard character when you want to pass control from any activity in the task flow to the wildcard control flow rule. Alternatively, use a trailing wildcard when you want to constrain the activities that can pass control to the wildcard control flow rule.

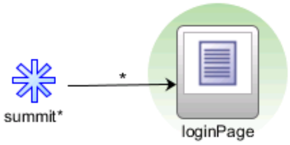
In Figure 23-6, the wildcard control flow rule contains a single wildcard character, indicating that control can pass to the activities connected to it in the task flow diagram from any activity within the task flow.

Figure 23-6   Wildcard Control Flow Rule With Single Wildcard



The trailing wildcard in Figure 23-7 indicates that control flow can pass to the `loginPage` view from any valid source activity whose `activity-id` begins with the characters `summit`.

Figure 23-7   Wildcard Control Flow Rule with Trailing Wildcard

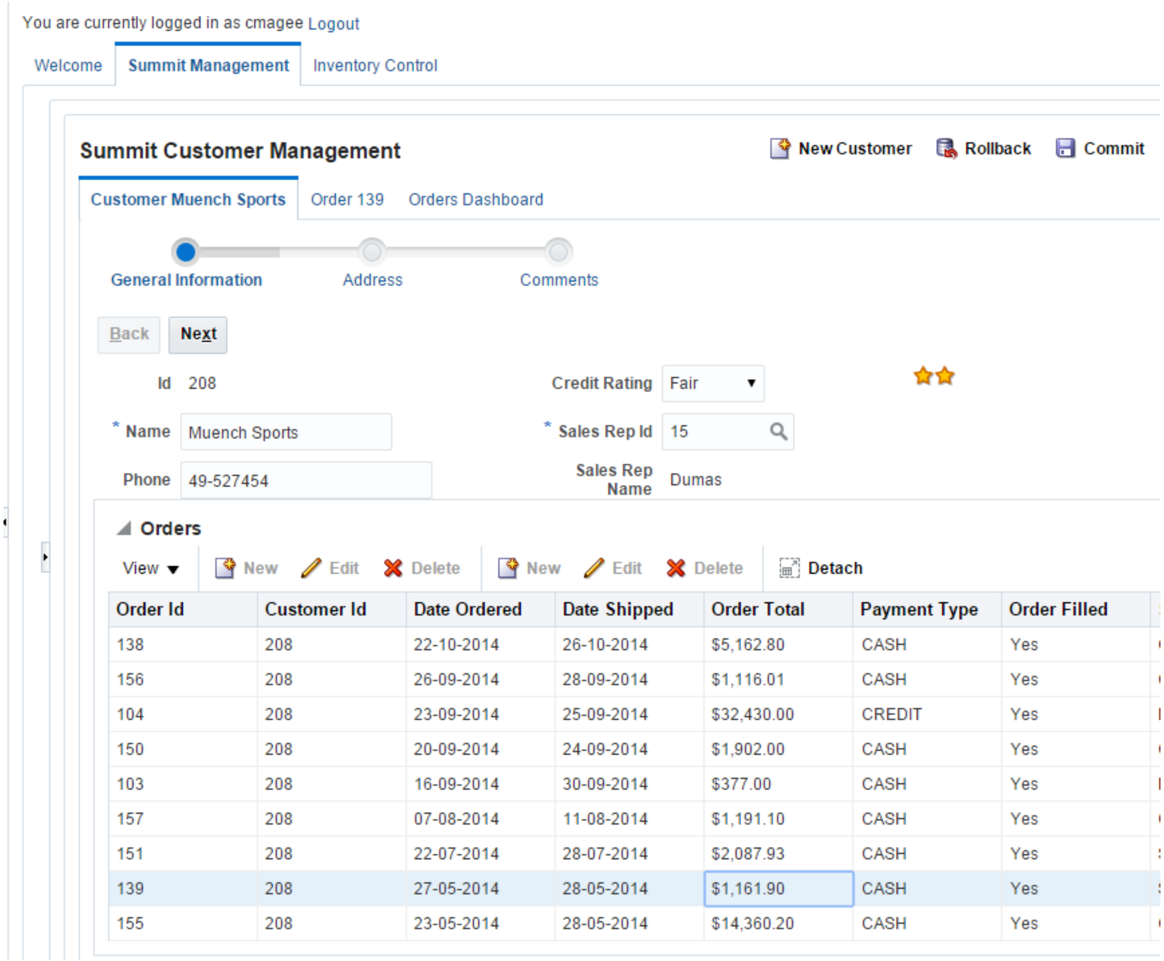


For more information about wildcard control flow rules, see How to Add a Wildcard Control Flow Rule .

ADF Task Flow Use Cases and Examples

Figure 23-8 shows a screen from the Summit sample application for ADF task flows. End users of the screen invoke a task flow that displays a list of customers. Using controls exposed on the screen, end users can also invoke a task flow that allows them to create or to edit an order.

Figure 23-8 Task Flows for Creating and Editing Data



Id	208	Credit Rating	Fair
* Name	Muench Sports	* Sales Rep Id	15
Phone	49-527454	Sales Rep Name	Dumas

Order Id	Customer Id	Date Ordered	Date Shipped	Order Total	Payment Type	Order Filled
138	208	22-10-2014	26-10-2014	$5,162.80	CASH	Yes
156	208	26-09-2014	28-09-2014	$1,116.01	CASH	Yes
104	208	23-09-2014	25-09-2014	$32,430.00	CREDIT	Yes
150	208	20-09-2014	24-09-2014	$1,902.00	CASH	Yes
103	208	16-09-2014	30-09-2014	$377.00	CASH	Yes
157	208	07-08-2014	11-08-2014	$1,191.10	CASH	Yes
151	208	22-07-2014	28-07-2014	$2,087.93	CASH	Yes
139	208	27-05-2014	28-05-2014	$1,161.90	CASH	Yes
155	208	23-05-2014	28-05-2014	$14,360.20	CASH	Yes

Task flows can also be used to secure your Fusion web application by reducing the number of access points that you expose to end users. For example, configure an unbounded task flow to display one page that provides navigation to the remaining pages in your application. Use bounded task flows for the remaining pages in the application. By setting the URL Invoke property of these bounded task flows to `url-invoke-disallowed`, your application has one access point (the page on the unbounded task flow). For more information about the URL Invoke property, see How to Call a Bounded Task Flow Using a URL. For more information about securing your Fusion web application, see Enabling ADF Security in a Fusion Web Application.

Additional Functionality for ADF Task Flows

You may find it helpful to understand other **Oracle ADF** features before you configure or use ADF task flows. Additionally, you may want to read about what you can do with your task flow configurations. Following are links to other functionality that may be of interest.

- Task flows can invoke managed beans. For more information about defining managed beans for use with a task flow, the supported memory scopes, and other related information, see What You May Need to Know About Memory Scope for Task Flows and Using a Managed Bean in a Fusion Web Application.

- Task flows are reusable. For more information about reusing functionality in your application, see Reusing Application Components .
- Task flows can be secured by defining the privileges that are required for someone to use it. For more information, see Enabling ADF Security in a Fusion Web Application.
- You can extend the functionality that your task flows implement by writing custom code. For example, you can write a custom exception handler that a task flow passes control to when a task flow activity raises an exception. For more information, see How to Designate Custom Code as an Exception Handler.

  Make sure when you write custom code that you do not import packages that are marked internal, as in the following example:

  ```
  import oracle.adfinternal.controller.*;
  ```

  For information about the APIs that you can use to write custom code, see the following reference documents:

  – *Java API Reference for Oracle ADF Controller*

  – *Java API Reference for Oracle ADF Faces*

Creating a Task Flow

To create an ADF task flow you must identify the activities nodes and the control flow rules between them. Also, a task flow can be created by joining multiple task flows. You can use the router activity for branching in the task flow and use the return activity to exit the task flow.

A task flow is made up of the task flow itself, plus a number of activities with control flow rules between those activities. In most cases, the majority of the activities are view activities which represent the different pages in the flow. When some method or operation needs to be called, for example before a page is rendered, you use a method call activity with a control flow case from that activity to the appropriate next activity. When you want to call another task flow, you use a task flow call activity. If the flow requires some sort of branching, you use a router activity. At the end of a bounded task flow, you use a return activity which allows the flow to exit and control is sent back to the flow that called this bounded task flow.

For more detailed information and procedures regarding the individual activities of a task flow, including the metadata created for each activity and additional configuration that you can set, see Adding Activities to a Task Flow.

How to Create a Task Flow

The processes for creating bounded and unbounded task flows are similar. The main difference is that you select the **Create as Bounded Task Flow** checkbox in the Create Task Flow dialog to create a bounded task flow.

> **Note:**
>
> When you create a project, you may not need to create the unbounded task flow for it. If **ADF Page Flow** is specified as a selected technology on the Features page of the Project Properties dialog, the new `adfc-config.xml` file is automatically created within the project. The `adfc-config.xml` file is the main source file for the unbounded task flow.

Before you begin:

It may be helpful to have an understanding of what constitutes a task flow. For more information, see Creating a Task Flow.

You may also find it helpful to understand functionality that can be added using other task flow features. For more information, see Additional Functionality for ADF Task Flows.

To create a task flow:

1. In the Applications window, right-click the project in which you want to create the task flow and choose **New** > **ADF Task Flow**.
2. In the Create Task Flow dialog, the **Create as Bounded Task Flow** checkbox is selected by default. Deselect it if you want to create to create a source file that will be incorporated into the application's unbounded task flow.

   Deselecting the checkbox automatically changes the default value in the **File Name** field. This value will be used to name the XML source file for the task flow you create. The XML source file contains metadata describing the activities and control flow rules in the task flow.

   > **Tip:**
   >
   > The default name for the unbounded task flow is `adfc-config.xml`. The default name for the source file for a bounded task flow matches the value specified in the **Task Flow ID** field.

   Because a single project can contain multiple task flows, a number may be added to the default value in the **File Name** field in order to give the source file a unique name, for example, `task-flow-definition3.xml`.
3. In the Create Task Flow dialog, the **Create with Page Fragments** checkbox is selected by default.

   Deselect this checkbox if you want the view activities that you add to the task flow to reference JSF pages that render in the main browser window as the root page. Leave the **Create with Page Fragments** checkbox selected if you want the view activities that you add to the task flow to reference page fragments files (`.jsff`) that the task flow displays in an ADF region at runtime. Note that this distinction does not apply to dialogs that you will invoke from a view activity; you may define a view activity that references either a JSF page or a page fragment file when you select **Create with Page Fragments**.
4. Click **OK**.

A diagram representing the task flow appears in the editor.

> **Tip:**
>
> You can view a thumbnail of the entire task flow diagram by clicking the diagram and then choosing **Window** > **Thumbnail** from the main menu.

After you create the task flow, you can update it using the diagram, overview, and source editors. You can also use the Structure window to update the task flow.

> **Tip:**
>
> There are other ways to create task flows, for example, by refactoring the contents of an existing task flow into a new task flow. For more information, see Refactoring to Create New Task Flows and Task Flow Templates.

5. In the ADF Task Flow page of the Components window, from the Component panel, in the Activities group, drag and drop an activity onto the diagram.

   Normally, you would start with a view activity. For more detailed procedures for adding any type of activity, see How to Add an Activity to a Task Flow.

   - If you drag a view activity onto the diagram, you can double-click it to display the wizard for the JSF page or page fragment that the task flow is configured to invoke. Use the wizard to define characteristics for the page or page fragment. For more information, see Using View Activities.

     > **Note:**
     >
     > You can also add a view activity to a task flow by dragging a page from the Applications window and dropping it on the diagram for the task flow.

   - If you drag a router activity onto the diagram, you can use the Properties window to create an expression whose evaluation determines which control flow rule to follow. For more information, see Using Router Activities.
   - If you drag a method call activity onto the diagram, you can use the Properties window to configure the method to call. For more information, see Using Method Call Activities.
   - If you drag a task flow call activity onto the diagram, you can double-click it to display the Create Bounded Task Flow dialog where you can define settings for a new bounded task flow. For more information, see Using Task Flow Call Activities.
   - If you are creating a bounded task flow, and you drag a task flow return activity onto the diagram, you can use the Properties window to configure the activity. For more information, see Using Task Flow Return Activities .
6. Create control flow cases between the activities (for more information and detailed procedures, see How to Add a Control Flow Rule to a Task Flow):

   a. In the ADF Task Flow page of the Components window, from the Component panel, in the Control Flow group, choose **Control Flow Case**.
   b. On the diagram, click a source activity, for example a view, and then click the destination activity. For example in Figure 23-14, two activities have been linked with a control flow. Both the source (`view1`) and the destination (`view2`) activities are linked.
   c. In the Properties window, expand the **General** section, and choose the outcome value using either the **From Action** attribute (if a method determines the outcome) or the **From Outcome** attribute (if the outcome can be set as a `String`).

7. If you are creating a bounded task flow, you may want to designate one of the activities as the default activity. This makes sure that this specific activity executes first whenever the bounded task flow runs. By default, JDeveloper makes the first activity you add to the task flow the default. To change to a different activity, right-click the appropriate activity in the diagram and choose **Mark Activity > Default Activity**. For more information, see What You May Need to Know About the Default Activity in a Bounded Task Flow.

What Happens When You Create a Task Flow

A new XML source file is created every time you create a new unbounded or bounded task flow. By default, the XML source file for the unbounded task flow is called `adfc-config.xml`.

As shown in the following example, `<adfc-config>` appears first as the top-level element in all ADF Controller XML source files. Bounded task flows, activities and control flow rules are defined inside the `<adfc-config>` element. Bounded task flows are identified within the source file by the `<task-flow-definition>` metadata element.

```
<?xml version="1.0" encoding="windows-1252" ?>
<adfc-config xmlns="http://xmlns.oracle.com/adf/controller" version="1.2"
id="__1">
  <task-flow-definition id="task-flow-definition">
    <use-page-fragments/>
  </task-flow-definition>
</adfc-config>
```

A bounded task flow is identified by its task flow reference, which is comprised of a unique combination of identifier and document name. Example 23-1 shows a sample task flow reference within a task flow call activity.

> **Note:**
>
> If you use JDeveloper to create the bounded task flow, specify only one ID (indicating one bounded task flow) per document.

You assign both identifier and document name when you create the bounded task flow. As shown in Example 23-1, the identifier is the value in the **Task Flow ID** field. The document name is the value in the **File Name** field.

Example 23-1   Task Flow Reference

```
<adfc-config xmlns="http://xmlns.oracle.com/adf/Controller" version="1.2">
  <task-flow-definition id="task-flow-definition">
   <use-page-fragments/>
    ...
    <task-flow-call id="taskFlowCall">
      <task-flow-reference>
        <document>/WEB-INF/target-task-flow-definition.xml</document>
        <id>my-task-flow</id>
      </task-flow-reference>
    </task-flow-call>
    ...
  </task-flow-definition>
</adfc-config>
```

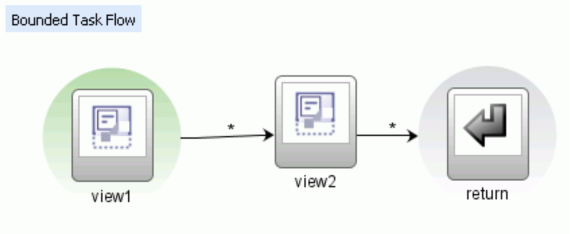
What You May Need to Know About the Default Activity in a Bounded Task Flow

The default activity is the first activity to execute in a bounded task flow. For example, the default activity always executes first when a task flow call activity passes control to the bounded task flow.

Unbounded task flows do not have default activities.

As shown in Figure 23-9, a green circle identifies the default activity in a task flow diagram.

Figure 23-9   Default Activity in a Bounded Task Flow



The first activity that you add to a new bounded task flow diagram is automatically identified as the default activity. You can also right-click any activity in the task flow diagram and choose **Mark Activity > Default Activity**. The default can be any activity type and it can be located anywhere in the control flow of the bounded task flow. To find the default activity, right-click anywhere on the task flow diagram and choose **Go to Default Activity**.

A bounded task flow can have only one default activity. If you mark a second activity as the default, the first is unmarked automatically. To unmark an activity manually, right-click the activity in the task flow diagram and choose **Unmark Activity > Default Activity**.

You should not specify a train stop in the middle of a train as a default activity. For more information, see Using Train Components in Bounded Task Flows.

Example 23-2 contains sample metadata for a default activity called `SurveyPrompt` in a bounded task flow:

Example 23-2 Default Activity Metadata in a Bounded Task Flow

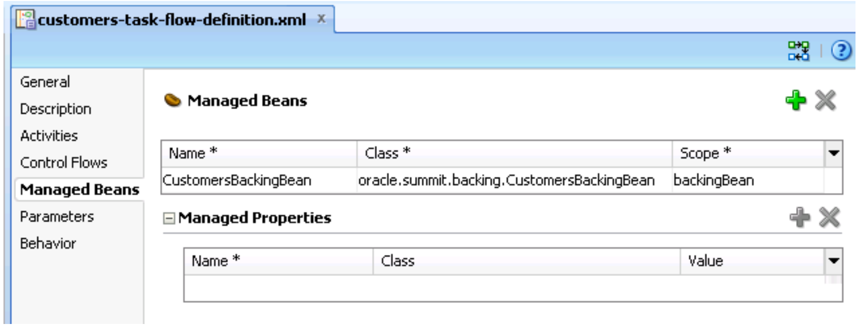
```
<task-flow-definition id="survey">
    <default-activity>SurveryPrompt</default-activity>
    <view id="SurveryPrompt">
      <page>/SurveryPrompt.jsff</page>
    </view>
    <use-page-fragments/>
</task-flow-definition>
```

What You May Need to Know About Memory Scope for Task Flows

Each task flow in your Fusion web application defines a pageFlow scope to manage state. The pageFlow scope begins when the task flow begins and ends when the task flow ends. A pageFlow scope defines a unique storage area for each instance of a task flow within an application which is used to pass data values between the activities in the task flow. When one task flow calls another, the calling task flow cannot access the called task flow's pageFlow scope. This means, for example, that a UI component on a page referenced by a task flow's view activity cannot access the pageFlow scope of another task flow even if this task flow is an ADF region embedded in the same page as the UI component.

You can define multiple managed beans with task flows. Figure 23-10 shows an example from the Summit ADF task flow sample application where a bounded task flow (`customers-task-flow-definition.xml`) references a managed bean with a backingBean scope. You can determine the scope assigned to a managed bean.

Figure 23-10 Managed Bean Registered With Task Flow



Name *	Class *	Scope *
CustomersBackingBean	oracle.summit.backing.CustomersBackingBean	backingBean

Name *	Class	Value

Table 23-4 lists available scopes for managed beans and describes when it is appropriate to use each scope in a managed bean that you define with a task flow. The table lists the scopes in order of their life span. For example, the application scope has a longer life span than the request scope. For more information about the life span of object scopes, see About Object Scope Lifecycles.

Table 23-4 Memory Scope for ADF Managed Beans

Scope	Description
application	The application scope lasts until the application stops. Values that you store in a managed bean with this scope are available to every session and every request that uses the application. Avoid using this scope in a task flow because it persists beyond the life span of the task flow.
session	The session scope begins when a user first accesses a page in the application and ends when the user's session times out due to inactivity, or when the application invalidates the session. Use this scope only for information that is relevant to the whole session, such as user or context information. Avoid using it to pass values from one task flow to another. Instead, use parameters to pass values between task flows. Using parameters gives your task flow a clear contract with other task flows that call it or are called by it. Another reason to avoid use of session scope is because it may persist beyond the life span of the task flow. Finally, if an end user opens multiple browser windows within the same session, session scoped values are shared across these browser windows. Conflicts can occur between these browser windows.
pageFlow	Choose this scope if you want the managed bean to be accessible across the activities within a task flow. A managed bean that has a pageFlow scope shares state with pages from the task flow that access it. A managed bean that has a pageFlow scope exists for the life span of the task flow.
view	Use this scope for managed bean objects that are needed only within the current view activity and not across view activities. The life span of this scope begins and ends when the current `viewId` of a view port changes. If you specify view, the application retains managed bean objects used on a page as long as the user continues to interact with the page. These objects are automatically released when the user leaves the page. Both JSF and Oracle ADF have an implementation of this scope. In a Fusion web application, when you use view scope in an EL expression, it resolves to the Oracle ADF implementation.
request	Use request scope when the managed bean does not need to persist longer than the current request.
backingBean	A backing bean is a convention to describe a managed bean that stores accessors for UI components and event handling code on a JSF page. It exists for the duration of a request and should not be used to maintain state. Use this scope if it is possible that your task flow appears in two ADF regions on the same JSF page and you want to isolate each instance of ADF region. In scenarios where you have more than one task flow, consider using a naming convention that assigns a unique name to each managed bean. Adopting this approach makes sure that you do not attempt to put two managed beans with the same name from different task flows into backingBean scope.

When you define a managed bean with a task flow, JDeveloper generates entries similar to the following in the task flow's source file:

```
<managed-bean id="__15">
    <managed-bean-name id="__16">egBackingBean</managed-bean-name>
    <managed-bean-class id="__13">oracle....egBackingBean</managed-bean-class>
    <managed-bean-scope id="__14">backingBean</managed-bean-scope>
</managed-bean>
```

The <managed-bean-scope> element holds the value for the scope of the managed bean (backingBean in the example).

When you bind a UI component to a managed bean, JDeveloper appends Scope to the scope name in the EL expression that it generates to reference the managed bean. For example, the binding attribute of a table component that references the managed bean has the following EL expression:

```
<af:table id="cartTab"
      ...
      binding="#{backingBeanScope.egBackingBean.table}"
     ...
</af:table>
```

Restrict the scope of the managed bean that you reference through a UI component's binding attribute to backingBean or request scope. Instances of UI components cannot be serialized. Objects in scopes other than backingBean and request are expected to be serializable. For this reason, you should not bind UI components to managed beans that have a scope other than backingBean or request. Note that JDeveloper defaults the binding attribute for UI components and region fragments to use the backingBean scope.

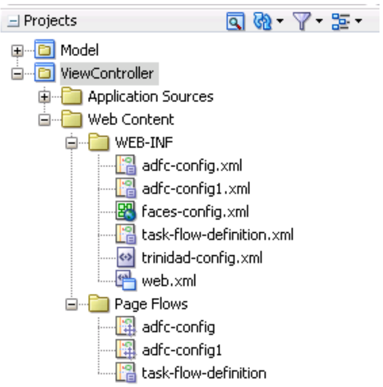
> **Note:**
>
> Write EL expressions that explicitly qualify the scope to access when writing EL expressions to access custom scopes unique to Oracle ADF (pageFlow, backingBean, and view scopes). For example, write an EL expression to access a pageFlow scope as follows:
>
> ```
> #{pageFlowScope.inpTxtBB.uiComponent}
> ```

What You May Need to Know About the Source Files for Task Flows

A single application can have multiple unbounded task flow XML source files and multiple bounded task flow XML source files. The set of files that combine to produce the unbounded task flow is referred to as the application's **ADF Controller bootstrap configuration files**. The unbounded task flow is assembled at runtime by combining one or more ADF Controller bootstrap configuration files. All activities within the bootstrap configuration files that are not contained within a bounded task flow are considered to be within the unbounded task flow.

The names of the source files within a single application must be different. The example in Figure 23-11 contains two unbounded task flows (adfc-config, adfc-config1) and a bounded task flow (task-flow-definition).

Figure 23-11 Application with Two Unbounded Task Flow XML Source Files



```
Projects
  Model
  ViewController
    Application Sources
    Web Content
      WEB-INF
        adfc-config.xml
        adfc-config1.xml
        faces-config.xml
        task-flow-definition.xml
        trinidad-config.xml
        web.xml
      Page Flows
        adfc-config
        adfc-config1
        task-flow-definition
```

Adding Activities to a Task Flow

You can modify any ADF task flow once created in the ADF application. You can add activities by dragging from the activities group from the Component panel and drop it onto the diagram.

After you create a task flow, you add activities to the task flow and configure the control flow between the activities. The Components window displays available activities and control flows. You drag and drop the activities and control flows from the Components window to the diagram for the task flow. You then configure control flow between the activities so that the task flow performs the task you want it to perform.

How to Add an Activity to a Task Flow

After you create a task flow, a task flow diagram and the Components window automatically display. The task flow diagram is a visual editor on which you can add the activities and control flows for the task flow. You can add them to the diagram by dragging them from the Components window.

Before you begin:

It may be helpful to have an understanding of what activities you can add to a task flow. For more information, see Adding Activities to a Task Flow.

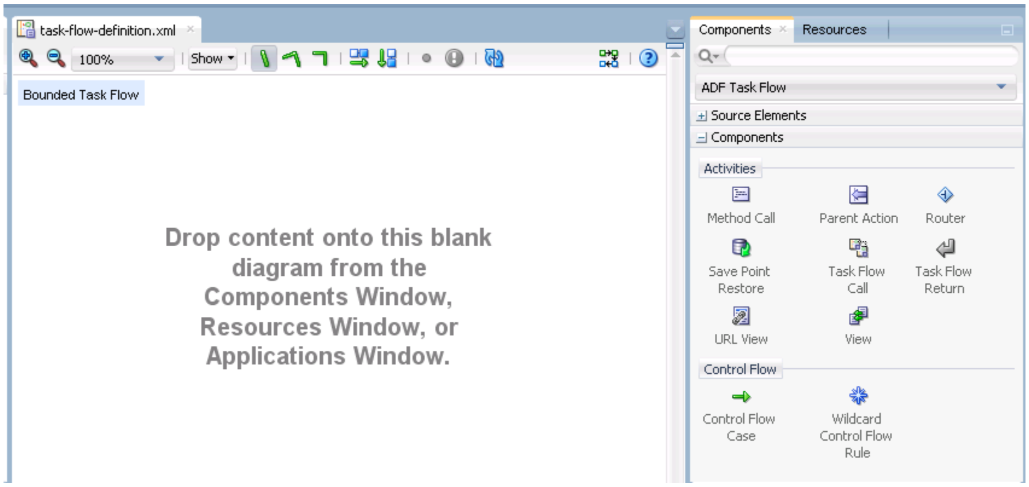
You may also find it helpful to understand functionality that can be added to task flows using other Oracle ADF features. For more information, see Additional Functionality for ADF Task Flows.

To add an activity to a task flow:

1. In the Applications window, expand the **WEB-INF** node and double-click the task flow to which you want to add an activity.

   For example, Figure 23-12 shows the editor and the Components window that automatically appears when you double-click a task flow source file (`task-flow-definition.xml`).

Figure 23-12   Task Flow Diagram



2. In the ADF Task Flow page of the Components window, from the Component panel, in the Activities group, drag and drop an activity onto the diagram.
   - If you drag a view activity onto the diagram, you can double-click it to display the wizard to create a JSF page or page fragment. For more information, see Using View Activities.
   - If you drag a task flow call activity onto the diagram, you can double-click it to display the Create Bounded Task Flow dialog where you can define settings for a new bounded task flow. For more information, see Using Task Flow Call Activities.

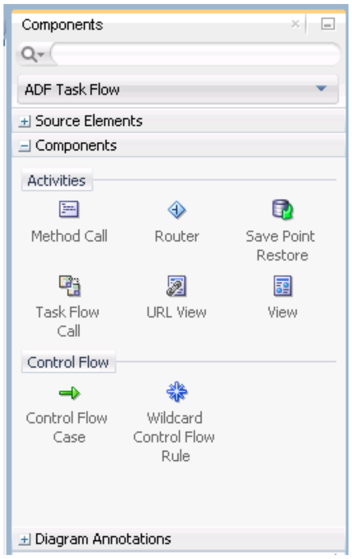
> **Tip:**
>
> Each activity you drag to the task flow diagram can display optional status icons and a tooltip that provides additional information about the activity. For example, after you drag a view activity to the task flow diagram, it may display a warning icon until you associate it with a JSF page.
>
> To turn on the icons, choose **Show** at the top of the task flow diagram, and then choose **Status** and one of the following:
>
> - **Error**: Displays when there is a problem in the task flow metadata which prevents it from running. For example, a view activity in the metadata can contain a `<bookmark>` or `<redirect>` element, but not both.
> - **Warning**: Displays when there is a problem in the task flow metadata that does not prevent the task flow from running.
> - **Incomplete**: Displays when an activity is incomplete. For example, a view activity that does not have a physical page associated with it or a task flow call that does not have a task flow reference associated with it are both considered incomplete activities. The resulting task flow metadata may prevent it from running.
>
> You can drag your mouse over a secondary icon to read a tooltip describing the icon's meaning.

What Happens When You Add an Activity to a Task Flow

As shown in Figure 23-13, the Components window contains separate sections for components and diagram annotations. The contents of the **Components** section differ slightly depending on whether you are creating a bounded or an unbounded task flow. For example, if you are creating a bounded task flow, the **Components** section contains an additional task flow return activity.

Figure 23-13 displays the activities you can add to an unbounded task flow.

Figure 23-13 Activities for an Unbounded Task Flow



Adding Control Flow Rules to Task Flows

ADF control flow rules define the path for the control to pass from one activity to another in a task flow. You can also define conditional navigation, value based navigation, or even merge multiple control flows to form a single control flow.

An ADF Controller control flow rule defines how control passes from one activity to another in a task flow. A control flow rule can contain one or more control flow cases. A control flow case identifies the activity to which control flow passes. A control case also has options that allow you to configure conditional navigation (using the `<if>` element) and/or limit navigation based on the value of the action from where the control flow originates (using the `from-action` element).

An ADF Controller control flow rules are based on JSF navigation rules, but capture additional information. JSF navigation is always between pages, whereas control flow rules describe transitions between activities. For example, a control flow rule can indicate a transition between a view activity and a subsequent method call activity. Or, it can indicate that control passes from the page (view activity) to another task flow.

The following task flow activities cannot be the source of a control flow rule:

- Save Point Restore

- Task Flow Return
- URL View

The basic structure of a control flow rule mimics a JSF navigation rule. Table 23-5 describes how metadata maps from JSF navigation rules to ADF Controller control flow rules.

Table 23-5 Mapping of JSF Navigation Rules to ADF Controller Control Flow Rules

JSF Navigation Rule	ADF Controller Control Flow Rule
Navigation Rule	Control Flow Rule
From View ID	From Activity ID
Navigation Case	Control Flow Case
From Action	From Action
From Outcome	From Outcome
If	If
To View ID	To Activity ID

When using ADF task flows, perform all application navigation using ADF Controller control flow rules instead of using JSF navigation rules in the `faces-config.xml` file. ADF Controller delegates navigation handling when it does not find a matching control flow case in its metadata. However, not all ADF Controller functionality is guaranteed to work correctly if navigation is performed by a non-ADF Controller navigation handler. For more information about how ADF Controller evaluates control flow rules, see What Happens at Runtime: How Task Flows Evaluate Control Flow Rules.

Use the task flow diagram as a starting point for creating basic control flows between activities. Later, you can edit control flow properties in the Structure window, Properties window or overview editor for the task flow diagram.

> **Tip:**
>
> You can drag and drop an activity onto an existing control flow. This splits the existing control flow into two, with the activity in the center.

How to Add a Control Flow Rule to a Task Flow

You create a control flow rule by dragging a Control Flow Case from the ADF Task Flow page of the Components window and dropping it on a source task flow activity and a target task flow activity.

Before you begin:

It may be helpful to have an understanding of what a control flow rule is. For more information, see Adding Control Flow Rules to Task Flows.

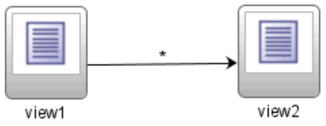
You may also find it helpful to understand functionality that can be added using other task flow features. For more information, see Additional Functionality for ADF Task Flows.

To add a control flow rule to a task flow:

1. In the Applications window, expand the **WEB-INF** node and double-click the task flow where you want to add a control flow rule.
2. In the ADF Task Flow page of the Components window, from the Component Panel, in the Control Flow group, select **Control Flow Case**.
3. On the diagram, click a source task flow activity and then click the destination task flow activity.

   In many cases, the source and destination task flow activities will be view activities. In Figure 23-14, two such task flow activities (`view1`) and (`view2`) have been linked with a control flow case.

   Figure 23-14 Control Flow Case

   JDeveloper adds the control flow case to the diagram. Each line that JDeveloper adds between task flow activities represents a control flow case. The arrow indicates the direction of the control flow case. The **From Outcome** element contains a value that can be matched against values specified in the `action` attribute of UI components.
4. To change the value of **From Outcome**, choose the text next to the control flow in the diagram and overwrite the default wildcard `*` character.

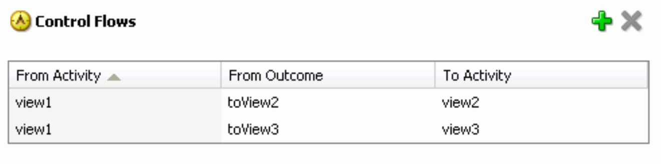
   A useful convention is to cast the control flow rule in the form `toDestination` or some other form that describes the purpose of the control flow rule. To define the control flow shown in Figure 23-14, for instance, you would overwrite the wildcard with `toView2`.
5. To change the values of **From Activity ID** (which identifies the source activity) or **To Activity ID** (which identifies the target activity), drag either end of the arrow in the diagram to a new activity.

Tip:

After you choose the control flow in the task flow diagram, you can also change its properties in the Properties window or the Structure window. The Structure window is helpful for displaying the relationship between control rules and cases.

You can also click **Control Flows** on the overview editor for the task flow diagram to add cases, as shown in Figure 23-15. To add a case, make sure that the **From Activity** (source activity) and the **To Activity** (target activity) for the rule have already been added to the task flow.

Figure 23-15 Control Flows on Overview Editor for the Task Flow



Control Flows

From Activity	From Outcome	To Activity
view1	toView2	view2
view1	toView3	view3

6. Optionally, in the Properties window expand the **Behavior** section, and write an EL expression in the **If** field that must evaluate to `true` before control can pass to the activity identified by **To Activity ID**.

How to Add a Wildcard Control Flow Rule

You can add a wildcard control flow rule to an unbounded or bounded task flow. The steps for adding it are similar to those for adding any activity to a task flow diagram.

Before you begin:

It may be helpful to have an understanding of the control flow rule options available to you. For more information, see Adding Control Flow Rules to Task Flows.

You may also find it helpful to understand functionality that can be added using other task flow features. For more information, see Additional Functionality for ADF Task Flows.

To add a wildcard control flow rule:

1. In the Applications window, expand the **WEB-INF** node and double-click the task flow where you want to add a wildcard control flow rule.
2. In the ADF Task Flow page of the Components window, from the Component panel, in the Control Flow group, drag and drop a **Wildcard Control Flow Rule** onto the diagram.
3. In the ADF Task Flow page of the Components window, from the Component panel, in the Control Flow group, select **Control Flow Case**.
4. In the task flow diagram, drag the control flow case from the wildcard control flow rule to the target activity.

   The target can be any activity type.
5. By default, the label below the wildcard control flow rule is *. This is the value for the **From Activity ID** element. To change this value, select the wildcard control

flow rule in the diagram. In the Properties window for the wildcard control flow rule, enter a new value in the **From Activity ID** field. A useful convention is to cast the wildcard control flow rule in a form that describes its purpose. For example, enter `project*`. The wildcard must be a trailing character in the new label.

> **Tip:**
>
> You can also change the **From Activity ID** value in the overview editor for the task flow diagram.

6. Optionally, in the Properties window expand the **Behavior** section, and write an EL expression in the **If** field that must evaluate to `true` before control can pass to the activity identified by **To Activity ID**.

What Happens When You Create a Control Flow Rule

Understanding the elements that define the rules in the source file for the task flow helps when creating control flow rules directly in the task flow diagram, task flow overview editor, or Structure window, or when adding them directly in the XML source file. Example 23-3 shows the general syntax of a control flow rule element in the task flow source file.

Control flow rules can consist of the following metadata:

- `control-flow-rule`: A mandatory wrapper element for control flow case elements.
- `from-activity-id`: The identifier of the activity where the control flow rule originates, for example, `source`.

  A trailing wildcard (*) character in `from-activity-id` is supported. The rule applies to all activities that match the wildcard pattern. For example, `login*` matches any logical activity ID name beginning with the literal `login`. If you specify a single wildcard character in the metadata (not a trailing wildcard), the control flow automatically converts to a wildcard control flow rule activity in the diagram. For more information, see How to Add a Wildcard Control Flow Rule .
- `control-flow-case`: A mandatory wrapper element for each case in the control flow rule. Each case defines a different control flow for the same source activity. A control flow rule must have at least one control flow case.
- `from-action`: An optional element that limits the application of the rule to outcomes from the specified action method. The action method is specified as an EL binding expression, such as `#{backing_bean.cancelButton_action}`.

  In Example 23-3, control passes to `destinationActivity` only if `outcome` is returned from `actionmethod`.

  The value in `from-action` applies only to a control flow originating from a view activity, not from any other activity types. Wildcards are not supported in `from-action`.
- `from-outcome`: Identifies a control flow case that will be followed based on a specific originating activity outcome. All possible originating activity outcomes should be accommodated with control flow cases.

  If you leave both the `from-action` and the `from-outcome` elements empty, the case applies to all outcomes not identified in any other control flow cases defined for the activity, thus creating a default case for the activity. Wildcards are not supported in from-outcome.

- to-activity-id: A mandatory element that contains the complete identifier of the activity to which the navigation is routed if the control flow case is performed. Each control flow case can specify a different to-activity-id.
- if: An optional element that accepts an EL expression as a value. If the EL expression evaluates to true at runtime, control flow passes to the activity identified by the to-activity-id element.

Example 23-3 Control Flow Rule Syntax in the Source File

```
<control-flow-rule>
    <from-activity-id>from-view-activity</from-activity-id>
  <control-flow-case>
    <from-action>actionmethod</from-action>
    <from-outcome>outcome</from-outcome>
    <to-activity-id>destinationActivity</to-activity-id>
    <if>#{myBean.someCondition}</if>
  </control-flow-case>
  <control-flow-case>
   ....
  </control_flow-case>
</control-flow-rule>
```

What Happens at Runtime: How Task Flows Evaluate Control Flow Rules

At runtime, task flows evaluate control flow rules from the most specific to the least specific match to determine the next transition between activities. Evaluation is based on the following priority, which is similar to that for JSF navigation rules:

1. from-activity-id, from-action, from-outcome
2. from-activity-id, from-outcome
3. from-activity-id

ADF Controller first searches for a match in all three elements: from-activity-id, from-action, and from-outcome. If there is no match, ADF Controller searches for a match in just the from-activity-id and from-outcome elements. Finally, ADF Controller searches for a match in the from-activity-id element alone.

If ADF Controller cannot find a control flow rule within its metadata to match a request, it allows the standard JSF navigation handler to find a match.

The unbounded task flow can have more than one ADF Controller XML source file. Because control flow rules can be defined in more than one ADF Controller XML source file, similar rules may be defined in different files. If there is a conflict in which two or more cases have the same from-activity-id, and the same from-action or from-outcome values, the last case (as listed in the adfc-config.xml, bootstrap, or bounded task flow source file) is used. If the conflict is among rules defined in different source files, the rule in the last source file to be loaded is used.

ADF Controller also implements the following interface with a number of restrictions:

```
javax.faces.application.ConfigurableNavigationHandler
```

The restrictions are:

- The Map object returned by getNavigationCases() is not modifiable. Any runtime changes to control flow rules must be made with the customization features provided with the MDS framework. For more information, see Customizing Applications with MDS .
- Do not invoke the performNavigation() method after the JSF Invoke Application phase. This is to make sure that the view ID does not change between the ADF Prepare Render phase and the JSF Render Response phase. For more information about how the JSF and ADF phases integrate in the lifecycle of a page request, see Understanding the Fusion Page Lifecycle .
- The metadata values for a task flow view activity's Bookmark and Redirect properties populate the corresponding information for the navigation case objects returned by the getNavigationCases().

Testing Task Flows

ADF task flows, as in any other application, can be tested to debug any issues or test cases to either fix the issue or to improve its performance. The task flow can be either tested and then deployed to the WebLogic Server or the test flow can be tested while the execution is in progress on the WebLogic Server without interrupting the execution. The running and debugging process is different for different types of task flows.

The procedure for running and debugging task flows differs depending on whether the task flow is bounded or unbounded, whether it contains pages or page fragments, or whether it accepts input parameters.

JDeveloper and Oracle ADF provides a number of features that facilitate your testing of the task flows that you create. Among these is the ability to set ADF declarative breakpoints on task flow activities. See How to Set and Use Task Flow Activity Breakpoints. In addition, you can also modify a task flow's metadata for an application that is currently executing in the Integrated WebLogic Server. The changes that you make to the task flow can be deployed to the application without stopping the application and redeploying it to the Integrated WebLogic Server (also known as **hot reloading**). To do this, you invoke the make command on the modified task flow (right-click the task flow and choose **Make**). This deploys the modified task flow to the application executing in the Integrated WebLogic Server. For information about hot reloading, see Reloading Oracle ADF Metadata in Integrated WebLogic Server.

The Create Default Domain dialog appears the first time you run your application and start a new domain in Integrated WebLogic Server. Use the dialog to define an administrator password for the new domain. Passwords you enter can be eight characters or more and must have a numeric character.

How to Run a Bounded Task Flow That Contains Pages

You can run or debug a bounded task flow that contains view activities that are pages.

For information on running a bounded task flow that contains view activities that are page fragments, see How to Run a Bounded Task Flow That Uses Page Fragments .

Before you begin:

It may be helpful to have an understanding of the factors that affect how you test a task flow. For more information, see Testing Task Flows.

You may also find it helpful to understand functionality that can be added using other task flow features. For more information, see Additional Functionality for ADF Task Flows.

To run or debug a bounded task flow that uses pages:

- In the task flow diagram, right-click the task flow and choose either **Run** or **Debug**.
- You can also run the task flow directly by entering its URL in the browser. For example:

  ```
  http://somecompany.com/internalApp/MyApp/faces/
  adf.task-flow?adf.tfId=displayHelp&adf.tfDoc=%2FWEB-
  INF%2Fdisplayhelp.xml&topic=createPurchaseOrder
  ```

  For more information, see What You May Need to Know About Calling a Bounded Task Flow Using a URL.
- In the Applications window, expand the **WEB-INF** node, right-click the bounded task flow and choose either **Run** or **Debug**.

How to Run a Bounded Task Flow That Uses Page Fragments

Bounded task flows that use page fragments are intended to run only within an ADF region. A **page fragment** is a JSF document that renders as content in another JSF page. For more information, see About Page Fragments and ADF Regions.

Before you begin:

It may help to understand what factors affect how you test a task flow. For more information, see Testing Task Flows.

You may also find it helpful to understand functionality that can be added using other task flow features. For more information, see Additional Functionality for ADF Task Flows.

You will need to complete these tasks:

1. Create a JSF page containing a region that binds to the bounded task flow. JDeveloper automatically creates the region for you in a JSF page when you drop a bounded task flow containing page fragments on a page. For more information about creating regions, see Creating an ADF Region.
2. Create a view activity in the project's unbounded task flow that refers to the page. For more information, see How to Add an Activity to a Task Flow.

To run or debug a bounded task flow that uses page fragments:

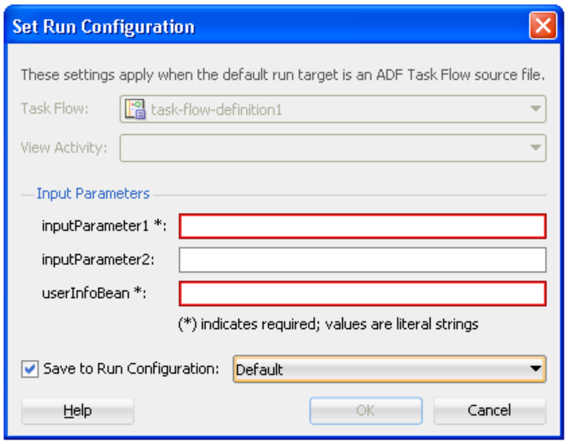
- Right-click the view activity in the Applications window or in the task flow diagram and choose **Run**.

How to Run a Bounded Task Flow That Has Parameters

Before you run a bounded task flow with parameters, you must first run a bounded task flow containing pages. For more information, see How to Run a Bounded Task Flow That Uses Page Fragments .

When you attempt to run or debug a bounded task flow with input parameters defined, the Set Run Configuration dialog appears, as shown in Figure 23-16. You enter values that you want to pass to the bounded task flow in this dialog and click **OK**. For more information about how to define input parameters for a bounded task flow, see Using Parameters in Task Flows.

Figure 23-16 Set Run Configuration dialog



Set Run Configuration

These settings apply when the default run target is an ADF Task Flow source file.

Task Flow: task-flow-definition1

View Activity:

Input Parameters

inputParameter1 *:

inputParameter2:

userInfoBean *:

(*) indicates required; values are literal strings

☑ Save to Run Configuration: Default

Before you begin:

It may be helpful to have an understanding of the factors that affect how you test a task flow. For more information, see Testing Task Flows.

You may also find it helpful to understand functionality that can be added using other task flow features. For more information, see Additional Functionality for ADF Task Flows.

To run or debug a bounded task flow that has input parameter definitions:

1. In the diagram for the bounded task flow, right-click the task flow and choose either **Run** or **Debug**.
2. In the **Input Parameters** list of the Set Run Configuration dialog, enter values that you want to be passed as input parameters to the task flow. If you do not specify a value, the input parameter is not used when calling the bounded task flow.

   Each required input parameter in the list appears with an asterisk, as shown in Figure 23-16. You must specify the parameter value as a literal string. You cannot specify an EL expression.
3. Click **OK**.

How to Run a JSF Page When Testing a Task Flow

You can run a JSF page by right-clicking the page in the Applications window and choosing **Run**. However, if the page contains navigation UI components, such as a button or link, navigation is not guaranteed to work.

Before you begin:

It may be helpful to have an understanding of the factors that affect how you test a task flow. For more information, see Testing Task Flows.

You may also find it helpful to understand functionality that can be added using other task flow features. For more information, see Additional Functionality for ADF Task Flows.

You will need to complete these tasks:

1. Create a bounded or unbounded task flow.

   For more information, see How to Create a Task Flow.

2. Add a view activity to the task flow.

   For more information, see How to Add an Activity to a Task Flow.

To run a JSF Page with fully functioning navigation:

1. In the Applications window, drag the JSF page you want to run and drop it onto the view activity in the task flow diagram.

   This associates the view activity with the JSF page.

2. In the diagram, right-click the view activity and choose **Run**.

How to Run an Unbounded Task Flow

To run or debug an unbounded task flow, you choose a specific view activity with which to start the unbounded task flow.

Before you begin:

It may be helpful to have an understanding of the factors that affect how you test a task flow. For more information, see Testing Task Flows.

You may also find it helpful to understand functionality that can be added using other task flow features. For more information, see Additional Functionality for ADF Task Flows.

To run a view activity in the unbounded task flow:

- In the task flow diagram, right-click the view activity and choose either **Run** or **Debug**.

  The unbounded task flow runs beginning with the selected view activity.

- If you have selected something other than a single view activity (or have nothing selected), you are prompted to choose one in the Set Run Configuration dialog.

How to Set a Run Configuration for a Project

A **run configuration** contains settings that determine how projects run, such as specifying the first activity to run in a task flow. You can define one or more run configurations for a project. Within a run configuration, you can designate an ADF Controller source file as the default run target. When you run the project, the source file is the first to run.

Before you begin:

It may be helpful to have an understanding of the factors that affect how you test a task flow. For more information, see Testing Task Flows.

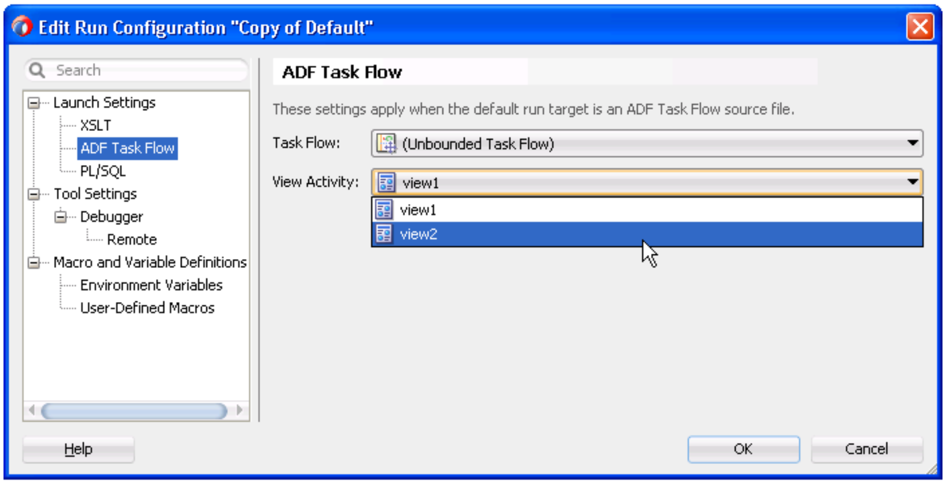
You may also find it helpful to understand functionality that can be added using other task flow features. For more information, see Additional Functionality for ADF Task Flows.

To define a default task flow run target:

1. In the Applications window, right-click the project where you want to define a default task flow run target and choose **Project Properties**.
2. In the Project Properties dialog, select **Run/Debug** and click **New** in the Run Configurations list.
3. In the Create Run Configuration dialog, enter the name for the new run configuration.
4. If you want to base the new configuration on an existing one, choose a configuration in the **Copy Settings From** dropdown list.
5. Click **OK** to exit the Create Run Configuration dialog.
6. In the Run Configurations list, click **Edit**.
7. In the **Default Run Target** field in the Edit Run Configuration dialog, specify a source file for the task flow that should run first when you run the project.

   Once you choose a task flow, you can set a view activity (for the unbounded task flow) or input parameters (for bounded task flows).
8. In the left panel of the Edit Run Configuration dialog, click **ADF Task Flow**.
9. In the **Task Flow** dropdown list, located on the right panel, choose the task flow containing the run target.
10. If you are running the unbounded task flow, the Edit Run Configuration dialog displays the **View Activity** dropdown list where you choose the view activity that runs first in the application, as shown in Figure 23-17.

Figure 23-17 Edit Run Configuration Dialog



Edit Run Configuration "Copy of Default"

ADF Task Flow

These settings apply when the default run target is an ADF Task Flow source file.

Task Flow: (Unbounded Task Flow)

View Activity: view1

11. Click **OK**.

    The next time you run the project, the saved run configuration will be available in the **Run** > **Choose Active Run Configuration** menu.

    If you are running a bounded task flow that has been set up to accept input parameters, a dialog displays a section for specifying values for all input parameters defined for the bounded task flow. For more information, see How to Run a Bounded Task Flow That Has Parameters .

What Happens at Runtime: Testing Task Flows

The behavior that you observe when you test a task flow may depend on a number of factors. For example, if you attempt to run or debug a bounded task flow with input parameters defined, the Set Run Configuration dialog appears. Also, if it is the first time that you attempt to test an application in this installation of JDeveloper, the Create Default Domain dialog appears so that you can start a new domain in Integrated WebLogic Server.

Refactoring to Create New Task Flows and Task Flow Templates

ADF task flows allow you to create task flows and task flow templates by reusing the existing activities, JSF page flows, and JSF pages. The new bounded task flow that you extract displays in the task flow diagram.

You can convert existing activities, JSF page flows, and JSF pages into new ADF Controller components such as bounded task flows and task flow templates.

How to Extract a Bounded Task Flow from an Existing Task Flow

You can create a new bounded task flow based on activities you choose in an existing bounded or unbounded task flow.

Before you begin:

It may be helpful to have an understanding of the factors that affect how you refactor a task flow. For more information, see Refactoring to Create New Task Flows and Task Flow Templates.

You may also find it helpful to understand functionality that can be added using other task flow features. For more information, see Additional Functionality for ADF Task Flows.

To extract a bounded task flow from an existing task flow:

1. In the Applications window, expand the **WEB-INF** node and double-click the unbounded or bounded task flow containing the activities you want to extract to a new bounded task flow.
2. In the task flow diagram, select one or more activities.

   > **Tip:**
   >
   > To select multiple activities in a diagram, click the left mouse button and drag the cursor over the activities.
   >
   > You can also press the Ctrl key while selecting each activity.

3. Right-click your selection and choose **Extract Task Flow**.

4. In the Extract Bounded Task Flow dialog, enter a file name for the new bounded task flow and enter the name of the directory that stores it.

   For more information about creating a bounded task flow, see Creating a Task Flow.

What Happens When You Extract a Bounded Task from an Existing Task Flow

The new bounded task flow that you extract displays in the task flow diagram. Table 23-6 describes the properties that JDeveloper automatically sets for the new bounded task flow.

Table 23-6 Properties Updated in the New Bounded Task Flow

Property	Value
Task flow definition ID	Value you entered in the **File Name** field in the Extract Bounded Task Flow dialog.
Default activity	Determined as the destination of all incoming control flow cases. If more than one destination exists, an error is flagged and the entire operation is rolled back.
Control flow rules	Control flow cases with selected source activities are included in the new bounded task. A **source activity** is an activity from which a control flow leads. The new bounded task flow includes the following types of control flow cases: • Both the source and target activities in the control flow case were selected to create the new task flow. • Only the source activity was selected to create the new task flow. Destinations are changed to the corresponding new task flow return activities added for each outcome.

The following changes automatically occur in the originating task flow (the task flow containing the activities you selected as the basis for the new task flow):

- A new task flow call activity is added to the originating task flow. The task flow call activity calls the new bounded task flow.
- The selected activities are removed from the originating task flow.
- Existing control flow cases associated with the activities you selected are removed from the originating task flow. They are replaced with new control flow cases:
  - An incoming control flow case to the old activity is redirected to the new task flow call activity.
  - An outgoing control flow case from the old activity is redirected from the new task flow call activity.

How to Create a Task Flow from JSF Pages

You can create a new bounded task flow based on selected pages in a JSF page flow. JDeveloper converts the JSF pages that are part of a flow (that is, those that are linked by JSF navigation cases) to view activities in the new task flow.

Before you begin:

It may be helpful to have an understanding of the factors that affect how you refactor a task flow. For more information, see Refactoring to Create New Task Flows and Task Flow Templates.

You may also find it helpful to understand functionality that can be added using other task flow features. For more information, see Additional Functionality for ADF Task Flows.

To create a new task flow from selected JSF pages in a page flow:

1. In the Applications window, double-click the page flow containing the pages you want to use in the new bounded task flow.
2. In the task flow diagram, select one or more JSF pages.

   > **Tip:**
   >
   > To select multiple elements in a diagram, click the left mouse button and drag the cursor over the elements.
   >
   > You can also press the Ctrl key while selecting each element.

3. Right-click your selection and choose **Generate ADF Task Flow**.

   The Create Task Flow dialog appears, which allows you to create a new unbounded or bounded task flow.
4. In the Create Task Flow dialog, enter a file name and directory for the task flow. Clear the **Create as Bounded Task** checkbox if you want to create an unbounded task flow.

   For more information, see Creating a Task Flow.
5. Click **OK**.

How to Convert Bounded Task Flows

You can convert an existing bounded task flow to an unbounded task flow or change whether the views it contains are pages or page fragments. Table 23-7 describes the results of each conversion.

Table 23-7 Converting Bounded Task Flows

Conversion	Result
Bounded task flow to unbounded task flow	Loses all metadata not valid for unbounded task flows, such as parameter definitions and transactions.
Bounded task flow to use JSF pages	Converts page fragments associated with any view activities in the task flow to JSF pages. Old page fragments are saved if you select the **Keep Page Fragment** checkbox. New JSF page names default to the name of the old page fragment.
Bounded task flow to use page fragments	Converts all pages associated with view activities in the bounded task flow to page fragments. Old pages are saved if you select the **Keep Page** checkbox. New page fragment names default to the name of the old page

Before you begin:

It may be helpful to have an understanding of the factors that affect how you refactor a task flow. For more information, see Refactoring to Create New Task Flows and Task Flow Templates.

You may also find it helpful to understand functionality that can be added using other task flow features. For more information, see Additional Functionality for ADF Task Flows.

To convert a bounded task flow:

1. In the Applications window, expand the **WEB-INF** node and double-click the bounded task flow that you want to convert.
2. In the task flow diagram, right-click anywhere other than on an activity or a control flow.
3. Choose the appropriate menu option:
   - **Convert to Unbounded Task Flow**
   - **Convert to Task Flow with Page Fragments**
   - **Convert to Task Flow with Pages**

24

Working with Task Flow Activities

This chapter describes how to use activities in ADF task flows that you create in your Fusion web application. The chapter contains detailed information about each task flow activity type that displays in the Components window in addition to describing the properties that you can configure for each task flow activity type.
This chapter includes the following sections:

About Task Flow Activities

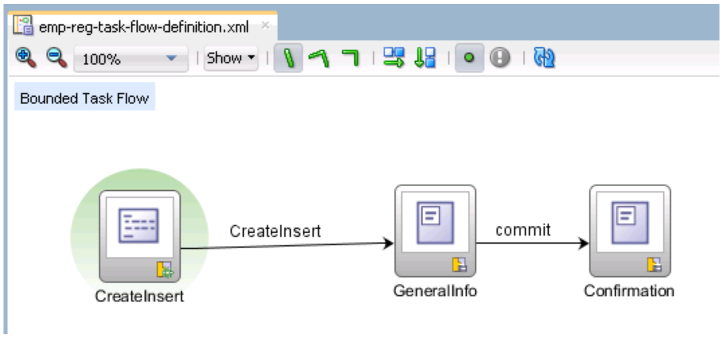
The ADF task flow activity is a module or independent set of tasks to be executed to complete a task. You can add activities to both bounded and unbounded task flows; however, certain activities can only be added to a bounded task flow.

An **activity** represents a piece of work that is performed when the task flow runs. It displays in the task flow's diagram editor as a node. You can add most activities to both bounded and unbounded task flows, although some activity types can be added only to bounded task flows.

Figure 24-1 shows the `emp-reg-task-flow-definition.xml` bounded task flow from the Summit sample application for ADF task flows. This task flow contains the following activities:

1. A method call activity that invokes a `CreateInsert` action
2. A view activity that renders a page fragment where end users can enter general information about the employee to be created
3. A view activity that renders a page fragment where a confirmation message appears if the end user successfully creates the new employee

Figure 24-1   Bounded Task Flow with Task Flow Activities



A task flow consists of activities and control flow cases that define the transitions between activities. Table 24-1 describes the types of activities and control flows you can add to a task flow by dragging and dropping from the Components window.

Table 24-1   Task Flow Activities and Control Flows

Icon	Component Name	Description
	Method Call	Invokes a method, typically a method on a managed bean. A method call activity can be placed anywhere within an application's control flow to invoke application logic based on control flow rules. See Using Method Call Activities.
	Parent Action	Allows a bounded task flow to generate outcomes that are passed to its parent view activity. Typically used for a bounded task flow running in an ADF region that needs to trigger navigation of its parent view activity. See Navigating Outside an ADF Region's Task Flow.
	Router	Evaluates an EL expression and returns an outcome based on the value of the expression. For example, a router in a credit check task flow might evaluate the return value from a previous method call and generate success, failure, or retry outcomes based on various cases. These outcomes can then be used to route control to other activities in the task flow. See Using Router Activities.
	Save Point Restore	Restores a previous persistent save point, including application state and data, in an application supporting save for later functionality. A save point captures a snapshot of the Fusion web application at a specific instance. Save point restore enables the application to restore whatever was captured when the save point was originally created. See Using Save Points in Task Flows.

Table 24-1 (Cont.) Task Flow Activities and Control Flows

Icon	Component Name	Description
	Task Flow Call	Calls a bounded task flow from the unbounded task flow or another bounded task flow. See Using Task Flow Call Activities.
↵	Task Flow Return	Identifies when a bounded task flow completes and sends control flow back to the caller. (Available for bounded task flows only). See Using Task Flow Return Activities .
	URL View	Redirects the root view port (for example, a browser page) to any URL-addressable resource, even from within the context of an ADF region. For more information see Using URL View Activities.
	View	Displays a JSF page or page fragment. Multiple view activities can represent the same page or same page fragment. See Using View Activities for more information. For more information about pages and page fragments, see Creating a Web Page.
➡	Control Flow Case	Identifies how control passes from one activity to the next in the application. See About Control Flows.
	Wildcard Control Flow Rule	Represents a control flow case that can originate from any activities whose IDs match a wildcard expression. For example, it can represent a control case `from-activity-id` containing a trailing wildcard such as `foo*`. See How to Add a Wildcard Control Flow Rule .

Table 24-2 describes the annotations (notes and attachments) you can add to a task flow.

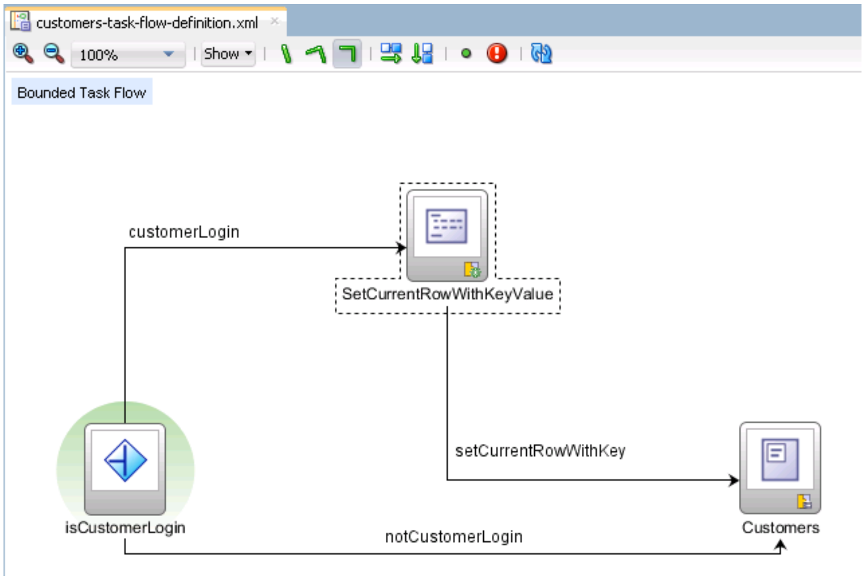
Table 24-2   Task Flow Diagram Annotations

Icon	Icon Name	Description
🟨	Note	Adds a note to the task flow diagram. You can choose the note in the diagram to add or edit text.
	Note Attachment	Attaches an existing note to an activity or a control flow case in the diagram.

Task Flow Activities Use Cases and Examples

Figure 24-2 shows the diagram for the customer task flow in the Summit ADF task flow sample application. This task flow uses a number of the activities that a bounded task flow supports in order to complete a task. For more information about the Summit ADF sample applications, see Introduction to the ADF Sample Application.

Figure 24-2   Customers Task Flow in the Summit ADF Task Flow Sample Application



Additional Functionality for Task Flow Activities

You may find it helpful to understand other **Oracle ADF** features before you configure or use task flow activities. Additionally, you may want to read about what you can do with your configured task flows. Following are links to other functionality that may be of interest.

- Task flows can invoke managed beans. For more information about managed beans, see Using a Managed Bean in a Fusion Web Application.
- Bounded task flows can be secured by defining the privileges that are required for someone to use them. For more information, see Enabling ADF Security in a Fusion Web Application.
- Bounded task flows can be packaged in ADF Library JARs. For more information, see Packaging a Reusable ADF Component into an ADF Library.
- Task flow activities can be associated with page definition files. For more information about page definition files, see Working with Page Definition Files.

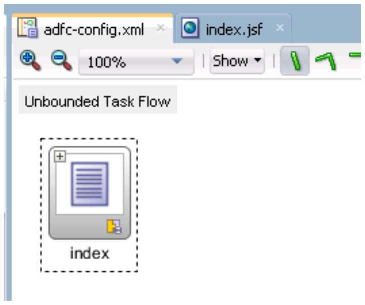
Using View Activities

The ADF task view activity allows you to display a JSF page or a page fragment. At runtime, you can transfer the control from one task flow to another. Users can bookmark a view activity within an unbounded task flow. Also, you can allow users to visit a URL by configuring a view activity.

One of the primary types of task flow activity is the view activity. A view activity displays a JSF page or page fragment. A **page fragment** is a JSF document that renders as content in another JSF page. Page fragments are typically used in bounded task flows that can be added to a JSF page as a region, as described in Creating an ADF Region.

Figure 24-3 shows the `index` view activity in the Summit ADF task flow sample application.

Figure 24-3 View Activity



> **Tip:**
>
> Click the + icon in the upper-left part of the view activity to see a thumbnail preview of the referenced page or page fragment.

XML metadata in the source file of the task flow associates a view activity with a physical JSF page or page fragment. An `id` attribute identifies the view activity.

The <page> element identifies the page or page fragment's file name. The following example shows the metadata that corresponds to the view activity in Figure 24-3:

```
<view id="index">
   <page>/index.jsf</page>
 </view>
```

The view activity ID and page name do not have to be the same.

The steps for adding a view activity to a task flow are the same as those for adding any activity to a task flow diagram. For more information, see How to Add an Activity to a Task Flow. After you add the view activity, you can double-click it to display a wizard that allows you to create a new page or page fragment. The wizard allows you to choose between one of the following two document types for the JSF page or page fragments that you create:

- Facelets

  The file extension for pages in this document type is .jsf and .jsff for page fragments.
- JSP XML

  The file extension for pages in this document type is .jspx and .jsff for page fragments.

Use one document type only in your task flows. For example, do not use the Facelets document type for view activities in one task flow and the JSP XML document type for view activities in another task flow. For more information about the document types, see Implementing the User Interface with JSF.

You also use the wizard to choose the page layout for the page or page fragment that you create. For more information about page layouts, see the "Using Quick Start Layouts" section in *Developing Web User Interfaces with Oracle ADF Faces*. In addition, you can specify whether or not to automatically expose UI components on the page or page fragment in a managed bean. For more information, see Using a Managed Bean in a Fusion Web Application.

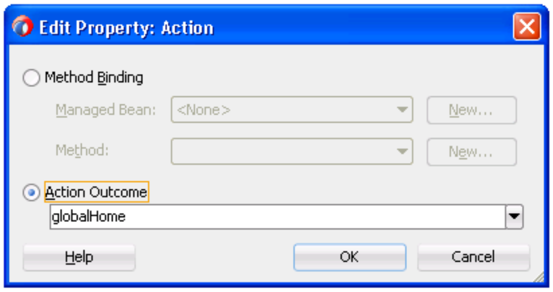
After you complete the wizard, JDeveloper automatically associates the completed page or page fragment with the view activity. As an alternative to using the wizard, you can associate a page or page fragment with a view activity by dragging an existing page or page fragment from the Applications window and dropping it on top of an existing view activity. If no view activity exists, drag a page or page fragment to any other location on the diagram. When you drop the page or page fragment, JDeveloper automatically creates a new view activity associated with the page or page fragment. During creation, a default id for the view activity is automatically generated (for example index) based on the name of the page or page fragment.

Passing Control Between View Activities

You can configure view activities in your task flow to pass control to each other at runtime. For example, to pass control from one view activity (view activity A) to a second view activity (view activity B), you could configure a command component (a button or a link) on the page associated with view activity A. Set the command component's **Action** attribute to the control flow case from-outcome that corresponds to the task flow activity that you want to invoke (for example, view activity B). At runtime, the end user initiates the control flow case by invoking the command component. You can navigate from a view activity to another activity using either a constant or dynamic value on the **Action** attribute of the UI component.

A constant value of the component's **Action** attribute is an action outcome, as shown in Figure 24-4, that always triggers the same control flow case. When an end user clicks the component, the activity specified in the control flow case is performed. There are no alternative control flows.

Figure 24-4   Edit Property Dialog



Edit Property: Action

○ Method Binding

Managed Bean: <None>

Method:

● Action Outcome

globalHome

A dynamic value of the component's **Action** attribute is bound to a managed bean or a method. The value returned by the method binding determines the next control flow case to invoke. For example, a method might verify user input on a page and return one value if the input is valid and another value if the input is invalid. Each of these different action values trigger different navigation cases, causing the application to navigate to one of two possible target pages.

You can also write an EL expression that must evaluate to `true` before control passes to the target view activity. You write the EL expression as a value for the `<if>` child element of the control flow case in the task flow. For more information, see How to Add a Control Flow Rule to a Task Flow.

How to Pass Control Between View Activities

You pass control to a view activity by specifying the value of the control flow case's `from-outcome` attribute as the value for the `action` attribute of the command component.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you configure the passing of control between view activities. For more information, see Passing Control Between View Activities.

You may also find it helpful to understand other ADF functionality and features. For more information, see Additional Functionality for Task Flow Activities.

You will need to complete these tasks:

- Create a target view activity to receive control after an end user invokes the command component at runtime
- Create a JSF page that will host a command component for the end user to invoke at runtime

To pass control to a view activity:

1. In the Applications window, double-click the JSF page.
2. In the editor window, use one of the following options to add a UI component to the JSF page:
   - In the ADF Faces page of the Components window, from the General Controls panel, drag a navigation UI component, such as **Button** or **Link**, and drop it onto the page.
   - From the Data Controls panel, drag and drop an operation or a method onto the JSF page and, from the context menu, choose **ADF Button** or **ADF Link**.
3. In the Properties window, expand the **Common** section, and select **Edit** in the dropdown menu that you invoke from the icon that appears when you hover over the **Action** property field.
4. In the Edit Property: Action dialog, select **Action Outcome** and choose a value from the associated dropdown list.

   The list contains the control flow case already defined for the view activity associated with the page.

> **Tips:**
>
> The action attribute of the UI component can be bound either to a literal string to hard code a navigation case, or it can be bound to a method binding expression that points to a method, which takes no arguments and returns a String. It cannot be bound to any other type of EL expression.

What Happens When You Pass Control Between View Activities

The following example shows a control flow case defined in the XML source file for a bounded or unbounded task flow.

```
<control-flow-rule>
    <from-activity-id>Start</from-activity-id>
       <control-flow-case>
          <from-outcome>toOffices</from-outcome>
          <to-activity-id>WesternOffices</to-activity-id>
       </control-flow-case>
</control-flow-rule>
```

As shown in Example 24-1, a button on a JSF page associated with the Start view activity specifies toOffices as the **action** attribute. When the user clicks the button, control flow passes to the WesternOffices activity specified as the to-activity-id in the control flow metadata.

Example 24-1   Static Navigation Button Defined in a View Activity

```
<af:button text="Go" id="b1" action="toOffices">
```

Bookmarking View Activities

Bookmarking is available only for view activities within unbounded task flows.

When an end user bookmarks a page associated with a view activity, the URL that displays in the browser's address field for the view is saved as the bookmark. In most cases, this URL cannot be used to redisplay the page associated with the view. For example, the URL may contain state information that cannot be used to redisplay the page.

The bookmark URL should contain information that enables dynamic content on the page to be reproduced. For example, if an end user bookmarks a page displaying a customer's contact information, the bookmark URL needs to contain not only the page but also some identifier for the customer. This enables contact information for the same customer to display when the customer returns to the page using the bookmark.

To make sure that the URL for a page displayed in a browser can be used as a bookmark, identify the view activity associated with the page as bookmarkable.

At runtime, you can identify if a view activity within the unbounded task flow has been designated as bookmarkable using the `isViewBookmarkable()` method. The method is located off the view port context.

After you designate a view activity as bookmarkable, you can optionally specify one or more URL parameters. The value of `url-parameter` is an EL expression. The EL expression specifies where the parameters that will be included in the URL are retrieved when the bookmarkable URL is generated. The EL expression also stores a value from the URL when the bookmarkable URL is dereferenced. The `converter` option identifies a method that performs conversion and validation when parameters are passed via bookmarkable view activity URLs.

In addition, you can specify an optional method that is invoked after updating the application model with submitted URL parameter values and before rendering the view activity. You can use this method to retrieve additional information based on URL parameter key values.

The following example contains the URL syntax for a bookmarked view activity.

```
<server root>/<app_context>/faces/<view activity id>?<param name>=<param
value>&...
```

The syntax of the URL for the bookmarked view activity is:

- `<server root>`: Provided by customization at site or administrator level, for example, `http://mycompany.com/internalApp`.
- `<app context>`: The web application context root, for example, `myapp`. The context root is the base path of a web application. For example, `<app_context>` maps to the physical location of the WEB-INF node on the server.
- `faces`: The Faces servlet mapping. The value in `faces` points to the node containing the `faces-config.xml` configuration file.
- `<view activity id>`: The identifier for the bookmarked view activity, for example, `edit-customers`.
- `<param name>`: The name of the bookmarked view activity URL parameter, for example, `customer-id`.
- `<param value>`: The parameter value, derived from an EL expression, for example, `#{pageFlowScope.employee.id}`. The value of the EL expression must be capable of being represented as a string.

Example 24-2 contains a sample URL for a bookmarkable view activity in an unbounded task flow.

Example 24-2   Sample URL for Bookmarkable View Activity

```
http://mycompany.com/internalApp/MyApp/faces/edit-customers?customer-
id=1234&...
```

How to Create a Bookmarkable View Activity

To create a bookmarkable view activity, designate a view activity as bookmarkable, specify a URL parameter in the bookmark, and specify a method that is executed after the bookmark is dereferenced.

Before you begin:

It may be helpful to have an understanding of the syntax required to create a bookmarkable view activity. For more information, see Bookmarking View Activities.

You may also find it helpful to understand other ADF functionality and features. For more information, see Additional Functionality for Task Flow Activities.

To designate a view activity as bookmarkable:

1. In the Applications window, expand the **WEB-INF** node and double-click the unbounded task flow where you want to designate a view activity as bookmarkable.
2. In the diagram for the unbounded task flow, choose the view activity that you want to designate as bookmarkable.
3. In the Properties window, expand the **Bookmark** section, and select `true` from the **Bookmark** dropdown list.
4. Enter an EL expression in the **Method** field to specify a method to invoke after ADF Controller updates the application model with the corresponding bookmark URL parameter values.
5. To add optional URL parameters to include in the URL for the bookmarked view activity, click **Add** and enter the URL parameter in the Bookmark URL Parameters list:
   - **Name**: the name for the URL parameter.
   - **Value**: A settable EL expression that, when evaluated, specifies the parameter value, for example, `#{pageFlowScope.employeeID}`. The value must be capable of being represented as a string.

     The value property is where the parameters to include in the URL are retrieved from when the bookmarkable URL is generated. In addition, parameters are stored here when the bookmarkable URL is dereferenced. If the EL expression returns `NULL`, the parameter is omitted from the bookmarked view activity URL. Example 24-2 shows how name and value are used to append a bookmark parameter to a view activity URL.
   - **Converter**: (optional) Enter a value binding for the bookmark URL parameter value, for example: `#{pageFlowScope.employee.idConverter}`.

     The object that the EL expression references must implement: `oracle.adf.controller.URLParameterConverter`

A URL parameter converter's `getAsObject()` method takes a single string value as its input parameter and returns an object of the appropriate type. ADF Controller invokes the converter method on the URL parameters before applying the parameter value to the application's model objects. Similarly, the URL parameter converter's `getAsString()` method takes an object as its input parameter and returns a string representation that is used on the URL.

In a JSF application, data values are converted and validated using the converters and validators specified with the UI components on the submitting page. In a Fusion web application using a bookmarkable URL, there is no submitting page to handle conversion and validation. Therefore, you have the option of designating a converter to use for each URL parameter.

What Happens When You Designate a View as Bookmarkable

When you designate a view activity as bookmarkable, a bookmark element is added to the metadata for the view activity, as shown in the following example. The bookmark element can optionally contain metadata specifying URL parameters and a method to execute after the bookmark is dereferenced.

```
<view id="employee-view">
  <page>/folderA/folderB/display-employee-info.jsf</page>
     <bookmark>
        <url-parameter>
          <name>employee-id</name>
          <value>#{pageFlowScope.employee.id}</value>
          <converter>#{pageFlowScope.employee.validateId}</converter>
        </url-parameter>
       <method>#{pageFlowScope.employee.queryData}</method>
     </bookmark>
</view>
```

Specifying HTTP Redirect for a View Activity

You can configure a view activity so that it redirects at runtime to a URL in response to a client request. This causes ADF Controller to create a new browser URL for the view activity. The original URL for the view activity is no longer used. You cannot configure HTTP redirect for a view activity if you have already made the view activity bookmarkable, as described in Bookmarking View Activities.

How to Specify HTTP Redirect for a View Activity

You set a view activity's `redirect` property to `true`.

Before you begin:

It may be helpful to have an understanding of the affect of setting a view activity's `redirect` property to `true`. For more information, see Specifying HTTP Redirect for a View Activity.

You may also find it helpful to understand other ADF functionality and features. For more information, see Additional Functionality for Task Flow Activities.

To specify HTTP redirect for a view activity:

1. In the Applications window, expand the **WEB-INF** node and double-click the task flow.

2. In the task flow diagram, select the view activity.
3. In the Properties window, expand the **General** section, and in the **Redirect** dropdown list, select `true`.

What Happens When You Specify HTTP Redirect for a View Activity

The ADF Controller issues an HTTP redirect in response to a view activity request. The redirected request creates a new browser URL for the view activity. The original view URL is no longer used.

When specified, a HTTP redirect occurs in response to a client `GET` request. For a client `GET`, the `#{bindings}` EL scope is invalid until ADF Controller and ADF Model set up a new bindings context for the page. Therefore, you cannot use EL expressions, such as the following, for view activities for which you specified HTTP redirect:

```
#{bindings.foo}
```

> **Note:**
>
> If you want `http://www.mycompany.org/x.html` to instead display what is at `http://www.mycompany.org/y.html`, do not use refresh techniques such as:
>
> ```
> <META HTTP-EQUIV=REFRESH CONTENT="1; URL=http://www.example.org/
> bar">
> ```
>
> This technique could adversely affect back button behavior. If an end user clicks a browser back button, the refresh occurs again, and navigation is forward, not backward as expected.
>
> In this situation, use HTTP redirect instead.

Specifying URL Aliases for View Activities

The URL that a Fusion web application renders has a format that is generated at runtime based on the current context:

- If the view activity is in a bounded task flow, the application renders a URL in the following format:

  ```
  http://www.example.com/web-app-context-root/faces/task-flow-id/view-activity-
  ID
  ```

- If the view activity is in an unbounded task flow, the application renders a URL in the following format:

  ```
  http://www.example.com/web-app-context-root/faces/view-activity-ID
  ```

In order for end users to see a URL that may have a more meaningful name, you can specify a URL alias for a view activity. The URL alias value you specify renders instead of the default generated values. For example, in a task flow that manages data about fruit, you might have view activities to view and edit information about apples and oranges. In this scenario, your Fusion web application renders URL aliases like the following when an end user navigates to the view activity pages for apples and oranges:

```
http://www.example.com/web-app-context-root/faces/fruit/apples
http://www.example.com/web-app-context-root/faces/fruit/oranges
```

You can specify a number of different levels in URL aliases to communicate a hierarchy to end users. This means that both of the following values are valid for a view activity's URL alias:

```
/food/fruit/pineapple
/drink/juice/pineapple
```

However, the URL alias that you specify must be unique within the task flow that contains the view activity.

Specify URL aliases for view activities that render JSF pages, not JSF page fragments. This latter type of document does not render directly in the browser.

How to Specify a URL Alias for a View Activity

You specify a URL alias for a view activity by writing a value for the view activity's View-ID property (`<view-id>`).

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you specify a URL alias for a view activity. For more information, see Specifying URL Aliases for View Activities.

You may also find it helpful to understand other ADF functionality and features. For more information, see Additional Functionality for Task Flow Activities.

You will need to complete these tasks:

- Create the view activity that renders a JSF page at runtime
- Create the JSF page that the end user views at runtime

To specify a URL alias for a view activity:

1. In the Applications window, expand the **WEB-INF** node and double-click the task flow that contains the view activity for which you want to specify a URL alias.
2. In the diagram for the task flow, select the view activity for which you want to specify a URL alias.
3. In the Properties window, expand the **General** section, and enter the URL alias value in the **View-ID** field.

   For example, enter a value in the following format:

   ```
   /food/fruit/citrus/lemons
   ```

What Happens When You Specify a URL Alias for a View Activity

JDeveloper adds the value that you specify in the **View-ID** field to the metadata for the view activity, as shown in the following example.

```
<view id="view1">
    <page>/view1.jsf</page>
    <view-id>/food/fruit/citrus/lemons</view-id>
  </view>
```

Using URL View Activities

The ADF task flow activity can display URLs and allow users to navigate to an external resource when you use the URL view activity. You can use this activity to redirect users to a location within the same application or you can redirect users to navigate to a region out of the web application.

You can use a URL view activity to redirect the root view port (for example, a browser page) to any URL-addressable resource, even from within the context of an ADF region. URL addressable resources include:

- Bounded task flows
- View activities in the unbounded task flow
- Addresses external to the current web application (for example, `http://www.oracle.com`)

To display the resource, you specify an EL expression that evaluates at runtime to generate the URL to the resource. In addition, you can specify EL expressions that, when evaluated, are added as parameters and parameter values to the URL.

A URL view activity redirects the client regardless of the view port (root view port or an ADF region) from which it is executed. The `<redirect>` element of a view activity performs in a similar way, except that it can be used only if the view activity is within the root view port. The `<redirect>` element is ignored within the context of an ADF region. See Specifying HTTP Redirect for a View Activity.

Redirecting elsewhere within the same application using URL view activities (not the `<redirect>` element) is handled similarly to back button navigation since the task flow stack is cleaned up. Redirecting out of the web application is handled like dereferencing a URL to a site external to the application.

How to Add a URL View Activity to a Task Flow

You can add a URL view activity to a bounded or unbounded task flow.

Before you begin:

It may be helpful to have an understanding of the affect of adding a URL view to the functionality of a task flow. For more information, see Using URL View Activities.

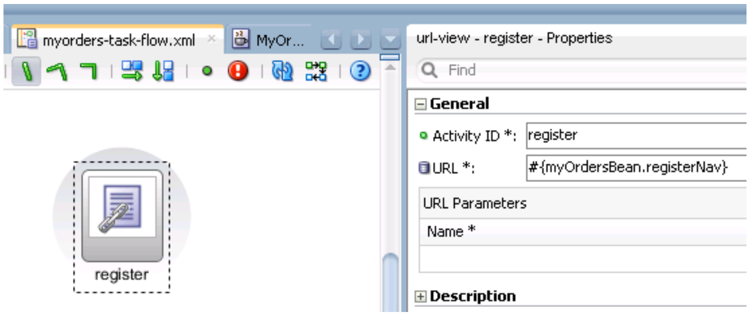
You may also find it helpful to understand other ADF functionality and features. For more information, see Additional Functionality for Task Flow Activities.

To add a URL view activity to a task flow:

1. In the Applications window, expand the **WEB-INF** node and double-click the task flow to which you want to add a URL view activity.
2. In the ADF Task Flow page of the Components window, from the Components panel, in the Activities group, drag and drop a **URL View** onto the diagram.
3. In the Properties window, expand the **General** section, and enter an ID that identifies the URL view activity in the **Activity ID** field.
4. In the **URL** field, enter an EL expression that returns the URL at runtime. Click the icon that appears when you hover over the property field to use the Expression Builder if needed.

For example, Figure 24-5 shows a URL activity (**register**) in an application's `myorders-task-flow.xml` bounded task flow with an EL expression (`#{myOrdersBean.registerNav}`) that retrieves a URL at runtime.

Figure 24-5 URL View Activity



General

Activity ID *: register

URL *: #{myOrdersBean.registerNav}

URL Parameters

Name *

Description

5. Expand the **URL Parameters** section to add optional URL parameters to include in the URL:
   - **Name**: A name for the parameter.
   - **Value**: An EL expression that, when evaluated, generates the parameter value.
   - **Converter**: A settable EL expression that, when evaluated, specifies a method to perform conversion and validation when parameters are passed via bookmarkable view activity URLs. For more information, see Bookmarking View Activities.

What Happens When You Add a URL View Activity to a Task Flow

JDeveloper writes entries similar to those shown in the following example for the URL view activity in the source file of the task flow.

```
<url-view id="register">
      <url>#{myOrdersBean.registerNav}</url>
    </url-view>
```

What You May Need to Know About URL View Activities

Task flow URL view activities can be used within JSF portlets. Create the URL using one of the following options if you configure a task flow URL view activity for use within a JSF portlet:

- Invoke one of the following methods from the `ControllerContext` class in the `oracle.adf.controller` package:
  - `getLocalViewActivityURL()`
  - `getGlobalViewActivityURL()`

  Do not invoke the `encodeActionURL()` method from the `ADFPortletContainerExternalContext` class with the response from the `getLocalViewActivityURL()` or `getGlobalViewActivityURL()` methods before you invoke the `redirect()` method from the `ADFPortletContainerExternalContext` class. The `getLocalViewActivityURL()`

and `getGlobalViewActivityURL()` methods both perform the necessary encoding of the URL.

- Supply a fully qualified absolute URL.
- Supply a context path relative URL.
- Supply a URL relative to the current view.

A task flow URL view activity in a JSF portlet behaves as follows:

- If the redirect URL refers to a location in the JSF portlet application and does not contain a `queryString` parameter named `x_DirectLink` with a value of `true`, the portlet in the containing page navigates to the URL specified in the URL view activity.
- Otherwise, the client redirects to the URL specified in the URL view activity.

For more information about using a task flow's URL view activity, see Using URL View Activities.

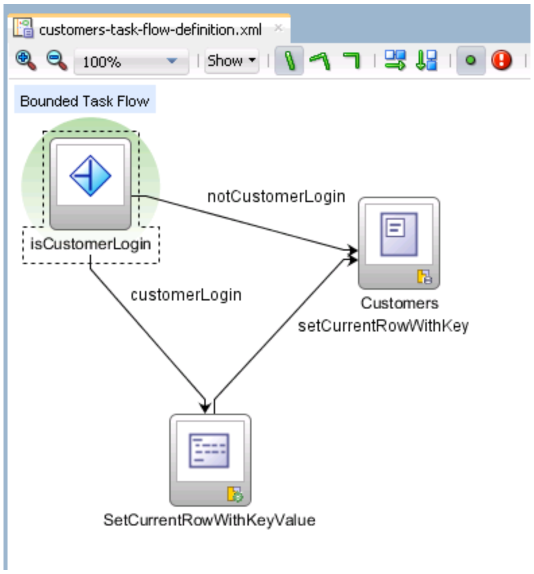
For information about the methods that you can use to get a URL, see the *Java API Reference for Oracle ADF Controller*.

Using Router Activities

While creating a task flow you may need to direct users to the activities based on the runtime evaluation of EL expressions. The ADF router activity allows you to route such controls to the appropriate activities.

Use a router activity to route control to activities based on the runtime evaluation of EL expressions. Figure 24-6 shows a router activity (**isCustomerLogin**) in the `customer-task-flow-definition.xml` task flow from the Summit ADF task flow sample application that can branch to different control flows leading from it to different activities.

Figure 24-6 Router for Alternate Control Flow Cases



Each control flow corresponds to a different router case. Each router case uses the following elements to choose the activity to which control is next routed:

- `expression`: an EL expression that evaluates to `true` or `false` at runtime.

  The router activity returns the outcome that corresponds to the EL expression that returns `true`.
- `outcome`: a value returned by the router activity if the EL expression evaluates to `true`, for example, `customerLogin`.

  If the router case `outcome` matches a `from-outcome` on a control flow case, control passes to the activity that the control flow case points to. If none of the cases for the router activity evaluate to `true`, or if no router activity cases are specified, the `outcome` specified in the router **Default Outcome** field (if any) is used.

Figure 24-6, for example, shows a router activity that passes control flow based on the evaluation of an EL expression that determines if the user is logged in and has a specific application role. If true, control passes to the SetCurrentRowWithKeyValue method call activity. Otherwise, control passes to the Customers view activity.

> **Note:**
>
> Use a router activity if your routing condition can be expressed in an EL expression.
>
> Using a router activity allows you to do more when you design the task flow that contains it. The router activity allows you to show more information about the condition on the task flow. This makes it more readable and useful to someone who looks at the diagram for your task flow.
>
> Using a router activity also makes it easier to modify your application later. For example, you may want to modify or add a routing condition later.

How to Configure Control Flow Using a Router Activity

You define a control flow by dragging a router activity from the Components window to the diagram for the task flow. You configure the Activity ID and Default Outcome properties of the router activity and add router cases to the router activity.

Before you begin:

It may be helpful to have an understanding of the properties of a router activity that can affect functionality. For more information, see Using Router Activities.

You may also find it helpful to understand other ADF functionality and features. For more information, see Additional Functionality for Task Flow Activities.

To configure control flow using the router activity:

1. In the Applications window, expand the **WEB-INF** node and double-click the task flow where you want to configure a control flow using a router activity.
2. In the ADF Task Flow page of the Components window, from the Components panel, in the Activities group, drag and drop a **Router** activity onto the diagram.

3. In the Properties window, expand the **General** section and enter values for the following:
   - **Activity ID**: write a value that identifies the router activity in the task flow's source file.
   - **Default Outcome**: specify an activity that the router activity passes control to if no control flow cases evaluate to `true` or if no control flow case is specified.
4. Click the **Add** icon for the **Cases** section and specify values for each router case that you add:
   - **Expression**: An EL expression that evaluates to `true` or `false` at runtime.

     For example, to reference the value in an input text field of a view activity, write an EL expression similar to the following:

     ```
     #{pageFlowScope.value=='view2'}
     ```

     If this EL expression returns `true`, the router activity invokes the outcome that you specify in the **Outcome** field.
   - **Outcome**: The outcome the router activity invokes if the EL expression specified by **Expression** returns `true`.

     Create a control flow case or a wildcard control flow rule for each outcome in the diagram of your task flow. For example, for each outcome in a control flow case, make sure that there is a corresponding `from-outcome`.

What Happens When You Configure Control Flow Using a Router Activity

JDeveloper writes values to the source file of the task flow based on the values that you specify for the properties of the router activity. Example 24-3 shows the values that appear in the `customer-task-flow-definition.xml` task flow from the Summit ADF task flow sample application that branch to different control flows leading from the router activity to different activities.

At runtime, the router activity passes control to the control flow case specified by `outcome` if the EL expression returns `true`.

Example 24-3 Router Activity in the Summit ADF Task Flow Sample Application

```
<router id="isCustomerLogin">
      <case id="__4">
        <expression>#{securityContext.userInRole['Application Customer Role']}</expression>
        <outcome>customerLogin</outcome>
      </case>
      <default-outcome>notCustomerLogin</default-outcome>
    </router>
...
<control-flow-case id="__7">
        <from-outcome>customerLogin</from-outcome>
        <to-activity-id>SetCurrentRowWithKeyValue</to-activity-id>
      </control-flow-case>
```

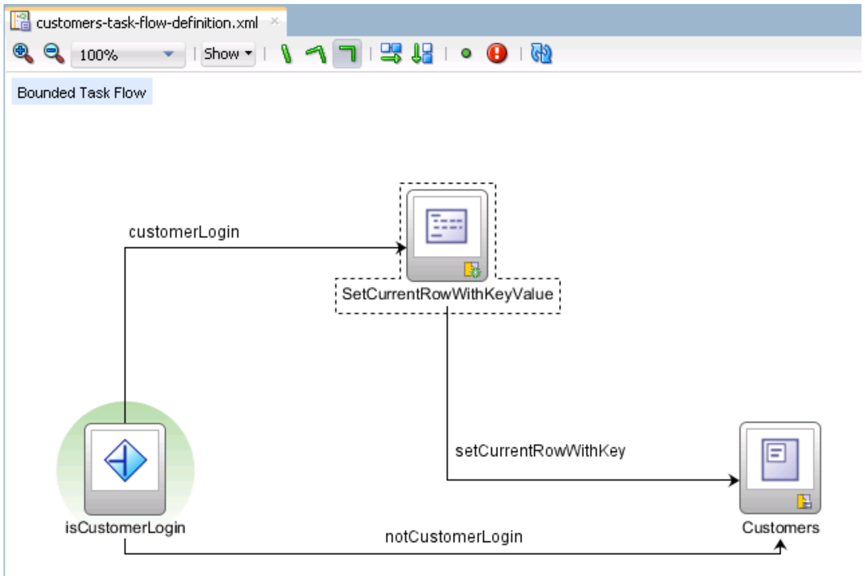
Using Method Call Activities

The ADF task flow allows you to call a custom method or invoke a built-in methods using method call activity. You can define the control flow to and from the method call activity. The method call outcome can be specified either as a fixed-outcome or as a to-string outcome.

Use a method call activity to call a custom or built-in method that invokes application logic from anywhere within an application's control flow. You can specify methods to perform tasks such as initialization before displaying a page, cleanup after exiting a page, exception handling, and so on.

Figure 24-7 shows the `customers-task-flow-definition.xml` task flow from the Summit ADF task flow sample application where the `SetCurrentRowWithKeyValue` method call activity invokes a `setCurrentRowWithKeyValues` operation from the `BackOfficeAppModuleDataControl` data control.

Figure 24-7 Method Call Activity to Set Row



You can set an outcome for the method that specifies a control flow case to pass control to after the method finishes. You can specify the outcome as either:

- `fixed-outcome`: On successful completion, the method always returns this single outcome, for example, `success`. If the method does not complete successfully, an outcome is not returned. If the method type is void, you must specify a `fixed-outcome`, not a `to-string`.
- `to-string`: If specified as `true`, the outcome is based on calling the `toString()` method on the Java object returned by the method. For example, if `toString()` returns `editBasicInfo`, navigation goes to a control flow case named `editBasicInfo`.

The following example shows the metadata for the method call activity in Figure 24-7 where the `fixed-outcome` element specifies `setCurrentRowWithKey` as the outcome to return.

```
<method-call id="SetCurrentRowWithKeyValue">
      <method>#{bindings.setCurrentRowWithKeyValue.execute}</method>
      <outcome>
        <fixed-outcome>setCurrentRowWithKey</fixed-outcome>
      </outcome>
```

For more information about control flows, see About Control Flows.

How to Add a Method Call Activity

Drag a method call activity from the Components window to the task flow diagram. You can associate the method call activity with an existing method by dropping a data control operation from the Data Controls panel directly onto the method call activity in the task flow diagram.

In the Summit ADF task flow sample application, for example, you could drag a `setCurrentRowWithKey` or `setCurrentRowWithKeyValues` operation to the diagram from the `BackOfficeAppModuleDataControl` data control in the Data Controls panel to create a method call activity that displays or selects the current row in a table.

Note:

Parameters for data control method parameters are defined in the page definition for the corresponding page rather than within ADF Controller metadata.

You can also drag methods and operations directly to the task flow diagram. A new method call activity is created automatically after you drop it on the diagram. You can specify an EL expression and other options for the method.

Tip:

To identify the method that a method call activity invokes, right-click the method call activity in the diagram of the task flow and choose **Go to Method**. JDeveloper navigates to the method that the method call activity invokes.

Before you begin:

It may be helpful to have an understanding of how the properties of a method call activity affect the functionality of a task flow. For more information, see Using Method Call Activities.

You may also find it helpful to understand other ADF functionality and features. For more information, see Additional Functionality for Task Flow Activities.

You will need to complete this task:

Create a bounded or unbounded task flow. For more information, see Creating a Task Flow.

To add a method call activity to a task flow:

1. In the Applications window, expand the **WEB-INF** node and double-click the task flow where you want to add a method call activity.
2. In the ADF Task Flow page of the Components window, from the Components panel, in the Activities group, drag and drop a method call activity onto the diagram.

   The method call activity optionally displays a default activity ID, `methodCall`*`n`*, and a warning icon that indicates that a method EL expression has not yet been specified.

   For more information about turning on the warning icons, see How to Add an Activity to a Task Flow.
3. In the Properties window, expand the **General** section, and enter an activity ID in the **Activity ID** field if you want to change the default value.

   If you enter a new value, the new value appears under the method call activity in the diagram.
4. In the **Method** field, enter an EL expression that identifies the method to call.

   For example, enter an EL binding expression similar to the following:

   ```
   #{bindings.setCurrentRowWithKeyValue.execute}
   ```

   > **Note:**
   >
   > The `bindings` variable in the EL expression references an ADF Model binding from the current binding container. In order to specify the bindings variable, you must specify a binding container definition or page definition. For more information, see Working with Page Definition Files.

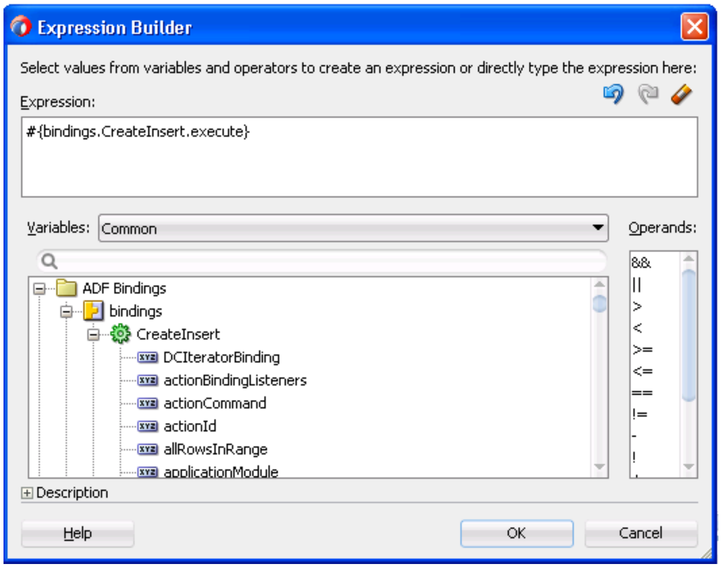
   You can also use the Expression Builder shown in Figure 24-8 to build the EL expression for the method:

   a. Choose **Method Expression Builder** from the context menu that appears when you click the icon that appears when you over the **Method** property field.

   b. In the Expression Builder dialog, navigate to the method that you want to invoke and select it.

      The Expression Builder dialog should look similar to Figure 24-8. In Figure 24-8, for example, the EL expression shown at the top of the Expression Builder references the `CreateInsert` action binding from the method call activity in the `emp-reg-task-flow-definition.xml` task flow of the Summit ADF task flow sample application.

Figure 24-8   EL Expression for Method in Expression Builder Dialog



Select values from variables and operators to create an expression or directly type the expression here:

Expression:

#{bindings.CreateInsert.execute}

Variables: Common

c. Click **OK**.

Tip:

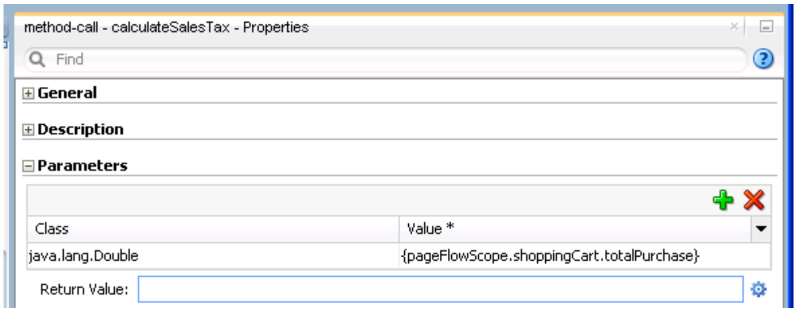
If the method call activity is going to invoke a managed bean method, double-click the method call activity in the diagram. This invokes a dialog where you can specify the managed bean method you want to invoke.

5. In the Properties window, expand the **General** section, and specify one of the following in the **Outcome** group:
   - **Fixed Outcome**: On successful completion, the method always returns this single outcome, for example, `success`. If the method does not complete successfully, an outcome is not returned. If the method type is void, you must specify a value for **Fixed Outcome** rather than for **to-string**.
   - **tostring()**: If you select **true**, the outcome is based on calling the `toString()` method on the Java object returned by the method.

How to Specify Method Parameters and Return Values

You can specify parameters and return values for a method. Figure 24-9 shows the Properties window with a single parameter defined for a method called `calculateSalesTax`. The **Value** field contains an EL expression that evaluates to the parameter value at runtime.

Figure 24-9 Method Parameters in a Method Call



Class	Value *
java.lang.Double	{pageFlowScope.shoppingCart.totalPurchase}

If parameters have not already been created by associating the method call activity to an existing method, add the parameters yourself.

Before you begin:

It may be helpful to have an understanding of how the properties of a method call activity affect the functionality of a task flow. For more information, see Using Method Call Activities.

You may also find it helpful to understand other ADF functionality and features. For more information, see Additional Functionality for Task Flow Activities.

You will need to complete this task:

> Add a method call activity to the task flow diagram, as described in How to Add a Method Call Activity.

To add method parameters:

1. In the Applications window, expand the **WEB-INF** node and double-click the task flow that contains the method call activity to which you want to add method parameters.
2. In the task flow diagram, select the method call activity.
3. In the Properties window, expand the **Parameters** section, and click **Add** to set the following values for the parameter that you want to add:
   - **Class**: Enter the parameter class, for example, `java.lang.Double`.
   - **Value**: Enter an EL expression that retrieves the value of the parameter. For example:

     ```
     #{pageFlowScope.shoppingCart.totalPurchasePrice}
     ```
4. In the **Return Value** field, enter an EL expression that identifies where to store the method return value, for example, `#{pageFlowScope.Return}`.
5. Repeat the above steps to add additional parameters.

What Happens When You Add a Method Call Activity

After you specify parameters and return values for a method, the XML source file is updated. Example 24-4 shows the `SetCurrentRowWithKeyValue` method call activity in the `customers-task-flow-definition.xml` task flow from the Summit ADF task flow sample application.

Example 24-4 Method Call Activity in the Summit ADF Sample Application

```
<method-call id="SetCurrentRowWithKeyValue">
      <method>#{bindings.setCurrentRowWithKeyValue.execute}</method>
      <outcome>
        <fixed-outcome>setCurrentRowWithKey</fixed-outcome>
      </outcome>
    </method-call>
```

Using Task Flow Call Activities

The ADF task flow allows you to call a bounded task flow, to perform a set of activities, from a bounded or unbounded task flow. This is helpful when you want a set of activity to be performed repeatedly in your task flow or you want to perform an activity based on the outcome of another activity.

You can use a task flow call activity to call a bounded task flow from either the unbounded task flow or a bounded task flow. A task flow call activity allows you to call a bounded task flow located within the same or a different application.

The called bounded task flow executes its default activity first. There is no limit to the number of bounded task flows that can be called. For example, a called bounded task flow can call another bounded task flow, which can call another, and so on resulting in the creation of *chained task flows* where each task flow is a link in a chain of tasks.

To pass parameters into a bounded task flow, you must specify input parameter values on the task flow call activity. These values must correspond to the input parameter definitions on the called bounded task flow. For more information, see How to Specify Input Parameters on a Task Flow Call Activity .

Also note the following:

- The value on the task flow call activity **Input Parameter** specifies where the value will be taken from within the calling task flow.
- The value on the **Input Parameter Definition** for the called task flow specifies where the value will be stored within the called bounded task flow once it is passed.

> **Tip:**
>
> When a bounded task flow is associated with a task flow call activity, input parameters are automatically inserted on the task flow call activity based on the input parameter definitions defined on the bounded task flow. Therefore, the application developer needs only to assign values to the task flow call activity input parameters.

By default, all objects are passed by reference. Primitive types (for example, `int`, `long`, or `boolean`) are always passed by value.

The technique for passing return values out of the bounded task flow to the caller is similar to the way that input parameters are passed. For more information, see Configuring a Return Value from a Bounded Task Flow.

To use a task flow call activity:

1. Call a bounded task flow using a task flow call activity.
2. Specify input parameters on a task flow call activity if you want to pass parameters into the bounded task flow.
3. Call a bounded task flow in another web application using a URL.
4. Specify before and after listeners on a task flow call activity.

How to Call a Bounded Task Flow Using a Task Flow Call Activity

Add a task flow call activity to the calling bounded or unbounded task flow to call a bounded task flow.

Before you begin:

It may be helpful to have an understanding of how a task flow call activity interacts with a task flow. For more information, see Using Task Flow Call Activities.

You may also find it helpful to understand other ADF functionality and features. For more information, see Additional Functionality for Task Flow Activities.

To call a bounded task flow using a task flow call activity:

1. In the Applications window, expand the **WEB-INF** node and double-click the task flow (calling task flow) to which you are going to add a task flow call activity to call a bounded task flow.
2. In the ADF Task Flow page of the Components window, from the Components panel, in the Activities group, drag and drop a **Task Flow Call** activity onto the diagram.
3. Choose one of these options to identify the called task flow:
   - In the task flow diagram, double-click the task flow call activity.

     The Create Bounded Task Flow dialog appears, where you specify options for creating a new bounded task flow.
   - Drag an existing bounded task flow from the Applications window and drop it on the task flow call activity.

> **Tips:**
>
> You can drop a bounded task flow on a page or page fragment. If the bounded task flow consists of pages (not page fragments), you can choose the **Task Flow Call as Button** or **Task Flow Call as Link** menu options that appear to add a button or link component on the page where you drop the task flow. An end user can click the button or link to call the task flow. This may in turn automatically generate the task flow call activity if the page is already associated with an existing view activity in a task flow.
>
> You cannot drop a bounded task flow from one application to a task flow diagram contained in another application using the Applications window, even though both applications appear in the Applications window. In addition, you cannot drop a bounded task flow contained in one project onto a task flow diagram contained in another project.
>
> Instead, you can package the bounded task flow in an ADF library, then reuse it in your current application or project. You can then drag the bounded task flow from the Resources window or from the Components window page that is created when you import the library. For more information, see Using the Resources Window.

- If you know the name of the bounded task flow that you want to invoke, perform the following steps:

  a. In the task flow diagram, select the task flow call activity.

  b. In the Properties window, expand the **General** section, and select **Static** from the **Task Flow Reference** dropdown list.

  c. In the **Document** field, enter the name of the source file for the bounded task flow to call. For example, `called-task-flow-definition.xml`.

  d. In the **ID** field, enter the bounded task flow ID contained in the XML source file for the called bounded task flow, for example, `targetTaskFlow`.

- If you do not know the name of the bounded task flow to invoke and it is dependent on an end user selection at runtime, perform the following steps:

  a. In the task flow diagram, select the task flow call activity.

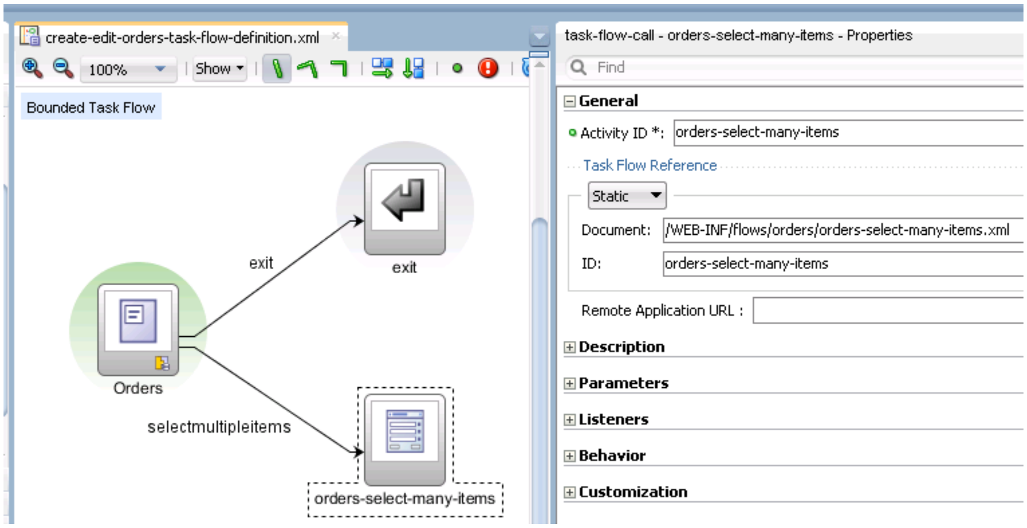
  b. In the Properties window, expand the **General** section, and select **Dynamic** from the **Task Flow Reference** dropdown list.

  c. Select **Expression Builder** from the context menu that appears when you click the icon that appears when you hover over the **Dynamic Task Flow Reference** property field.

  d. Write an EL expression that identifies the ID of the bounded task flow to invoke at runtime.

Figure 24-10 shows the `customers-task-flow` task flow in the Summit ADF task flow sample application. This task flow contains a task flow call activity (**create-edit-orders-task-flow-definition**) that invokes the task flow of the same name.

Figure 24-10 Task Flow Call Activity That Invokes a Bounded Task Flow



task-flow-call - orders-select-many-items - Properties

Find

General

Activity ID *: orders-select-many-items

Task Flow Reference

Static

Document: /WEB-INF/flows/orders/orders-select-many-items.xml

ID: orders-select-many-items

Remote Application URL :

Description

Parameters

Listeners

Behavior

Customization

What Happens When You Call a Bounded Task Flow Using a Task Flow Call Activity

JDeveloper generates metadata entries in the source file for the task flow that calls the bounded task flow. Example 24-5 shows the metadata that corresponds to Figure 24-10 from the Summit ADF task flow sample application. At runtime, the task flow call activity invokes the `orders-select-many-items` task flow.

Example 24-5 Task Flow Call Activity in the Summit ADF Sample Application

```
<task-flow-call id="orders-select-many-items">
      <task-flow-reference>
        <document>/WEB-INF/flows/orders/orders-select-many-items.xml</document>
        <id>orders-select-many-items</id>
      </task-flow-reference>
      <run-as-dialog>
        <display-type>
          <inline-popup/>
        </display-type>
      </run-as-dialog>
    </task-flow-call>
```

How to Specify Input Parameters on a Task Flow Call Activity

The suggested method for mapping parameters between a task flow call activity and its called bounded task flow is to first specify input parameter definitions for the called bounded task flow. Then you can drag the bounded task flow from the Applications window and drop it on the task flow call activity. The task flow call activity input parameters will be created automatically based on the bounded task flow's input parameter definition. For more information, see Passing Parameters to a Bounded Task Flow.

You can, of course, first specify input parameters on the task flow call activity. Even if you have defined them first, they will automatically be replaced based on the input parameter definitions of the called bounded task flow, once it is associated with the task flow call activity.

If you have not yet created the called bounded task flow, you may still find it useful to specify input parameters on the task flow call activity. Doing so at this point allows you to identify any input parameters you expect the task flow call activity to eventually map when calling a bounded task flow.

Before you begin:

It may be helpful to have an understanding of how a task flow call activity interacts with a task flow. For more information, see Using Task Flow Call Activities.

You may also find it helpful to understand other ADF functionality and features. For more information, see Additional Functionality for Task Flow Activities.

To specify input parameters on a task flow call activity:

1. In the Applications window, expand the **WEB-INF** node and double-click the task flow that contains the task flow call activity.
2. In the diagram, select the task flow call activity.
3. In the Properties window, expand the **Parameters** section, and click the **Add** icon to specify a new input parameter in the **Input Parameters** list as follows:
   - **Name**: Enter a name to identify the input parameter.
   - **Value**: Enter an EL expression that identifies the parameter value. The EL expression identifies where the parameter value will be retrieved from within the calling task flow. For example, enter an EL expression similar to the following:

     ```
     #{pageFlowScope.callingTaskflowParm}
     ```

     By default, all objects are passed by reference. Primitive types (for example, `int`, `long`, or `boolean`) are always passed by value.

     Click in the Value field and use the dropdown list to use the Expression Builder if needed.
4. After you have specified an input parameter, you can specify a corresponding input parameter definition for the called bounded task flow. For more information, see Passing Parameters to a Bounded Task Flow.

> **Note:**
>
> If you call a task flow without the required input parameter, the task flow is not executed and an error is logged in the `oracle.adfinternal.controller.activity` logger. Therefore, it is difficult to know the error when a task flow fails to execute.
> For a normal task flow call, an `ActivityLogicException` is thrown if the required input parameters are missing. When the task flow is invoked in a region or directly from the URL, without the required input parameters, the task flow is not processed and displays a blank region or page. In case of region, the rest of the page is rendered properly.

How to Call a Bounded Task Flow Using a URL

You can call a bounded task flow that does not use page fragments (`.jsff`) in another web application using a URL. Use a task flow call activity to call the bounded task flow that you want to invoke. You write an EL expression for the task flow call activity's `remote-app-url` property that, when evaluated, returns a URL.

In addition to writing a value for the `remote-app-url` property, you specify values for task flow reference properties that identify the bounded task flow to call. The task flow reference and the `remote-app-url` property are combined at runtime to generate a URL to the called bounded task flow in the remote web application.

You also need to set visibility properties for the bounded task flow in the remote web application that you want to call so that it invokes when it receives the URL generated from the task flow call activity in the calling task flow.

Be aware that JSF portlets provide all content from the same web application. As a result, do not configure your web application to invoke a task flow using a URL if you plan to use your web application in a JSF portlet.

> **Note:**
>
> If, in JDeveloper's Applications window, you right-click the bounded task flow that you want to call using a URL and select **Run**, the bounded task flow executes as if it were called using a URL at runtime. For this reason, make sure to set the visibility properties for the bounded task flow even if you want to execute it as part of testing in JDeveloper.

Before you begin:

It may be helpful to have an understanding of how a task flow call activity interacts with a task flow. For more information, see Using Task Flow Call Activities.

You may also find it helpful to understand other ADF functionality and features. For more information, see Additional Functionality for Task Flow Activities.

To call a bounded task flow using a URL:

1. In the Applications window, expand the **WEB-INF** node and double-click the task flow that you want to configure to invoke a bounded task flow in a remote web application.
2. In the ADF Task Flow page of the Components window, from the Components panel, in the Activities group, drag a **Task Flow Call** activity and drop it on the diagram for the task flow.
3. In the Properties window, expand the **General** section and specify values to invoke a bounded task flow.

   For more information, see How to Call a Bounded Task Flow Using a Task Flow Call Activity.
4. For the **Remote Application URL** property, use the Expression Builder to write an EL expression that, when evaluated, returns a string with the parts required to construct a URL of the remote web application. Invoke the Expression Builder by selecting **Expression Builder** from the context menu that appears when you click the icon that appears when you hover over the property field.

   For example, the following EL expression invokes a managed bean method that returns a string with the parts required to construct a URL:

   ```
   #{myOrdersBean.createOrder}
   ```

5. Open the bounded task flow in the remote web application that you want to invoke.

> **Note:**
>
> The bounded task flow you specify cannot use page fragments (`.jsff`).

6. In the Structure window, right-click the node for the bounded task flow (**task-flow-definition- *Task Flow ID***) and choose **Go to Properties**.
7. In the Properties window, expand the **General** section and select values for the following:
   - **URL Invoke**: select **url-invoke-allowed** from the dropdown list if you want to allow a URL to invoke the bounded task flow. Select **url-invoke-disallowed** if you do not want to allow a URL to invoke the bounded task flow. Selecting this value returns a HTTP 403 status code if a URL attempts to invoke the bounded task flow. The default value (**calculated**) allows a URL to invoke the bounded task flow if the bounded task flow does not specify an initializer and it has a view activity as its default activity. If the bounded task flow does not meet these conditions, a HTTP 403 status code is returned. Selecting **url-invoke-allowed** or **url-invoke-disallowed** overrides the default behavior.
   - **Library Internal:** select **true** if you want the bounded task flow to be internal when you package it in an ADF Library JAR. The default value is **false**.

     For more information about packaging a bounded task flow in an ADF Library JAR, see Packaging a Reusable ADF Component into an ADF Library.
8. Save and close the bounded task flow.

What Happens When You Configure a Bounded Task Flow to be Invoked by a URL

JDeveloper generates metadata entries in the source file of the task flow that invokes the task flow call activity to the bounded task flow in a remote web application. The following example shows an entry for a task flow call activity.

```
<task-flow-call id="createOrder">
    <task-flow-reference>
      <document id="__6">myorders-task-flow.xml</document>
      <id id="__5">myorders-task-flow</id>
    </task-flow-reference>
    <remote-app-url id="__7">#{myOrdersBean.createOrder}</remote-app-url>
  </task-flow-call>
```

The `createOrder` method in the previous example returns a string with the URL syntax required to invoke a bounded task flow. For more information about the URL syntax, including descriptions of the required parts in the returned string and an example URL, see What You May Need to Know About Calling a Bounded Task Flow Using a URL.

JDeveloper also generates entries in the source file for the bounded task flow to invoke when you configure it so it can be called by a URL. The following example shows sample metadata entries that allow a bounded task flow to be invoked by a URL.

```
<task-flow-definition id="task-flow-definition3">
    <visibility id="__2">
      <url-invoke-allowed/>
      <library-internal/>
    </visibility>
  </task-flow-definition>
```

What You May Need to Know About Calling a Bounded Task Flow Using a URL

Adding a context parameter in your local application's deployment descriptor may ease the administration of interaction with a remote web application.

Context Parameter for Remote Web Application

Consider adding a context parameter to the deployment descriptor (`web.xml`) for your Fusion web application (local application) if you use a URL to invoke a bounded task flow in another Fusion web application (remote application). Set the value of the context parameter to the URL of the remote application. Use the context parameter name when writing EL expressions in the local application, as shown in the following example where `remoteAppUrl` is the name of the context parameter:

```
#{initParam.remoteAppUrl}
```

If the URL of the remote application changes, you can update the context parameter to reference the changed URL.

Object Type for Parameter Converters

Parameter converters, if specified, can be used to convert task flow parameter values to and from the string representations used in a URL. A parameter converter is an EL expression that evaluates to an object of the following type:

```
oracle.adf.controller.UrlParameterConverter
```

If you do not specify a parameter converter, a default converter checks the parameters for cross-site-scripting (XSS) attacks. If you know that the parameter values used in your application contain special characters, you should create your own implementation of `UrlParameterConverter` and use it to perform conversion of the task flow parameter values.

URL Syntax to Invoke a Bounded Task Flow

Typically, you write an EL expression that references a managed bean method which, in turn, retrieves the required parts of the URL or you could write an EL expression that returns the URL directly, as shown in the following example.

```
http://somecompany.com/internalApp/MyApp/faces/adf.task-flow?
adf.tfId=displayHelp&
adf.tfDoc=%2FWEB-INF%2Fdisplayhelp.xml&topic=createPurchaseOrder
```

Example 24-6 describes the parts of the URL syntax to invoke a bounded task flow.

The following list describes each part of the URL syntax in Example 24-6:

- `<server root>`: Provided by customization at site or administrator level. For example:

  ```
  http://mycompany.com/internalApp
  ```

The <server root> value depends on the application server where you deploy the bounded task flow. The bounded task flow URL is a resource within the JSF servlet's URL path.

- <app context>: The web application context root, for example, MyApp. The context root is the base path of a web application.
- faces: Faces servlet mapping.
- adf.task-flow: A reserved keyword that identifies ADF Controller for the remote web application.
- adf.tfId: A URL parameter that supplies the task flow ID to call.
- <task flow ID>: The identifier of the bounded task flow to call, for example, displayHelp. This is the same task flow ID that is used when calling locally. Note that this identifier is not the same as the task flow call activity instance ID. The parameter value must be represented as a string.
- adf.tfDoc: A URL parameter that supplies the document name containing the bounded task flow ID to be called.
- <document name>: A document name containing the bounded task flow ID to be called, for example, %2FWEB-INF%2FtoUppercase%2FdisplayHelp.xml. If you are handcrafting the bounded task flow URL, you are responsible for the appropriate encoding.
- <named parameter>: (optional) The name of an input parameter definition for the called bounded task flow, for example, topic. You must supply all required input parameter definitions.
- <named parameter value>: (optional) The value of the input parameter.

> **Note:**
>
> URL parameter names that begin with an underscore ('_') are intended for internal use only and should not be used. Although you may see these names on URLs generated by ADF Controller, you should not attempt to use or depend on them.

Example 24-6 URL Syntax for a Call to a Bounded Task Flow Using Named Parameters

```
<server root>/<app_context>/faces/adf.task-flow?adf.tfid=<task flow definition
ID>&adf.tfDoc=<document name>&<named parameter>=<named parameter value>
```

How to Specify Before and After Listeners on a Task Flow Call Activity

Use task flow call activity before and after listeners to identify the start and end of a bounded task flow. Specifying a listener in the task flow call activity means that the listener executes on that specific usage of the called bounded task flow.

You specify the listener as an EL expression for a method that will be called upon entry or exit of a bounded task flow, for example:

```
<before-listener>#{global.showState}</before-listener}>
```

The method cannot have parameters.

- Before listener: An EL expression for a Java method called before a bounded task flow is entered. It is used when the caller needs to know when a bounded task flow is being initiated.
- After listener: An EL expression for a Java method called after a bounded task flow returns. It is used when the caller needs to know when a bounded task flow exits and control flow returns to the caller.

If multiple before listeners or multiple after listeners are specified, they are called in the order in which they appear in the source document for the unbounded or bounded task flow. A task flow call activity can have only have before listener and one after listener.

In order for the task flow call after listeners to be called, control flow must return from the bounded task flow using a control flow rule. If an end user leaves a bounded task flow using the browser back button or other URL, task flow call after listeners will not be called. You must use a bounded task flow finalizer to release all acquired resources and perform cleanup of a bounded task flow that the end user left by clicking a browser back button.

Before you begin:

It may be helpful to have an understanding of how a task flow call activity interacts with a task flow. For more information, see Using Task Flow Call Activities.

You may also find it helpful to understand other ADF functionality and features. For more information, see Additional Functionality for Task Flow Activities.

To specify a before or after listener on a task flow call activity:

1. In the Applications window, expand the **WEB-INF** node and double-click the calling task flow.
2. In the diagram for the calling bounded task flow, select the task flow call activity.
3. In the Properties window, expand the Listeners section and open the Expression Builder for either the **Before Listener** property or the **After Listener** property. To open the Expression Builder, choose **Method Expression Builder** from the context menu that invokes when you click the icon that appears when you hover over the property field.

   Use the Expression Builder to navigate to the Java class that contains the method for the listener. For example, you might create an EL expression similar to the following:

   ```
   #{pageFlowscope.managedBean.methodListener}
   ```
4. Click **OK**.

What Happens When You Add a Task Flow Call Activity

After you add a task flow call activity to a task flow diagram, you must specify a reference to the called bounded task flow using one of the methods described in How to Call a Bounded Task Flow Using a Task Flow Call Activity. For example, if you drop an existing bounded task flow on the task flow call activity, JDeveloper generates the task flow reference automatically. The task flow reference is used to invoke the called bounded task flow.

If the task flow reference is static, each task flow reference consists of:

- **Document**: The name of the XML source file containing the ID of the called bounded task flow. If **Document** is not specified, `adfc-config.xml` is assumed.

The document is a physical XML file and must be accessible via MDS.

- **ID**: The bounded task flow ID contained in the XML source file for the called bounded task flow. For example, a called task flow might have an ID called `targetFlow`. The same XML source file can contain multiple bounded task flows, each bounded task flow identified by a unique ID.

> **Note:**
>
> If you use JDeveloper to create the bounded task flow, there is only one bounded task flow per document.

The following example contains an example static task flow reference within a task flow call activity. In order to invoke a bounded task flow, you need to know its ID and the name of the file (`<document>`) containing the ID.

```
<adfc-config xmlns="http://xmlns.oracle.com/adf/controller" version="1.2"
id="__1">
...
  <task-flow-definition id="task-flow-definition">
    <default-activity>view1</default-activity>
    <task-flow-call id="taskFlowCall">
      <task-flow-reference>
        <document>/WEB-INF/called-task-flow-definition.xml</document>
        <id>called-task-flow-definition</id>
      </task-flow-reference>
    </task-flow-call>
  </task-flow-definition>
...
</adfc-config>
```

The following example shows the metadata that JDeveloper generates for a dynamic task flow reference within a task flow call activity.

```
<?xml version="1.0" encoding="windows-1252" ?>
<adfc-config xmlns="http://xmlns.oracle.com/adf/controller" version="1.2"
             id="__1">
  <task-flow-definition id="bounded_tf">
    <default-activity id="__2">taskFlowCall1</default-activity>
    <task-flow-call id="taskFlowCall1">
      <dynamic-task-flow-reference id="__3">#{EL_Expression_Retrieve_
                                  TaskflowID}</dynamic-task-flow-reference>
    </task-flow-call>
  </task-flow-definition>
</adfc-config>
```

What Happens at Runtime: How a Task Flow Call Activity Invokes a Task Flow

The ADF Controller performs the following steps when a bounded task flow is called using a task flow call activity:

1. Verifies that the user is authorized to call the bounded task flow.
2. Invokes task flow call activity before listener or listeners, if specified (see How to Specify Before and After Listeners on a Task Flow Call Activity).

3. Evaluates the input parameter values on the bounded task flow.
4. Pushes the called bounded task flow onto the stack and initializes its page flow scope.
5. Sets input parameter values in the called bounded task flow's context.
6. Invokes a bounded task flow initializer method, if one is specified.
7. Executes the bounded task flow's default activity.

Using Task Flow Return Activities

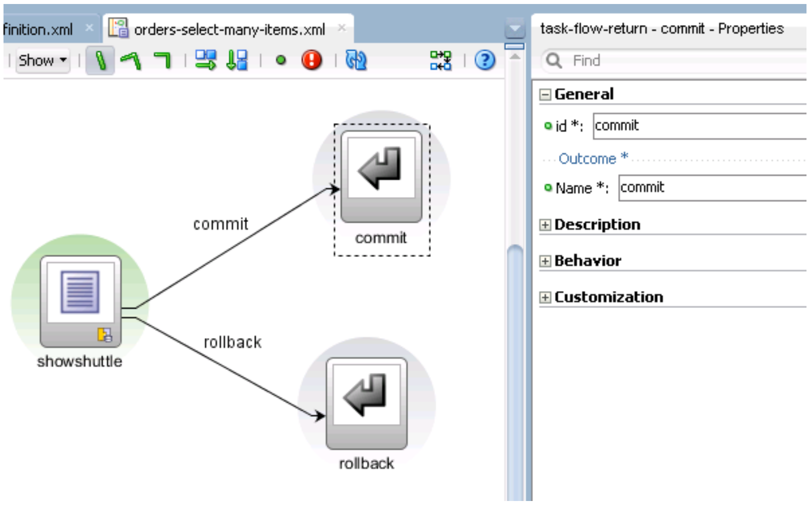
When the ADF task flow call activity is invoked and after it completes its default activity, it should return the control back to the invoked application. The task flow return activity is helpful in identifying and sending the control flow back to the caller application. This activity can be used only within a bounded task flow.

A task flow return activity identifies the point in an application's control flow where a bounded task flow completes and sends control flow back to the caller. You can use a task flow return activity only within a bounded task flow.

A gray circle around a task flow return activity icon indicates that the activity is an exit point for a bounded task flow. A bounded task flow can have zero to many task flow return activities. Figure 24-11 shows the `orders-select-many-items.xml` bounded task flow in the Summit ADF task flow sample application that contains task flow return activities named **commit** and **rollback**.

Each task flow return activity specifies an `outcome` that it returns to the calling task flow. For example, Figure 24-11 shows a task flow return activity in the Summit ADF task flow sample application. The value for the this task flow return activity's Outcome Name property is `commit`.

Figure 24-11 Multiple Task Flow Return Activities



task-flow-return - commit - Properties

General
id *: commit
Outcome *
Name *: commit

The `outcome` returned to an invoking task flow depends on the end user action. You can configure control flow cases in the invoking task flow to determine the next action by the invoking task flow. Set the **From Outcome** property of a control flow case to the value returned by the task flow return activity's `outcome` to invoke an action based on that outcome. This determines control flow upon return from the called task flow.

Set a value for the **Restore Save Point** property to specify if model changes made in a bounded task flow are saved or discarded when the bounded task flow exits by using a task flow return activity. Set to `true` to roll back transactions to the ADF Model save point that was created when the Fusion web application first entered the bounded task flow. The default value is `false`. You can specify a value for this property only if the bounded task flow on which the task flow return activity is located was entered without starting a new transaction. See How to Enable Transactions in a Bounded Task Flow .

How to Add a Task Flow Return Activity to a Bounded Task Flow

You drag a task flow return activity from the Components window and drop it on the diagram for the bounded task flow.

Before you begin:

It may be helpful to have an understanding of how a task flow return activity interacts with a task flow. For more information, see Using Task Flow Return Activities .

You may also find it helpful to understand other ADF functionality and features. For more information, see Additional Functionality for Task Flow Activities.

To add a task flow return activity to a bounded task flow:

1. In the Applications window, expand the **WEB-INF** node and double-click the bounded task flow to which you want to add a task flow return activity.
2. In the ADF Task Flow page of the Components window, from the Components panel, in the Activities group, drag and drop a **Task Flow Return** onto the diagram.
3. In the Properties window, expand the **General** section, and enter an outcome in the **Name** field.

   The task flow return activity returns this outcome to the calling task flow when the current bounded task flow exits. You can specify only one outcome per task flow return activity. The calling task flow should define control flow rules to handle control flow upon return. For more information, see How to Add a Control Flow Rule to a Task Flow.
4. Expand the **Behavior** section and in the **Reentry** dropdown list, choose one of the following options:
   - **<default> (not outcome dependent)**: Reentry is not dependent on an outcome.
   - **reentry-allowed**: Reentry is allowed on any view activity within the bounded task flow.
   - **reentry-not-allowed**: Reentry of the bounded task flow is not allowed.

     If you specify **reentry-not-allowed** on a bounded task flow, an end user can still click the browser back button and return to a page within the bounded task flow. However, if the user does anything on the page such as clicking a button, an exception (for example, `InvalidTaskFlowReentry`) is thrown indicating the bounded task flow was reentered improperly. The actual reentry condition is identified upon the submit of the reentered page.

For more information about reentering a bounded task flow, see Reentering Bounded Task Flows.

Your selection defines the default behavior when the bounded task flow is reentered by an end user clicking a browser's Back button. This selection applies only if **reentry-outcome-dependent** has been set on the bounded task flow where the task flow return activity is located. For more information, see Reentering Bounded Task Flows.

5. In the **End Transaction** dropdown list, choose one of the following options:
   - **commit**: Select to commit the existing transaction to the database.
   - **rollback**: Select to roll back the transaction to what it was on entry of the called task flow. This has the same effect as canceling the transaction, since it rolls back a new transaction to its initial state when it was started on entry of the bounded task flow.

     If you do not specify commit or rollback, the transaction is left open to be closed by the calling bounded task flow.
6. In the **Restore Save Point** dropdown list, select `true` when either of the following conditions apply:
   - If **Always Begin New Transaction** (`new-transaction`) is not selected on the bounded task flow on which the task flow return activity is located
   - ADF model changes made within a bounded task flow should be discarded when exiting using the task flow call activity. The transaction is rolled back to the save point created on entry of the bounded task flow.

   For more information, see How to Enable Transactions in a Bounded Task Flow .

What Happens When You Add a Task Flow Return Activity to a Bounded Task Flow

JDeveloper generates metadata entries in the source file of the bounded task flow for the changes that you configure for the task flow return activity in the Properties window. The following example shows entries that appear in the `orders-select-many-items.xml` task flow of the Summit ADF task flow sample application.

```
<task-flow-return id="commit">
     <outcome>
       <name>commit</name>
       <commit/>
     </outcome>
   </task-flow-return>
   <task-flow-return id="rollback">
     <outcome>
       <name>rollback</name>
       <rollback/>
     </outcome>
   </task-flow-return>
```

Using Task Flow Activities with Page Definition Files

A page definition file defines the binding objects that populate data in the ADF application at runtime. You can use the page definition files to bind the ADF task flow activities to data controls.

Page definition files define the binding objects that populate data at runtime. They are typically used in a Fusion web application to bind page UI components to data controls. A number of task flow activities can also use page definition files to bind to data controls. These are:

- Method call

  You can drag and drop a data control operation from the Data Controls panel onto a task flow to create a method call activity or onto an existing method call activity. In both cases, the method call activity binds to the data control operation.

- Router

  Associating a page definition file with a router activity creates a binding container. At runtime, this binding container makes sure that the router activity references the correct binding values when it evaluates the router activity cases' EL expressions. Each router activity case specifies an `outcome` to return if its EL expression evaluates to `true`. For this reason, only add data control operations to the page definition file that evaluate to `true` or `false`.

- Task flow call

  Associating a page definition file with a task flow call activity creates a binding container. At runtime, the binding container is in context when the task flow call activity passes input parameters. The binding container makes sure that the task flow call activity references the correct values if it references binding values when passing input parameters from a calling task flow to a called task flow.

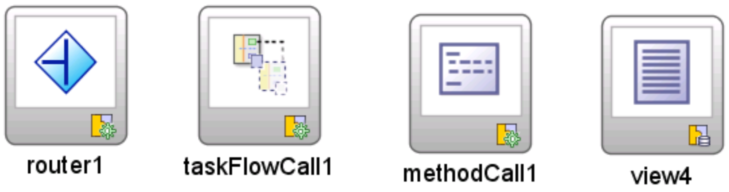
- View

  You cannot directly associate a view activity with a page definition file. Instead, you associate the page that the view activity references.

If you right-click any of the above task flow activities (except view activity) in the diagrammer for a task flow, JDeveloper displays an option on the context menu that enables you to create a page definition file (**Create Page Definition**) if one does not yet exist. If a page definition file does exist, JDeveloper displays a context menu option for all task flow activities to go to the page definition file (**Go to Page Definition**). JDeveloper also displays a context menu option (**Edit Binding**) when you right-click a method call activity that is associated with a page definition file.

A task flow activity that is associated with a page definition file displays an icon in the lower-right section of the task flow activity icon. Figure 24-12 shows an example for each of the task flow activities.

Figure 24-12 Task Flow Activities Associated with Page Definition Files



How to Associate a Page Definition File with a Task Flow Activity

JDeveloper provides a context menu option that you can access from the task flow activity. You use this context menu option to associate the task flow activity with a page definition file.

Before you begin:

It may be helpful to have an understanding of how task flow activities use page definition files. For more information, see Using Task Flow Activities with Page Definition Files.

You may also find it helpful to understand other ADF functionality and features. For more information, see Additional Functionality for Task Flow Activities.

To associate a page definition file with a task flow activity:

1. In the Applications window, expand the **WEB-INF** node and double-click the task flow that contains the task flow activity with which you want to associate a page definition file.
2. In the diagram, right-click the task flow activity for which you want to create a page definition file and choose **Create Page Definition**.
3. In the resulting page definition file, add the bindings that you want your task flow activity to reference at runtime.

   For more information about page definition files, see Working with Page Definition Files.

What Happens When You Associate a Page Definition File with a Task Flow Activity

At design time, JDeveloper generates a page definition file for the task flow activity. The filename of the page definition file comprises the originating task flow and either the name of the task flow activity or the data control operation to invoke. For example, `taskflowName_taskflowName_methodCall1PageDef.xml` or `taskflowName_taskflowName_CreateInsertPageDef.xml`.

JDeveloper also generates an EL expression from the task flow activity to the binding in the created page definition file. The following example shows a method call activity that references a `CreateInsert` action binding.

```
<method-call id="CreateInsert">
      <method>#{bindings.CreateInsert.execute}</method>
      <outcome>
        <fixed-outcome>CreateInsert</fixed-outcome>
      </outcome>
    </method-call>
```

At runtime, a binding container makes sure that a task flow activities' EL expressions reference the correct value.

25
Using Parameters in Task Flows

This chapter describes how to configure parameters in view activities and in ADF bounded task flows that you create in the Fusion web application. Using these capabilities allows you to manipulate data in task flows, share data between task flows across one or more transactions, and configure return values to return from one task flow to another task flow.
This chapter includes the following sections:

About Using Parameters in Task Flows

When used with an ADF task flow, parameters allow you to expand the horizon of the task flow from its default activity to multiple activities. For example, you can manipulate the data in task flows and share data between multiple task flows.

A task flow´s ability to accept input parameters and return parameter values allow you to manipulate data in task flows and share data between task flows. Using these abilities, you can optimize the reuse of task flows in your Fusion web application.

You can use view activity input page parameters as aliases. The alias allows you to map bounded task flow input parameters to page parameters. The view activity input page parameters map managed beans and any information available to the calling task flow to a managed bean on the page itself. To pass values out of view activities, store values in pageFlow scope or managed beans. For information about using view activities in a task flow, see Using View Activities.

For example, a page might specify `#{pageFlowScope.empNo}` as a page parameter and a bounded task flow might specify `#{pageFlowScope.employeeID}` as the value of an input parameter definition.

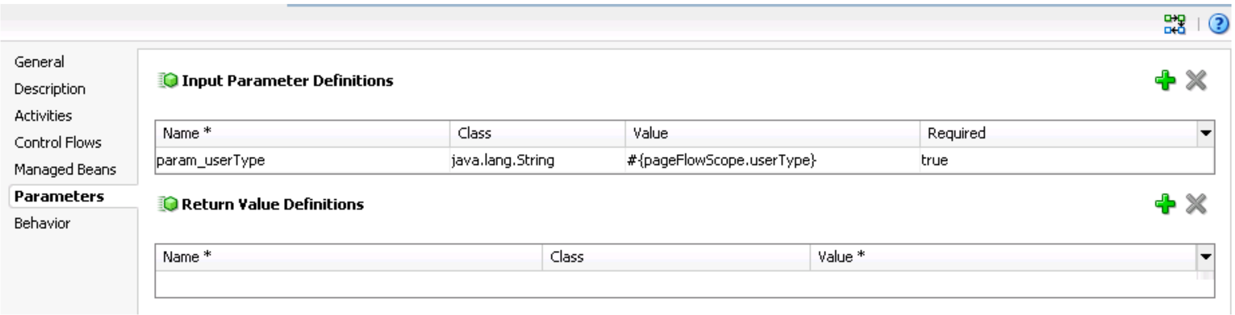
The `from-value` on the view activity input page parameter would be `#{pageFlowScope.employeeID}` and the `to-value` would be `#{pageFlowScope.empNo}`. This enables reuse of both the page definition and bounded task flow because you do not have to redefine parameters for every context in which each is used.

Other values contained within the task flow can be mapped to input page parameters, not just bounded task flow input parameter definition values.

Task Flow Parameters Use Cases and Examples

Figure 25-1 shows a task flow that defines an input parameter definition to hold information about a user in a pageFlow scope.

Figure 25-1 Task Flow Defining an Input Parameter



Input Parameter Definitions

Name *	Class	Value	Required
param_userType	java.lang.String	#{pageFlowScope.userType}	true

Return Value Definitions

Name *	Class	Value *

Additional Functionality for Task Flows Using Parameters

You may find it helpful to understand other **Oracle ADF** features before you configure or use a task flow with parameters. Additionally, you may want to read about what you can do with your configured task flows. Following are links to other functionality that may be of interest.

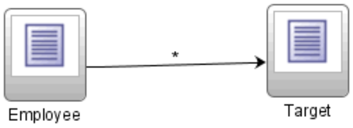
- Data controls can be shared between task flows. For more information about sharing data controls, see Sharing Data Controls Between Task Flows .
- A task flow can access a managed bean that is registered with it. For more information about managed beans and task flows, see Using a Managed Bean in a Fusion Web Application.
- Bounded task flows can be secured by defining the privileges that are required for someone to use them. For more information, see Enabling ADF Security in a Fusion Web Application.

Passing Parameters to a View Activity

In the cases where you want to pass the value from one activity to another without modifying the value, the ADF task flow view activity is useful. You can either pass a parameter to a value or a value to a managed bean using an EL expression or a literal expression.

Figure 25-2 illustrates how to configure an input page parameter mapping. You can pass a parameter to the Employee activity as a `pageFlowScope` value or a value on a managed bean. You can pass a parameter to the Employee activity using an EL expression or a literal expression. The Employee activity, in turn, can pass a value to the Target activity by specifying the value it wants to pass to the Target activity in the `to-value` element.

Figure 25-2 Task Flow with Two Activities



How to Pass Parameters to a View Activity

You add one or more input page parameters to the view activity that you want to pass parameters to.

Before you begin:

It may be helpful to have an understanding of the configuration options available to you before you define an input page parameter for a view activity. For more information, see Passing Parameters to a View Activity.

You may also find it helpful to understand functionality that can be added using other task flow features. For more information, see Additional Functionality for Task Flows Using Parameters.

To define an input page parameter for a view activity:

1. In the Applications window, double-click the task flow that contains the view activity to which you want to pass a parameter.
2. Select the view activity in the diagram for the task flow and, in the Properties window, expand the **Page Parameters** category.
3. In the **Input Page Parameters** list, click the **Add** icon and enter values as follows:
   - **From Value**: write an EL expression that, when evaluated, specifies where the view activity retrieves the value for the input page parameter. For example, write an EL expression similar to the following:

     ```
     #{pageFlowScope.EmployeeID}
     ```

     You can configure the view activity to retrieve a value that is input to the task flow if you write an EL expression for **From Value** that corresponds to the value for the input parameter that you defined for the task flow, as described in How to Pass an Input Parameter to a Bounded Task Flow.
   - **To Value**: write an EL expression, that when evaluated, specifies where the page associated with the view activity can retrieve the value of the input page parameter. For example, write an EL expression similar to the following:

     ```
     #{pageFlowScope.EmployeeNo}
     ```

What Happens When You Pass Parameters to a View Activity

JDeveloper writes entries to the source file of the task flow at design time, as illustrated in the following example.

```
<view id="reducedAccess">
 <page>/secured/Information.jsf</page>
      <input-page-parameter>
        <from-value>#{res['infoUsage.reducedAccess.messageHeader']}</from-value>
        <to-value>#{pageFlowScope.infoPageHeaderText}</to-value>
      </input-page-parameter>
      <input-page-parameter>
        <from-value>#{res['infoUsage.reducedAccess.messageHeader']}</from-value>
        <to-value>#{pageFlowScope.infoPageMsg}</to-value>
      </input-page-parameter>
      <input-page-parameter>
        <from-value>info</from-value>
        <to-value>#{pageFlowScope.infoPageType}</to-value>
```

```
        </input-page-parameter>
    </view>
```

At runtime, the view activity retrieves the value of the input parameter from the location you specified in the `<from-value>` element. The view activity makes the value of the parameter available to its associated page in the location you specified in the `<to-value>` element.

What You May Need to Know About Specifying Parameter Values

You can specify parameter values using standard EL expressions if you call a bounded task flow using a task flow call activity or you render the bounded task flow in an ADF region or an ADF dynamic region. For example, you can specify parameters using the following syntax for EL expressions:

- `#{bindings.bindingId.inputValue}`
- `#{CustomerBean.zipCode}`

The following example shows the metadata for a task flow binding that renders in an ADF region.

```
<taskFlow id="Department1" taskFlowId="/WEB-INF/Department.xml#Department"
              xmlns="http://xmlns.oracle.com/adf/Controller/binding">
    <parameters>
        <parameter id="DepartmentId" value="#{bindings.DepartmentId.inputValue}"
                   xmlns="http://xmlns.oracle.com/adfm/uimodel"/>
     </parameters>
</taskFlow>
```

Appending `inputValue` to the EL expression makes sure that you assign the parameter the value of the binding rather than the actual binding object.

Passing Parameters to a Bounded Task Flow

From an ADF task flow, you can call a bounded task flow and pass values to it. The bounded task flow can accept the values through input parameters or from a task flow binding. You must specify one or more input parameters and parameter definitions in the task flow invoking the bounded task flow.

A called bounded task flow can accept input parameters from the task flow that calls it or from a task flow binding. To pass an input parameter to a bounded task flow, you specify one or more:

- Input parameters on the task flow call activity in the calling task flow

  Input parameters specify where the calling task flow stores parameter values.
- Input parameter definitions on the called bounded task flow

  Input parameter definitions specify where the called bounded task flow can retrieve parameter values at runtime.

Specify the same name for the input parameter that you define on the task flow call activity in the calling task flow and the input parameter definition on the called bounded task flow. Do this so you can map input parameter values to the called bounded task flow.

If you do not specify an EL expression to reference the value of the input parameter, the EL expression for `value` defaults to the following at runtime:

```
#{pageFlowScope.parmName}
```

where *`parmName`* is the value you entered for the input parameter name.

In an input parameter definition for a called bounded task flow, you can specify an input parameter as required. If the input parameter does not receive a value at runtime or design time, the task flow raises a warning in a log file of the Fusion web application that contains the task flow. An input parameter that you do not specify as required can be ignored during task flow call activity creation.

Task flow call activity input parameters can be passed by reference or passed by value when calling a task flow using a task flow call activity unless you are calling a task flow in an ADF region. If the task flow renders in an ADF region, the task flow call activity passes the input parameters by reference. By default, primitive types (for example, `int`, `long`, or `boolean`) are passed by value (`pass-by-value`).

The Pass By Value checkbox applies only to objects, not primitives and is used to override the default setting of passing by reference. Mixing the two, however, can lead to unexpected behavior in cases where parameters reference each other. If input parameter A on the task flow call activity is passed by value and if input parameter B on the task flow call activity is passed by reference, and B has a reference to A, the result can be two different instances of A and B.

How to Pass an Input Parameter to a Bounded Task Flow, describes how to pass an input parameter from a calling task flow to a called bounded task flow using a task flow call activity. Although you can pass parameter values from any activity on the calling task flow, the passed parameter in How to Pass an Input Parameter to a Bounded Task Flow contains the value of an input text field on a page in the calling task flow.

If you call a bounded task flow using a URL rather than a task flow call activity, you pass parameters and values on the URL itself. See How to Call a Bounded Task Flow Using a URL.

Instead of explicitly passing data controls as parameters between task flows, you can simply share them by specifying the `data-control-scope` option on the called bounded task flow. For more information, see Sharing Data Controls Between Task Flows .

A called task flow can also return values to the task flow that called it when it exits. For more information about returning values from a bounded task flow, see Configuring a Return Value from a Bounded Task Flow.

How to Pass an Input Parameter to a Bounded Task Flow

You define values on the calling task flow and the called task flow.

Before you begin:

- It may be helpful to have an understanding of the configuration options available to you before you configure a bounded task flow to receive an input parameter. For more information, see Passing Parameters to a Bounded Task Flow.
- You may also find it helpful to understand functionality that can be added using other task flow features. For more information, see Additional Functionality for Task Flows Using Parameters.

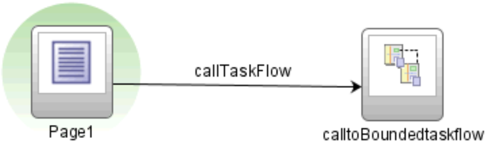
You will need to complete these tasks:

- Create a calling and called task flow

  The calling task flow can be bounded or unbounded. The called task flow must be bounded. For more information about creating task flows, see Creating a Task Flow.
- Add a task flow call activity to the calling task flow

  Figure 25-3 shows an example where the view activity passes control to the task flow call activity.

Figure 25-3 Calling Task Flow



To pass an input parameter to a bounded task flow:

1. In the Applications window, double-click the JSF page that contains an input component where an end user enters a value that gets passed to a bounded task flow as a parameter at runtime.

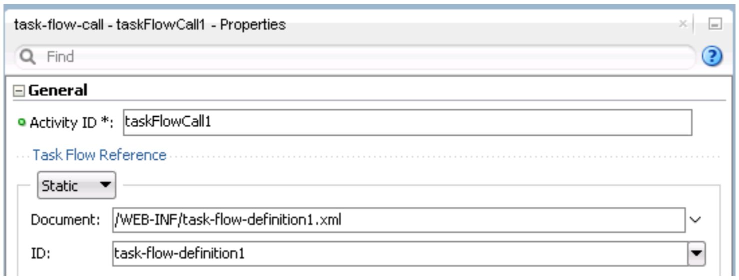
   The JSF page that you open should be referenced by a view activity in the calling task flow.
2. Select an input text component on the JSF page where an end user enters a value at runtime.
3. In the Properties window, expand the **Common** section and enter a value for the input text component in the **Value** field.

   You can specify the value as an EL expression, for example `#{pageFlowScope.inputValue}`.
4. In the Applications window, double-click the task flow that is going to be called.
5. In the editor window, click the **Overview** tab.
6. In the overview editor, click the **Parameters** navigation tab and then click the **Add** icon to define a new entry in the **Input Parameter Definition** section.
7. In the **Name** field, enter a name for the parameter, for example, `inputParm1`.
8. In the **Value** field, enter an EL expression where the parameter value is stored and referenced, for example, `#{pageFlowScope.inputValue}`.
9. In the Applications window, double-click the calling task flow that contains the task flow call activity to invoke the called bounded task flow.
10. In the Applications window, drag the called bounded task flow and drop it on top of the task flow call activity that is located in the diagram of the calling task flow.

    Dropping a bounded task flow on top of a task flow call activity in a diagram automatically creates a task flow reference to the bounded task flow. As shown in Figure 25-4, the Properties window for the task flow call activity, the task flow reference contains the bounded task flow ID and a document name. The bounded task flow ID (`id`) is an attribute of the bounded task flow's `<task-flow-`

definition> element. The document name points to the source file for the task flow that contains the ID.

Figure 25-4 Task Flow Reference


task-flow-call - taskFlowCall1 - Properties

General

Activity ID *: taskFlowCall1

Task Flow Reference

Static

Document: /WEB-INF/task-flow-definition1.xml

ID: task-flow-definition1

11. In the Properties window for the task flow call activity, expand the **Parameters** section to view the **Input Parameters** section.
12. Enter a name that identifies the input parameter.

    Because you dropped the bounded task flow on a task flow call activity having defined input parameters, the name should already be specified. You must keep the same input parameter name.
13. Enter a parameter value, for example, `#{pageFlowScope.parm1}`.

    The value on the task flow call activity input parameter specifies where the calling task flow stores parameter values.

    The value on the input parameter definition for the called task flow specifies where the value will be retrieved from for use within the called bounded task flow once it is passed.
14. At runtime, the called task flow is able to use the input parameter. Since you specified `pageFlowScope` as the `value` in the input parameter definition for the called task flow, you can use the parameter value anywhere in the called bounded task flow. For example, you can pass it to a view activity on the called bounded task flow. For more information, see What Happens When You Pass Control Between View Activities.

What Happens When You Pass an Input Parameter to a Bounded Task Flow

JDeveloper writes entries to the source files for the calling task flow and called task flow based on the values that you select. Example 25-1 shows an input parameter definition specified on a a bounded task flow.

Example 25-2 shows the input parameter metadata for the task flow call activity that calls the bounded task flow shown in Example 25-1. At runtime, the task flow call activity calls the bounded task flow and passes it the value specified by its `value` element.

Example 25-1 Input Parameter Definition

```
<task-flow-definition id="sourceTaskflow">
...
   <input-parameter-definition>
      <name>inputParameter1</name>
      <value>#{pageFlowScope.parmValue1}</value>
      <class>java.lang.String</class>
   </input-parameter-definition>
...
</task-flow-definition>
```

Example 25-2   Input Parameter on Task Flow Call Activity

```
<task-flow-call id="taskFlowCall1">
...
    <input-parameter>
      <name>inputParameter1</name>
      <value>#{pageFlowScope.newCustomer}</value>
      <pass-by-value/>
    </input-parameter>
...
</task-flow-call>
```

Configuring a Return Value from a Bounded Task Flow

A bounded task flow can be invoked from an ADF task flow and values can be passed to it. The bounded task flow, upon processing the values submitted, returns a parameter value to the task flow that has called it. The return value is in addition to the outcome activity that the bounded task flow returns to the calling task flow. You must add return value definitions in the bounded task flow and return values on the calling task flow.

You can configure a bounded task flow to return a parameter value to the task flow that calls it. The value that the bounded task flow returns is in addition to the outcome that it returns to the caller when the bounded task flow invokes a task flow return activity, as described in Using Task Flow Return Activities . To return a value, you must configure:

- Return value definitions on the called bounded task flow

  The return value definition specifies where you store the value that you want to return from the called bounded task flow.

- Return values on the task flow call activity in the calling task flow to identify where the calling task flow can find the returned value

You can configure the calling task flow to ignore return value definition from the called task flow by not identifying any return values on the task flow call activity in the calling task flow.

The task flow call activity returns values by reference. For this reason, you do not need to make a copy of the values that you want to return to the calling task flow.

How to Configure a Return Value from a Bounded Task Flow

You configure a return value definition on the called task flow and add a parameter to the task flow call activity in the calling task flow that retrieves the return value at runtime.

Before you begin:

It may be helpful to have an understanding of the interaction between the calling task flow and the called task flow. For more information, see Configuring a Return Value from a Bounded Task Flow.

You may also find it helpful to understand functionality that can be added using other task flow features and parameters. For more information, see Additional Functionality for Task Flows Using Parameters.

You will need to complete this task:

Create a bounded or unbounded task flow (calling task flow) and a bounded task flow (called task flow). For more information, see Creating a Task Flow.

To configure a return value from a called bounded task flow:

1. In the Applications window, double-click the called task flow.
2. In the editor window, click the **Overview** tab.
3. In the overview editor, click the **Parameters** navigation tab.
4. In the Parameters page, click the **Add** icon for the **Return Value Definitions** section and add values as follows to define a return value:
   - **Name**: Enter a name to identify the return value. For example, `returnValue1`.
   - **Class**: Enter a Java class that defines the data type of the return value. The default value is `java.lang.String`.
   - **Value**: Enter an EL expression that specifies where to read the return value from. For example, enter an EL expression similar to the following:

     ```
     #{pageFlowScope.ReturnValueDefinition}
     ```
5. In the Applications window, double-click the calling task flow.
6. In the ADF Task Flow page of the Components window, from the Components panel, in the Activities group, drag and drop a task call activity onto the diagram.
7. In the Properties window for the task flow activity, expand the **Parameters** section, click the **Add** icon next to the **Return Values** entry and add values as follows to define a return value:
   - **Name**: Enter a name to identify the return value. For example, `returnValue1`.

     The value you enter must match the value you entered for the **Name** field when you defined the return value definition in Step 4.
   - **Value**: Enter an EL expression that specifies where to store the return value. For example, enter an EL expression similar to the following:

     ```
     #{pageFlowScope.ReturnValueDefinition}
     ```

     The value you enter must match the value you entered for the **Value** field when you defined the return value definition in Step 4.

What Happens When You Configure a Return Value from a Bounded Task Flow

At design time, JDeveloper writes entries to the source files for the task flows that you configured. The following example shows an entry that JDeveloper writes to the source file for the calling task flow.

```
<task-flow-call id="taskFlowCall1">
      <return-value id="__3">
        <name id="__4">returnValue1</name>
        <value id="__2">#{pageFlowScope.ReturnValueDefinition}</value>
```

```
      </return-value>
    </task-flow-call>
```

The following example shows entries that JDeveloper writes to the source file for the called task flow.

```
<return-value-definition id="__2">
      <name id="__3">returnValue1</name>
      <value>#{pageFlowScope.ReturnValueDefinition}/</value>
      <class>java.lang.String</class>
 </return-value-definition>
```

At runtime, the called task flow returns a value. If configured to do so, the task flow call activity in the calling task flow retrieves this value.

26

Using Task Flows as Regions

This chapter describes how to render ADF task flows in pages or page fragments using ADF regions in your Fusion web application. Key features such as the ability to specify parameters, how to refresh or activate an ADF region, navigate to a parent page, and create dynamic ADF regions are also described.
This chapter includes the following sections:

About Using Task Flows in ADF Regions

An ADF page fragment or a JSF page with af:region tag is the ADF region where you can execute a bounded task flow. You can use these ADF regions to add pieces of application functionality in the bounded task flow and reuse this packaged functionality throughout the application. The ADF regions are not dependent on a parent page.

You can execute a bounded task flow in a JSF page or page fragment (`.jsff`) by using an ADF region. A primary reason for executing a bounded task flow as an ADF region is reuse. You can isolate specific pieces of application functionality in a bounded task flow and an ADF region in order to reuse it throughout the application. You can extract, configure, and package application functionality within a bounded task flow so that it can be added to other pages using an ADF region. ADF regions can be reused wherever needed, which means they are not dependent on a parent page. This also means that you can isolate the presentation of the parent pages from the ADF region; menus, buttons, and navigation areas are not affected by what is displayed in the ADF region. If you modify a bounded task flow, the changes apply to any ADF region that uses the task flow.

An ADF region comprises the following:

- An `af:region` tag that appears in the page or page fragment where you render the region
- An instance object that implements `RegionModel` from the following package:

  `oracle.adf.view.rich.model`

  For more information about `RegionModel`, see the *Java API Reference for Oracle ADF Faces*
- One or other of the following:
  - A task flow binding (`taskFlow`) in the page definition that identifies the bounded task flow to use in the ADF region
  - A multi task flow binding (`multiTaskFlow`) in the page definition that identifies a list of bounded task flows to use in the ADF region

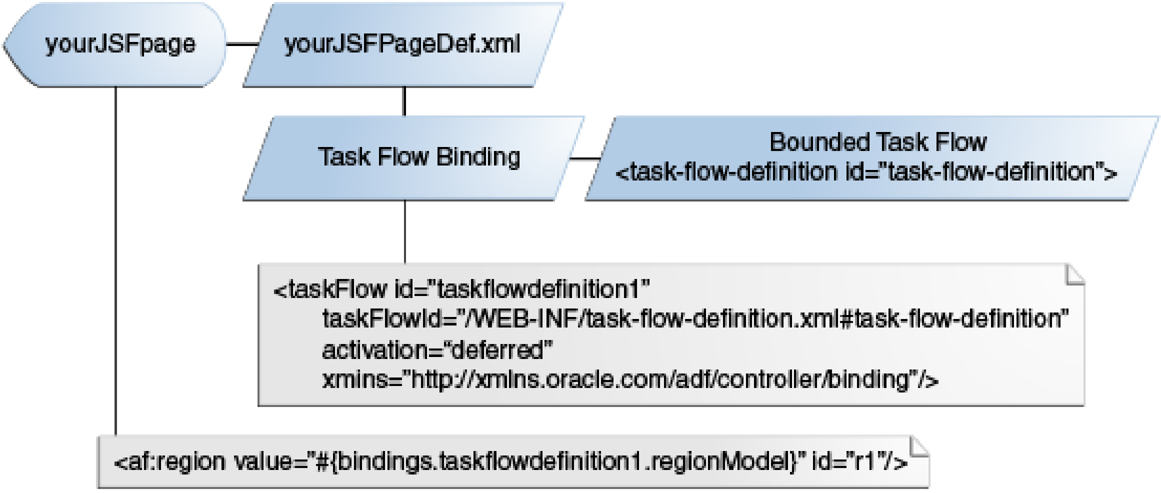
When first rendered, the ADF region's content is that of the first view activity in the bounded task flow. The view activities used in the bounded task flow must be associated with page fragments, not pages.

You can pass values to the ADF region using task flow binding input parameters or contextual events. In addition, you can configure the task flow binding's `parametersMap` property to determine what input parameters the task flow binding passes from the bounded task flow to the ADF region.

ADF regions can be configured so that you determine when the region activates or refreshes. You can also configure an ADF region and a bounded task flow so that navigation control stops in the bounded task flow and passes to the page that contains the ADF region. Finally, you can create dynamic regions (ADF dynamic regions) where the task flow binding determines at runtime what bounded task flow renders in the region and configure a dynamic region link so that end users can change the bounded task flow that renders in the ADF dynamic region at runtime.

Figure 26-1 shows how an ADF region references a bounded task flow using a task flow binding in the page definition file of the page that hosts the ADF region.

Figure 26-1   ADF Region Referencing a Bounded Task Flow



```
<taskFlow id="taskflowdefinition1"
      taskFlowId="/WEB-INF/task-flow-definition.xml#task-flow-definition"
      activation="deferred"
      xmins="http://xmins.oracle.com/adf/controller/binding"/>
```

```
<af:region value="#{bindings.taskflowdefinition1.regionModel}" id="r1"/>
```

About Page Fragments and ADF Regions

A page fragment is a JSF document (file extension is `.jsff`) that renders as content in another JSF page. A page fragment should not have more than one root component. Wrapping multiple root components in a single root component helps you to optimize the display performance of the page fragment. In addition, if a page fragment has only one visual root component and a `popup` component (which is invisible to end users until invoked), consider wrapping these components in a single root component. For example, place the `popup` component in a `panelStretchLayout` component's bottom facet with the `bottomHeight` attribute set to `0` pixels.

If a page fragment has more than one root component, the Fusion web application logs a message at runtime, as shown in the following example, where `r1` identifies the ADF region that renders the page fragment.

```
<RegionRenderer> <encodeAll> The region component with id: r1 has detected a page
 fragment with multiple root components. Fragments with more than one root
 component may not display correctly in a region and may have a negative impact
 on performance. It is recommended that you restructure the page fragment to have
 a single root component.
```

Apart from having only one root component element, a page fragment must not contain any of the following tags:

- `<af:document>`
- `<f:view>`
- `<f:form>`
- `<html>`
- `<head>`
- `<body>`

These tags can only appear once in a document and do not support nesting in a JSF page. For example, a page fragment embedded in a page cannot have an `<html>` tag because the JSF page already has one.

The following example shows a simple page fragment. Unlike a JSF page, it contains no `<f:view>` or `<f:form>` tags.

```
<?xml version='1.0' encoding='UTF-8'?>
<ui:composition xmlns:ui="http://java.sun.com/jsf/facelets"
                     xmlns:af="http://xmlns.oracle.com/adf/faces/rich">
  <af:button text="button 1" id="b1"/>
</ui:composition>
```

You must nest a page fragment that you include in another JSF page within a region (`af:region` tag). A bounded task flow that you add to a JSF page as a region cannot call pages; it must call page fragments.

About View Ports and ADF Regions

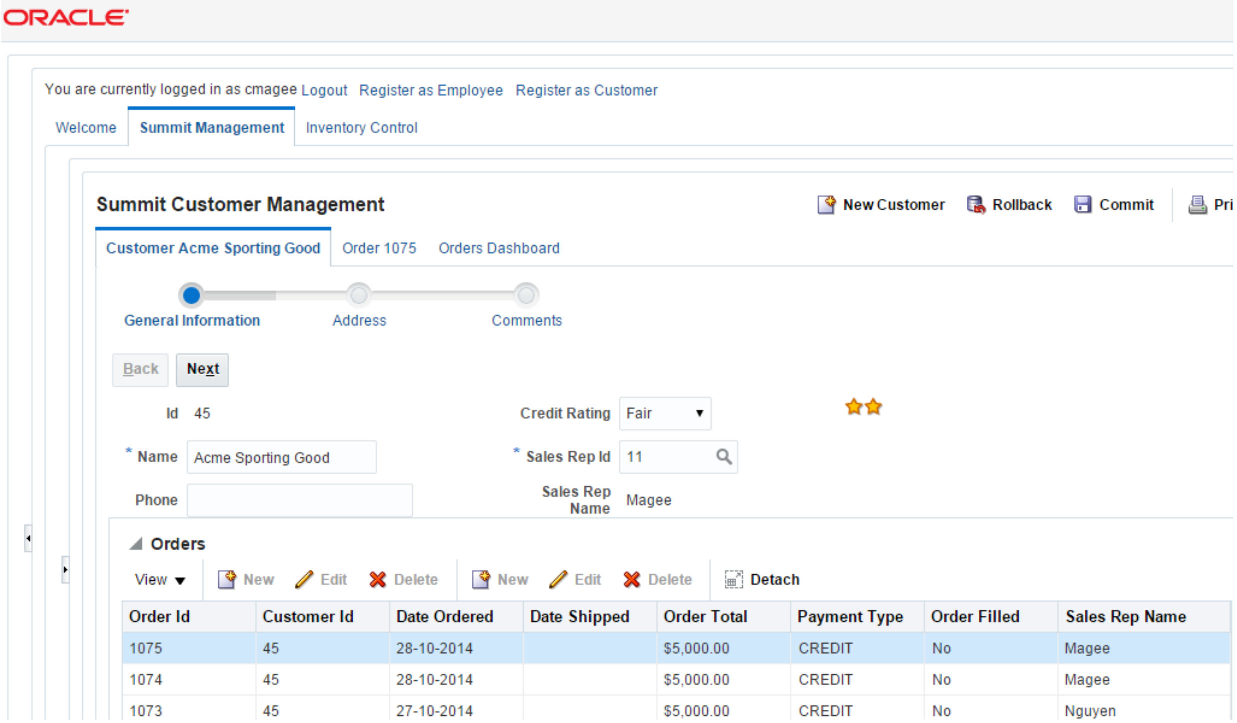
A view port is a display area capable of navigating independently of other view ports. A browser window and an ADF region are both examples of view ports. The root view port displays the main page in a browser window. The root view port may have child view ports, for example, regions on the page, but does not have a parent view port.

Java classes that implement the `ViewPortContext` interface in the `oracle.adf.controller` package get you more information about view ports. For more information, see the *Java API Reference for Oracle ADF Controller*.

Task Flows and ADF Region Use Cases and Examples

Figure 26-2 shows the General Information page fragment (`GeneralInfo.jsff`) from the `edit-customer-task-flow-definition.xml` task flow in the Summit sample application for ADF task flows. This task flow renders in an ADF region and displays information about customers in a series of page fragments that authenticated users can navigate through and edit using the provided controls. For more information about creating this type of region, see Creating an ADF Region.

Figure 26-2 ADF Region in the Summit ADF Task Flow Sample Application



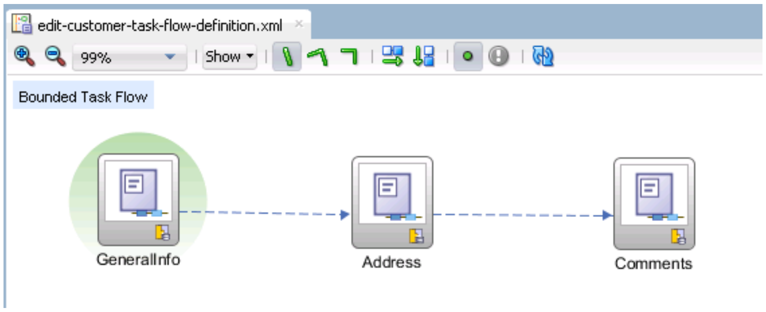
Id 45
Credit Rating Fair
* Name Acme Sporting Good
* Sales Rep Id 11
Phone
Sales Rep Name Magee

Order Id	Customer Id	Date Ordered	Date Shipped	Order Total	Payment Type	Order Filled	Sales Rep Name
1075	45	28-10-2014		$5,000.00	CREDIT	No	Magee
1074	45	28-10-2014		$5,000.00	CREDIT	No	Magee
1073	45	27-10-2014		$5,000.00	CREDIT	No	Nguyen

Figure 26-2 also shows links (labeled **Register as Employee** and **Register as Customer**) that invoke different task flows in the ADF dynamic region that the **showDetailItem** component labeled **Welcome** renders. For more information about creating this type of region, see Creating ADF Dynamic Regions.

Figure 26-3 shows the `edit-customer-task-flow-definition.xml` task flow in JDeveloper. It contains three view activities to display information about customers to end users.

Figure 26-3 Edit Customer Task Flow in ADF Summit Application for ADF Task Flows



Additional Functionality for Task Flows that Render in ADF Regions

You may find it helpful to understand how a task flow that renders in an ADF region interacts with other task flow and ADF functionality. Following are links to other functionality that may be of interest.

- You can set security on a bounded task flow that displays in an ADF region and associated page definitions. If an end user displays a page that contains an ADF region that the user is not authorized to view, the contents of the ADF region do not display. No authentication mechanism is triggered. For more information, see Enabling ADF Security in a Fusion Web Application.
- Task flows can invoke managed beans. For more information about managed beans, see Using a Managed Bean in a Fusion Web Application.
- You can use contextual events to exchange information with a bounded task flow. For more information, see Using Contextual Events.

Creating an ADF Region

You can create an ADF region in a JSF Page or an ADF page fragment by adding af:region tag. The ADF region must contain at least one view activity or one task flow call activity to the page where you want to render the ADF region.

You create an ADF region by dragging and dropping a bounded task flow that contains at least one view activity or one task flow call activity to the page where you want to render the ADF region. This makes sure that the ADF region you create has content to display at runtime.

The bounded task flow's view activities must be associated with page fragments (`.jsff`). If you attempt to drag and drop a bounded task flow that is associated with pages rather than page fragments, JDeveloper does not display the context menu that allows you to create an ADF region. You can convert a bound task flow that uses pages to use page fragments. See How to Convert Bounded Task Flows .

The context menu that appears to create an ADF region presents options to create a nondynamic and a dynamic region. A dynamic region (ADF dynamic region) determines the bounded task flow that it renders at runtime. For information about creating an ADF dynamic region, see Creating ADF Dynamic Regions. You determine at design time the bounded task flow that a nondynamic region (ADF region) displays.

How to Create an ADF Region

Drag a bounded task flow from the Applications window to the page where you want to render an ADF region.

Before you begin:

It may be helpful to have an understanding of the requirements for a bounded task flow that you use in an ADF region. For more information, see Creating an ADF Region.

You may also find it helpful to understand functionality that can be added using other task flow features and ADF region features. For more information, see Additional Functionality for Task Flows that Render in ADF Regions.

You will need to complete these tasks:

- Create a bounded task flow with one or more view activities associated with page fragments or one task flow call activity to a task flow with view activities.

  For more information, see Creating a Task Flow.

- Create a JSF page to host the ADF region.

  For more information, see the "How to Create JSF Pages" section of *Developing Web User Interfaces with Oracle ADF Faces*.

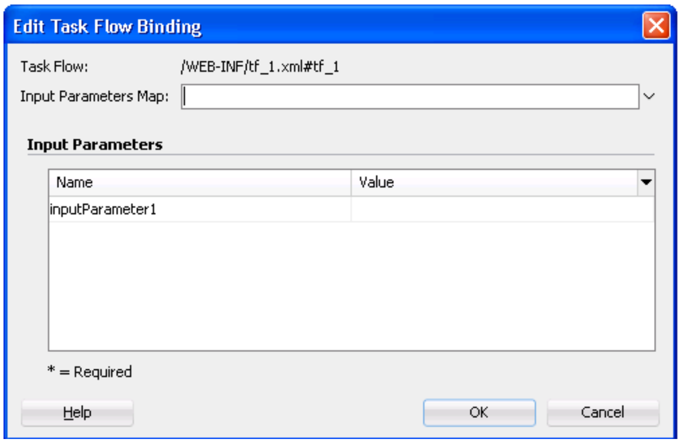
To create an ADF region:

1. In the Applications window, double-click the JSF page where you want to locate the ADF region.
2. From the Applications window, drag and drop the bounded task flow onto the JSF page, and from the context menu, choose **Create** > **Region**.
3. If the Edit Task Flow Binding dialog appears, click **OK**.

   The Edit Task Flow Binding dialog appears if the bounded task flow that you drop on the JSF page has an input parameter defined, as described in How to Pass an Input Parameter to a Bounded Task Flow.

   For more information about specifying parameters for an ADF region, see Specifying Parameters for an ADF Region.

Figure 26-4 Edit Task Flow Binding Dialog for an ADF Region



Edit Task Flow Binding

Task Flow: /WEB-INF/tf_1.xml#tf_1

Input Parameters Map:

Input Parameters

Name	Value
inputParameter1	

* = Required

4. In the Structure window, right-click the node for the ADF region that you added (**af:region**) and choose **Go to Properties**.
5. Review or modify (as appropriate) the following fields which JDeveloper automatically populates with default values in the Properties window for the ADF region:
   - **Id**: An ID that the JSF page uses to reference the ADF region, for example, `r1`.
   - **Value**: An EL reference to the ADF region model, for example, `#{bindings.region1.regionModel}`. This is the region model that describes the behavior of the region.
   - **Rendered**: If `true` (the default value), the ADF region renders when the JSF page renders.
6. For information about how to map parameters between the view activity associated with the JSF page and the ADF region, see Specifying Parameters for an ADF Region.

What Happens When You Create an ADF Region

When you drop a bounded task flow onto a JSF page to create an ADF region, JDeveloper adds an `af:region` tag to the page. The `af:region` tag references an object that implements `RegionModel`. The following example shows the `af:region` tag that appears in the `Customers.jsff` page fragment of the Summit ADF task flow sample application.

```
<af:region value="#{bindings.editcustomertaskflowdefinition1.regionModel}"
                    id="r2"/>
```

JDeveloper also adds a task flow binding to the page definition file of the page that hosts the ADF region. The following example shows a sample of the metadata that JDeveloper adds. The task flow binding provides a bridge between the ADF region and the bounded task flow. It binds a specific instance of an ADF region to its associated bounded task flow and maintains all information specific to the bounded task flow. The `taskFlowId` attribute specifies the directory path and the name of the source file for the bounded task flow.

```
<taskFlow id="editcustomertaskflowdefinition1"
    taskFlowId="/WEB-INF/flows/edit-customer-task-flow-definition.xml#edit
    -customer-task-flow-definition"
    activation="deferred"
    xmlns="http://xmlns.oracle.com/adf/controller/binding"/>
```

The bounded task flow preserves all of the data bindings when you drop it on the JSF page. At runtime, a request for a JSF page containing an ADF region initially handles like any other request for a JSF page. As the JSF page definition executes, data loads in to the JSF page. When the component tree for the parent JSF page encounters the `<af:region>` tag, it executes it in order to determine the first page fragment of content to display. Once it determine the first page fragment of content, it adds the appropriate UI components from the page fragment to the component tree for the parent JSF page.

The task flow binding creates an object for its task flow that implements the following interface in order to get the current view activity:

```
oracle.adf.controller.ViewPortContext
```

The task flow binding's `taskFlowId` attribute can also reference an EL expression that evaluates to one of the following:

- `java.lang.String`
- `oracle.adf.controller.TaskFlowId`

You use this capability if you create an ADF dynamic region. For more information, see Creating ADF Dynamic Regions.

Specifying Parameters for an ADF Region

You may need to transfer values to the ADF region for the input parameters defined for a bounded task flow in that region. For this, you can the input parameters (defined in the ADF region) to the task flow binding the ADF region. You can also access the input parameters in the memory scopes, managed beans, or the ADF binding layer by using EL expressions in the ADF region.

You can make input parameters that you defined for a bounded task flow available to an ADF region by adding them to the task flow binding that the ADF region references. Use EL expressions to reference input parameters available in memory scopes, managed beans, or the ADF binding layer.

Specifying input parameters is one method of providing information to an ADF region. An alternative method is to use contextual events. The nature of the information that you want to provide to the ADF region determines the method you choose to provide the information. For example, choose:

- Input parameters if the required information is at the beginning of the task flow and a change in the value of this information requires a restart of the task flow.

  For example, you have a page that contains a table of employees. An ADF region on that page contains a task flow to enroll a selected employee in a benefits program. A change in the selected employee requires that you restart the benefits enrollment task flow from the beginning for the newly selected employee. Using input parameters to the task flow is the right decision for this use case.

  You can pass input parameters by reference or by value. If you pass by reference, an update on the main page for the selected employee's information, such as last name, is automatically reflected in the task flow running in the ADF region without restarting the task flow.

- Contextual events if you can only determine the information to exchange after the start of a task flow and a change in the information does not require a restart of the task flow. For example, the Inventory Control tab in the Summit ADF task flow sample application uses contextual events to display the appropriate chart for product inventory in one region in response to the product that a user selects in another region. For more information, see Using Contextual Events.

For information about creating an ADF region and adding task flow bindings, see Creating an ADF Region. For information about how to define an input parameter for a bounded task flow, see Passing Parameters to a Bounded Task Flow.

How to Specify Parameters for an ADF Region

Use EL expressions to specify parameters available in memory scopes, managed beans, or the ADF binding layer as input for the ADF region.

Before you begin:

It may be helpful to have an understanding of the requirements for specifying parameters for an ADF region. For more information, see Specifying Parameters for an ADF Region.

You may also find it helpful to understand functionality that can be added using other task flow features and ADF region features. For more information, see Additional Functionality for Task Flows that Render in ADF Regions.

To specify input parameters for an ADF region:

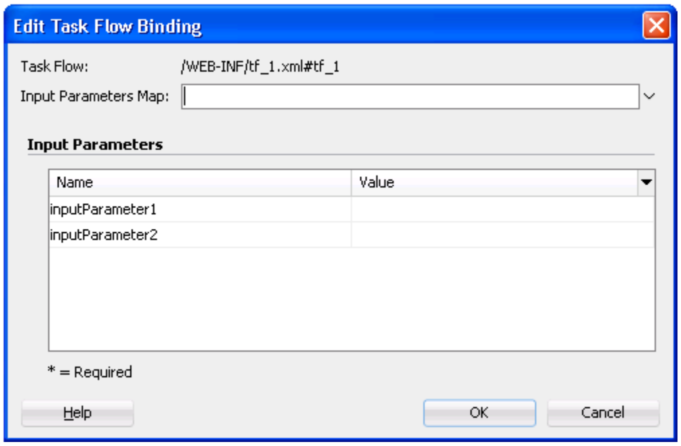
1. In the Applications window, right-click the JSF page that holds the ADF region and choose **Go to Page Definition**.
2. In the overview editor, click the **Bindings and Executables** navigation tab, select the task flow binding for which you want to specify parameters and click the **Edit** icon.
3. In the Edit Task Flow Binding dialog, write EL expressions that retrieve the values of each input parameter you want to specify for the ADF region. Note that you must write an EL expression for parameters that you defined as required. For example, write an EL expression similar to the following:

   ```
   #{pageFlowScope.inputParameter1}
   ```

   Figure 26-5 shows the Edit Task Flow Binding dialog where the Input Parameters section lists a number of input parameters that were defined for the bounded task, as described in Passing Parameters to a Bounded Task Flow.

   > **Note:**
   >
   > You can write an EL expression that references a list of input parameters specified in a managed bean using the Input Parameters Map field of the Edit Task Flow Binding dialog. For more information about implementing this functionality, see Specifying Parameters for ADF Regions Using Parameter Maps.

Figure 26-5 Edit Task Flow Binding Dialog



Edit Task Flow Binding

Task Flow: /WEB-INF/tf_1.xml#tf_1

Input Parameters Map:

Input Parameters

Name	Value
inputParameter1	
inputParameter2	

* = Required

4. Click **OK**.

What Happens When You Specify Parameters for an ADF Region

JDeveloper writes entries that are child elements of the `taskFlow` element in the page definition of the JSF page, as illustrated in the following example.

```
<taskFlow id="tflow_tf11"
   taskFlowId="/WEB-INF/tflow_tf1.xml#tflow_tf1"
   activation="deferred"
   xmlns="http://xmlns.oracle.com/adf/controller/binding">
  <parameters>
    <parameter id="inputParameter1"
      value="#{pageFlowScope.inputParameter1}"/>
    <parameter id="inputParameter2"
      value="#{pageFlowScope.inputParameter2}"/>
  </parameters>
</taskFlow>
```

At runtime, the values specified by the EL expression in the value `attribute` are passed to the ADF region.

Specifying Parameters for ADF Regions Using Parameter Maps

A parameter map object on a managed bean specifies the keys mapping to the values that you want to input to the ADF region. As you define the parameters in the parameter maps, the space required to specify them in the task flow binding element or to specify them in multiple task flow bindings in the page definition for the pages is reduced.

In addition (or as an alternative) to listing all the input parameters on the task flow binding, as described in Specifying Parameters for an ADF Region, you can use the `parametersMap` property of the task flow binding to specify a parameter map object on a managed bean. The parameter map object that you reference must be of a type that implements the following interface:

```
java.util.Map
```

The parameter map object that you reference specifies keys that map to the values you want to input to the ADF region. Using this approach reduces the number of `parameter` child elements that appear under the task flow binding (`taskFlow`) element or the multi task flow binding (`multiTaskFlow`) in the page definition for the page. This approach also allows you more flexibility in determining what input parameters pass to the ADF region. In particular, it provides a way to pass parameters to ADF dynamic regions as the number of input parameters can differ between the different task flow bindings in an ADF dynamic region. See Creating ADF Dynamic Regions.

You can configure an ADF region or an ADF dynamic region's task flow binding to reference a parameter map. You can also configure a multi task flow binding to reference a parameter map. Make sure that the name of an input parameter you define for a bounded task flow matches the name of a key that you define in the parameter map object.

If you specify a parameter with the same `id` attribute in a `<parameter>` element and in a `<parametersMap>` element, the `<parametersMap>` element always takes precedence.

This is because the `<parametersMap>` element may need to override the static value of the parameter if you create an ADF dynamic region, as described in Creating ADF Dynamic Regions.

How to Create a Parameter Map to Specify Input Parameters for an ADF Region

You configure the task flow binding's `parametersMap` property to reference the parameter map object that defines the key-value pairs you want to pass to the ADF region.

Before you begin:

It may be helpful to have an understanding of the configuration options available to you when passing input parameters to an ADF region. For more information, see Specifying Parameters for ADF Regions Using Parameter Maps.

You may also find it helpful to understand functionality that can be added using other task flow and ADF region features. For more information, see Additional Functionality for Task Flows that Render in ADF Regions.

You will need to complete this task:

> Create a managed bean or edit an existing managed bean so that it returns an object that implements the `java.util.Map` interface. Configure the managed bean so the object returns key-value pairs with the values that you want to pass to the ADF region. For more information about managed beans, see Using a Managed Bean in a Fusion Web Application.

To create a parameter map to specify input parameters for an ADF region:

1. In the Applications window, right-click the JSF page that holds the ADF region and choose **Go to Page Definition**.
2. In the overview editor, click the **Bindings and Executables** navigation tab, select the task flow binding or multi task flow binding for which you want to specify a parameter map, and then click the **Edit** icon.

   The type of task flow binding that you select determines the dialog that appears.
3. In the Edit Task Flow Binding dialog or the Edit Multi Task Flow Binding dialog, select **Expression Builder** from the **Input Parameters Map** dropdown menu.
4. Write or build an EL expression that references a parameter map. For example, write an EL expression similar to the following:

   ```
   #{pageFlowScope.userInfoBean.parameterMap}
   ```
5. Click **OK**.

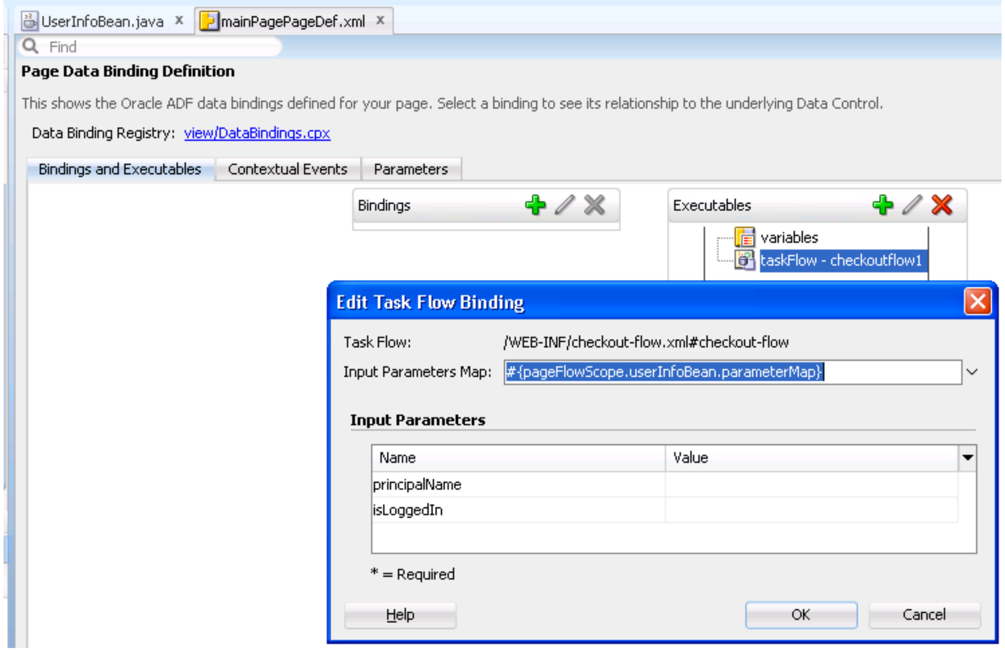
What Happens When You Create a Parameter Map to Specify Input Parameters

At runtime, the task flow binding or multi task flow binding evaluates the EL expression specified for its `parametersMap` property. It returns values to the ADF region from the managed bean for keys that match the name of the input parameters defined for the bounded task flow that render in the ADF region.

Example 26-1 shows code snippets from a managed bean that puts two values (`isLoggedIn` and `principalName`) into a parameter map named `parameterMap`.

Figure 26-6 shows the Edit Task Flow Binding dialog after you close the Expression Builder dialog, having specified the parameter map object (`parameterMap`) shown in Example 26-1. The **Input Parameters** field in the Edit Task Flow Binding dialog lists the input parameters defined for the bounded task flow associated with this ADF region (`checkout-flow`). The task flow binding retrieves the values for these parameters from the managed bean shown in Example 26-1.

Figure 26-6 EL Expression Referencing Parameter Map on Task Flow Binding



Page Data Binding Definition

This shows the Oracle ADF data bindings defined for your page. Select a binding to see its relationship to the underlying Data Control.

Data Binding Registry: view/DataBindings.cpx

Edit Task Flow Binding

Task Flow: /WEB-INF/checkout-flow.xml#checkout-flow

Input Parameters Map: #{pageFlowScope.userInfoBean.parameterMap}

Input Parameters

Name	Value
principalName	
isLoggedIn	

* = Required

The following example shows the metadata that appears for the task flow binding in the page definition file of the page that renders the ADF region. The metadata for the task flow binding references both the bounded task flow (`taskFlowId` attribute) and the managed bean (`parametersMap`).

```
<taskFlow id="checkoutflow1"
              taskFlowId="/WEB-INF/checkout-flow.xml#checkout-flow"
              activation="deferred"
              xmlns="http://xmlns.oracle.com/adf/controller/binding"
              parametersMap="#{pageFlowScope.userInfoBean.parameterMap}"/>
```

You specify the `<parameterMap>` element in the page definition.

Note:

If you specify `Refresh="ifNeeded"`, parameters are not supported in the `<parameterMap>` element. The only condition that determines whether the region need to be refreshed is the boolean value returned by the evaluation of `RefreshCondition`. For more information, see What You May Need to Know About Configuring an ADF Region to Refresh.

Example 26-1 Managed Bean Defining a Parameter Map

```
import java.util.HashMap;
import java.util.Map;

public class UserInfoBean {
    private Map<String, Object> parameterMap = new HashMap<String, Object>();

    public Map getParameterMap() {

        parameterMap.put("isLoggedIn", getSecurity().isAuthenticated());
        parameterMap.put("principalName", getSecurity().getPrincipalName());
        return parameterMap;
    }
}
```

Configuring an ADF Region to Refresh

The ADF region refreshes when the parent page binding is first displayed. For future refreshes you must configure one of these task flow binding attributes: taskFlowID, Refresh or RefreshCondition, RefreshCondition="#{EL.expression}", or Refresh="ifNeeded".

You can configure when an ADF region refreshes and whether it invokes a task flow. An ADF region can only invoke a task flow when the ADF region is in an active state. An ADF region in an inactive state cannot invoke a task flow and returns a null value for the ID of the referenced task flow to the parent page.

How to Configure the Refresh of an ADF Region

You set values for the task flow binding of the task flow associated with the ADF region to determine when an ADF region switches from an inactive to an active state and to determine when an ADF region refreshes.

Before you begin:

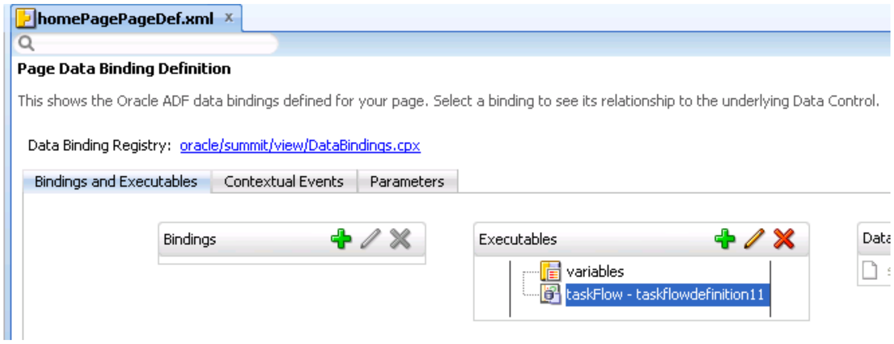
It may be helpful to have an understanding of the requirements for a bounded task flow that you use in an ADF region. For more information, see Configuring an ADF Region to Refresh.

You may also find it helpful to understand functionality that can be added using other task flow features and ADF region features. For more information, see Additional Functionality for Task Flows that Render in ADF Regions.

To configure the refreshing of an ADF Region:

1. In the Applications window, select the page that contains the ADF region, right-click and choose **Go to Page Definition**.
2. In the page definition file, select the task flow binding or multi task flowing binding in the **Executables** section, as illustrated in Figure 26-7.

Figure 26-7   Task Flow Binding



homePagePageDef.xml

Page Data Binding Definition

This shows the Oracle ADF data bindings defined for your page. Select a binding to see its relationship to the underlying Data Control.

Data Binding Registry: oracle/summit/view/DataBindings.cpx

3. Choose the appropriate option to refresh the ADF region.
   - Refresh if the value of a task flow binding parameter changes.

     In the Properties window, select **ifNeeded** from the **Refresh** dropdown list.

     Do not select **ifNeeded** if you want to set a value for the `RefreshCondition` property and can guarantee that the property evaluates to `true` only once. The `RefreshCondition` property will not be evaluated if you select **ifNeeded** for the `Refresh` property.
   - Refresh if a condition that you specify returns `true`. In the Properties window, write an EL expression that guarantees that `true` is returned only once and that returns a boolean in the **RefreshCondition** field. If the EL expression returns `true`, the ADF region refreshes. If the EL expression returns `true` more than once for each task flow refresh, page content may fail to load.

> **Note:**
>
> Use **ifNeeded** when possible. The usage of **RefreshCondition** is not recommended unless there is a good reason. For example, use **RefreshCondition** only when the application passes parameters to the task flow binding using a dynamic parameter map and when the application can guarantee **RefreshCondition** evaluates `true` exactly once when it refreshes the ADF task flow. For more information, see What You May Need to Know About Configuring an ADF Region to Refresh.

4. In the Properties window, select a value from the **activation** dropdown list, as described in the following list:
   - **conditional**: activates the ADF region if the EL expression set as a value for the task flow binding `active` property returns `true`.
   - **deferred**: select this option if your application uses Facelets XHTML pages in the view layer and you want to activate the ADF region when a Facelets XHTML page first requests a `viewID`. If your application uses JSP technology

in the view layer, selecting this option has the same effect as selecting **immediate** (the ADF region activates immediately) provided that the parent component for the ADF region is not a `popup` or `panelTabbed` component that has its `childCreation` attribute set to `deferred`. If this latter scenario occurs, the parent component (`popup` or `panelTabbed`) determines behavior.

Use this option if your application uses Facelets XHTML pages. For more information about Facelets with ADF Faces, see the "Getting Started with ADF Faces and JDeveloper" chapter in *Developing Web User Interfaces with Oracle ADF Faces*.

- **immediate**: activates the ADF region immediately. This is the default value.

The value that you select in the dropdown list determines when an ADF region switches from an inactive to an active state. An ADF region must have an active state before it can invoke a task flow.

5. If you selected **conditional** as the value for the activation property, write an EL expression that returns a boolean value at runtime for the active property. If the EL expression returns `true`, the ADF region invokes the task flow.
6. (Optional) For the `processRegionForRefresh` property, write an EL expression to determine if the region should participate in a refresh operation. If the EL expression evaluates to false, the region binding and its children will not be refreshed. This check is performed before the region activation, the `RefreshCondition` property, or input parameters are evaluated. If the EL expression evaluates to true or if you do not write an EL expression for the `processRegionForRefresh` property, the region will be included in refresh operations and the behavior that you specify for the activation dropdown list will be invoked. Writing an EL expression for the `processRegionForRefresh` property does not prevent the region from being rendered.

What You May Need to Know About Configuring an ADF Region to Refresh

An ADF region initially refreshes when the parent JSF page on which the region is located first displays. During the initial refresh, any ADF region task flow binding parameter values are passed in from the parent page. The parameter values are used to display the initial page fragment within the ADF region. If the bounded task flow relies on input parameter values that are not available when the page is first created, make sure that your task flow behaves correctly if the input parameter values are null by, for example, using a `NullPointerException` object. Alternatively (or additionally), make sure that the task flow does not activate until the input parameters become available by configuring the task flow binding's `active` property.

An ADF region task flow binding can be refreshed again based on one of the following task flow binding attributes:

- `taskFlowId`

  This attribute specifies the directory path and the name of the source file for the bounded task flow. The ADF region refreshes if this attribute value changes.

- Neither `Refresh` or `RefreshCondition` attributes are specified (default)

  If neither the `Refresh` or `RefreshCondition` task flow binding attribute is specified, the ADF region is refreshed only once at the time the parent page is first displayed unless you configure a value for the task flow binding's `active` property.

- `RefreshCondition="#{EL.expression}"`

  The `RefreshCondition` must evaluate to a boolean value, and the EL expression must return `true` just **one time** for each task flow refresh. This means the expression cannot be a reference to a state variable; instead, it must be a call to a method that possesses the internal check of the return value. If no check is implemented in the method call and the expression is allowed to return `true` more than once, then during the refresh of a dynamic region, the application will display the error "The content of this page failed to load as expected because data transmission was interrupted".

  > **Note:**
  >
  > A dynamic region may fail to load unless the application can guarantee that the `RefreshCondition` expression evaluates to `true` only once for each task flow refresh. For example, assume you have assigned an expression to the refresh condition for the task flow definition. If the EL expression evaluates to `true`, the region model defined by the controller is refreshed and, as a result, the client ID index for the region is incremented. This will cause any operation that expects the original client ID for the region to fail as it does not find the expected ID. One such operations is streaming of table row data.

  At the time the `RefreshCondition` is evaluated, if the variable `bindings` is used within the EL Expression, the context refers to the binding container of the parent page, not the page fragment displayed within the ADF region. `RefreshCondition` is independent of the change of value of binding parameters. If the task flow binding parameters do not change, nothing within the ADF region will change. The expression is evaluated during the `PrepareRender` phase of the ADF page lifecycle. For more information, see Understanding the Fusion Page Lifecycle .

- `Refresh="ifNeeded"`

  Any change to a task flow binding parameter value causes the ADF region to refresh. If the ADF region task flow binding does not have parameters, `Refresh="ifNeeded"` is equivalent to not specifying the `Refresh` attribute.

  `Refresh="ifNeeded"` is not supported if you pass parameters to the task flow binding using a dynamic parameter map. You must instead use the `RefreshCondition="#{EL.Expression}"`.

  For more information about specifying parameter values using a parameter map, see Specifying Parameters for ADF Regions Using Parameter Maps.

The `Refresh="ifNeeded"` property takes precedence over `RefreshCondition` and `RefreshCondition` will not be evaluated if you set `Refresh="ifNeeded"`. When possible, use the `Refresh="ifNeeded"` property to control region refresh behavior and to avoid the page content loading failure that can result when `RefreshCondition="#{EL.expression}"` evaluates to true more than once for each task flow refresh.

The following example contains a sample task flow binding located within the page definition of a page on which an ADF region has been added.

```
<taskFlow id="Department1" taskFlowId="/WEB-INF/Department#Department"
    xmlns="http://xmlns.oracle.com/adf/controller/binding">
```

```
  <parameters>
    <parameter id="DepartmentId" value="#{bindings.DepartmentId.inputValue}"
        xmlns="http://xmlns.oracle.com/adfm/uimodel"/>
  </parameters>
</taskFlow>
```

You do not need to refresh an ADF region to refresh the data controls inside the ADF region. During the ADF lifecycle, the refresh events telling the iterators to update will be propagated to the binding container of the current page of the ADF region.

What You May Need to Know About Refreshing an ADF Region

When a binding container hierarchy is refreshed, the region binding code evaluates the input parameter values and compare the new values with the previous values for the region binding only if the region binding has a refresh condition set to **ifNeeded**. See Configuring an ADF Region to Refresh. If a parameter value is changed, the region is refreshed. For example, if a region displays the employee details of an employee and if any of the details of the employee is changed, then region refreshes and displays the details for the selected employee.

Additionally, if you have added support for `childCreation` attribute set to **lazyUncached** on various UI components, the ADF view prunes the portions of the view tree and those portions are not included in the active view tree and are not visible. During the refresh of binding container hierarchy, the ADF controller evaluates the input parameters for all the regions that were pruned and are not visible in the active view tree of the page layout. The operations on the corresponding binding containers that are not in active tree view might result in poor performance.

To avoid refreshing the regions that are not visible in the active tree view, set the `oracle.adfinternal.controller.regionPruning` parameter value to `true`. This parameter checks if the region is part of the active view tree before evaluating if the input parameter values are modified. The region will then be refreshed only if the region is part of the active view tree and one or more parameter values are modified.

This property uses the Trinidad property service, therefore, you can set this property using the Fusion Apps property mechanism. Alternatively, you can enable this property using the `adf-settings.xml` file.

```
<adf-properties-settings
xmlns="http://xmlns.oracle.com/adf/properties/settings">
      <properties>
         <property>
           <name>oracle.adfinternal.controller.regionPruning</name>
           <default-value>true</default-value>
         </property>
      </properties>
   </adf-properties-settings>
```

Configuring Activation of an ADF Region

The ADF regions can be configured to be activated at the appropriate time. This is helpful to optimize the performance of the task flow binding these regions. You can set the activation of the region either as conditional, deferred, or immediate based on your requirement.

You can configure when an ADF region activates. This has the effect of determining when the task flows contained in ADF regions activate. Configuring the activation of

ADF regions can optimize the performance of a Fusion web application's page. For example, a page that contains 5 ADF regions may reference 5 task flows. You may not want these 5 task flows to execute simultaneously when the Fusion web application page loads so you configure an `activation` property to determine when a task flow executes.

How to Configure Activation of an ADF Region

You configure the `activation` property for the task flow binding associated with the ADF region to determine when to activate the ADF region.

Before you begin:

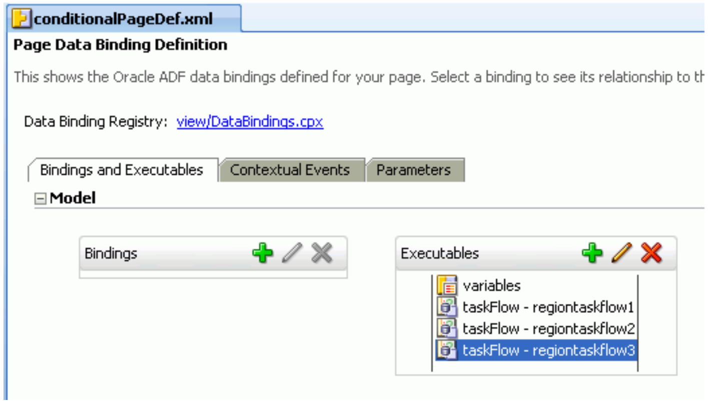
It may be helpful to have an understanding of the configuration options for activating an ADF region. For more information, see Configuring Activation of an ADF Region.

You may also find it helpful to understand functionality that can be added using other task flow features and ADF region features. For more information, see Additional Functionality for Task Flows that Render in ADF Regions.

To configure the activation of an ADF region:

1. In the Applications window, select the page that contains the ADF region(s), right-click and choose **Go to Page Definition**.
2. In the page definition file, select the task flow binding or multi task flow binding in the **Executables** section, as illustrated in Figure 26-8.

Figure 26-8 Task Flow Bindings for ADF Region Configured for Activation



conditionalPageDef.xml

Page Data Binding Definition

This shows the Oracle ADF data bindings defined for your page. Select a binding to see its relationship to th

Data Binding Registry: view/DataBindings.cpx

variables
taskFlow - regiontaskflow1
taskFlow - regiontaskflow2
taskFlow - regiontaskflow3

3. In the Properties window, select a value from the **activation** dropdown list, as described in the following list:
   - **conditional**: activates the ADF region if the EL expression set as a value for the task flow binding `active` property returns `true`.
   - **deferred**: select this option if your application uses Facelets XHTML pages in the view layer and you want to activate the ADF region when a Facelets XHTML page first requests a `viewID`. If your application uses JSP technology in the view layer, selecting this option has the same effect as selecting **immediate** (the ADF region activates immediately).

     Use this option if your application uses Facelets XHTML pages. For more information about Facelets with ADF Faces, see the "Getting Started with

ADF Faces and JDeveloper" chapter in *Developing Web User Interfaces with Oracle ADF Faces*.

- **immediate**: activates the ADF region immediately. This is the default value.

The value that you select in the dropdown list determines when an ADF region switches from an inactive to an active state. An ADF region must have an active state before it can invoke a task flow.

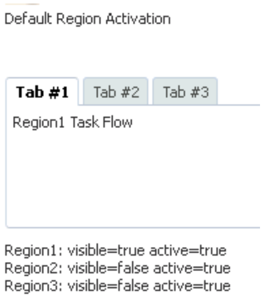
4. If you selected **conditional** as the value in the **activation** dropdown list, write an EL expression that returns a boolean value at runtime for the active property. If the EL expression returns `true`, the ADF region invokes the task flow. If the EL expression returns `false`, the ADF region deactivates a task flow that was active in the ADF region.

What Happens When You Configure Activation of an ADF Region

The behavior of the ADF region at runtime depends on the options that you set for the `activation` property.

Figure 26-9 shows the default behavior where all task flows in ADF regions execute when the page loads the ADF regions (Tab # 1, Tab # 2, Tab # 3).

Figure 26-9 Default Activation of ADF Regions

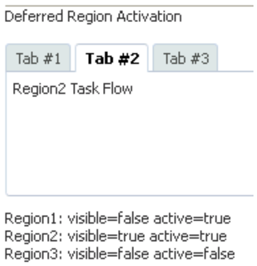


Region1 Task Flow

Region1: visible=true active=true
Region2: visible=false active=true
Region3: visible=false active=true

Figure 26-10 shows an example where the activation property was set to `deferred` and the application uses Facelets. The regions in Tab # 1 and Tab # 2 have an active state because the end user navigated to these regions. The region in Tab # 3 is inactive.

Figure 26-10 Deferred Activation of ADF Regions



Region2 Task Flow

Region1: visible=false active=true
Region2: visible=true active=true
Region3: visible=false active=false

Figure 26-11 shows an example where the `activation` property was set to `conditional` and the `active` property to an EL expression that returns a `boolean` value at runtime. The region in Tab # 1 is in an active state because the EL expression specified for the `active` property returns `true`. The regions in Tab # 2 and Tab # 3 are inactive because the EL expression specified for the `active` property returns `false`.

Note that the following events occur if a region deactivates a task flow (the value returned by the `active` property changes from `true` to `false`):

- A task flow with an active transaction rolls back the transaction.

  For more information about transaction options in task flows, see Managing Transactions in Task Flows.

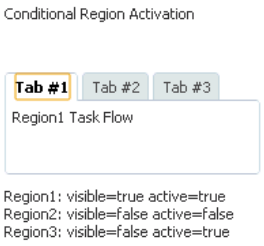
- If a task flow has a data control frame with a `data-control-scope` value of `isolated`, the task flow releases the data control frame and any data controls in the data control frame.

  For more information about data controls, see Sharing Data Controls Between Task Flows and What You May Need to Know About Sharing Data Controls and Managing Transactions.

- ADF Controller releases the view port data structures for the region (including pageFlow and view scopes).

  For more information about view ports, see About View Ports and ADF Regions. For more information about memory scopes, see What You May Need to Know About Memory Scope for Task Flows.

Figure 26-11   Conditional Activation of ADF Regions



```
Region1: visible=true active=true
Region2: visible=false active=false
Region3: visible=false active=true
```

Navigating Outside an ADF Region's Task Flow

You can navigate out of the ADF region and then display the parent view activity of the task flow that binds the region, or you can display the root page of its application. To do so, you must expose the properties (parent-outcome and root-outcome) of the parent action activity that you configure.

A bounded task flow running in an ADF region may need to trigger navigation of its parent view activity or to navigate to the root page of its application. The parent action activity exposes properties (`parent-outcome` and `root-outcome`) that you configure if you want to implement either of these use cases.

For example, you might specify a value for `parent-outcome` if you have a page that displays employee information and an ADF region that contains an enroll button for an employee. After the enroll page completes and the ADF region returns, the employee information page refreshes with the next employee.

> **Note:**
>
> An ADF region does not maintain state when you navigate away from the region.

How to Trigger Navigation Outside of an ADF Region's Task Flow

You add a parent action activity to the bounded task that runs in the ADF region and configure it so that it navigates to the parent's view activity.

Before you begin:

It may be helpful to have an understanding of the configuration options for activating an ADF region. For more information, see Configuring Activation of an ADF Region.

You may also find it helpful to understand functionality that can be added using other task flow features and ADF region features. For more information, see Additional Functionality for Task Flows that Render in ADF Regions.

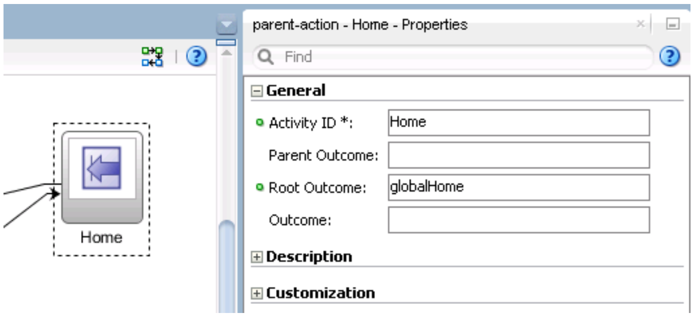
To trigger navigation outside of an ADF region's task flow:

1. In the Applications window, double-click the bounded task flow that runs in the ADF region.
2. In the ADF Task Flow page of the Components window, from the Component panel, in the Activities group, drag and drop a **Parent Action** onto the diagram for the bounded task flow.
3. Select the parent action activity in the diagram for the bounded task flow.
4. In the Properties window, choose the appropriate option:
   - **Parent Outcome**: enter a literal value or write an EL expression that returns an outcome so that the parent action activity navigates to the parent view activity. The outcome is then used to navigate the parent view activity's task flow rather than navigating the ADF region's bounded task flow.

     The parent view activity's task flow should contain a control flow case or wildcard control flow rule that accepts the value you specify for `parent-outcome`.
   - **Root Outcome**: enter a literal value or write an EL expression that returns an outcome so that the parent action activity navigates to the root page of the application.

     For example, to navigate from an ADF region to the home page in your application, you specify a value for **Root Outcome**, as illustrated in Figure 26-12.

Figure 26-12 Navigating to Home Page



Activity ID *: Home
Parent Outcome:
Root Outcome: globalHome
Outcome:

> **Note:**
>
> **Parent Outcome** and **Root Outcome** are mutually exclusive.

5. (Optional) In the **Outcome** field, enter a literal value that specifies an outcome for control flow within the ADF region after the parent action activity adds `parent-outcome` or `root-outcome` to the appropriate queue.

   Specifying an `outcome` element is useful in cases where the parent view activity or root page does not navigate as a result of the `parent-outcome` or `root-outcome` sent by the parent action activity. In addition, the ADF region should not continue to display the same view. If you do not specify a value for `outcome`, the ADF region's `viewId` remains unchanged.

What Happens When You Configure Navigation Outside a Task Flow

At design time, JDeveloper writes entries to the source file for the bounded task flow based on the property values you set for the parent action activity. The following example shows sample entries that JDeveloper generates when you write a literal value for the Parent Outcome and Outcome properties of the parent action activity.

```
<parent-action id="parentAction1">
    <parent-outcome>parent_outcome</parent-outcome>
    <outcome id="__2">outcome</outcome>
</parent-action>
```

The following example shows sample entries that JDeveloper generates when you write a literal value for the Root Outcome and Outcome properties of the parent action activity.

```
<parent-action id="parentAction1">
      <root-outcome>root_outcome</root-outcome>
      <outcome id="__2">outcome</outcome>
</parent-action>
```

At runtime, when navigation in an ADF region’s bounded task flow encounters a parent view activity, the parent view activity’s outcome is queued and the current navigation operation in the ADF region’s bounded task flow continues. Once the current navigation operation completes, the queued outcome triggers a new navigation operation on the parent. If, for example, the next activity in the bounded task flow of the ADF region is a method call activity and the parent action activity navigates to a task flow that also includes a method call activity, the method call activity in the bounded task flow of the ADF region executes first.

What You May Need to Know About How a Page Determines the Capabilities of an ADF Region

In some cases, a page may need to determine the capabilities currently available within one of the ADF regions that it contains. For example, the page may need to initiate control flow within the ADF region based on its current state using the `queueActionEventInRegion()` method. **Region capabilities** are used by a parent page to identify the current outcomes of one of its regions based on the region's current state. They identify to the parent page whether or not an outcome is possible within the region.

The following scenario describes how region capabilities might typically be used in an application:

1. An ADF region within a page displays its associated page fragment.
2. A user selects a button or performs some other action in the ADF region.
3. The parent page identifies the outcomes (region capabilities) for the ADF region.
4. The parent page updates its buttons based on the ADF region capabilities.

The syntax of an EL expression to determine an ADF region's capabilities is as follows:

```
#{bindings.[regionId].regionModel.capabilities['outcome']}
```

where `regionId` is the ID of the ADF region component on the page and `outcome` is the possible outcome being verified within the ADF region.

Region capabilities require the availability of the specified ADF region's `regionModel` through an EL expression. The EL expression should never access bindings in any other binding container other than the current binding container. Region capabilities cannot be used in some cases. For example, a nested ADF region where a `regionModel` cannot be reached within the current binding container. This is because a nested region's nested binding container might not yet exist or might have already been released.

Configuring Transaction Management in an ADF Region

The ADF task flow allows you to optionally choose if you want to allow a task flow to share or isolate its data controls with a calling talk flow. For this, you must set a value for the data-control-scope property of the bounded task flow. You can also rollback or commit the changes made by the calling task flow by exposing the methods by the DataControlFrame interface of the ADF Model layer.

In the ADF Controller layer, bounded task flows optionally provide their own abstract implementation of transactions over the underlying ADF Model layer, as described in Managing Transactions in Task Flows. You can choose whether a task flow shares or isolates its data controls with a calling task flow by setting a value for its `data-control-scope` property, as described in Sharing Data Controls Between Task Flows . Whether a task flow creates a new transaction depends on the values you choose for the `transaction` and `data-control-scope` properties, as described in What You May Need to Know About Sharing Data Controls and Managing Transactions.

If, for example, you share a task flow's data control with a calling task flow, a new transaction is created at the ADF Controller layer level if you configure the task flow's `transaction` option to **Always Begin New Transaction** (`new-transaction`) or to **Use Existing Transaction If Possible** (`requires-transaction`). The `requires-transaction` option creates a new transaction only if a transaction does not already exist. The created transaction has the same lifespan as the ADF region that renders the task flow. This may cause behavior that you do not intend when the task flow renders in an ADF region. For example, if an ADF region refreshes, it restarts the task flow that it renders and causes the following events to occur:

- Discards the current view port that the ADF region uses

  > **Note:**
  >
  > Save points require a root view port to work. For more information about view ports, see About View Ports and ADF Regions. For more information about save points, see Using Save Points in Task Flows.

- Releases the data control that the task flow uses

  This may cause the loss of changes made in the parent page of the calling task flow because it shares the data control of the task flow that renders in the ADF region.

- Executes a commit or rollback if the End Transaction property of the task flow's task flow return activity specifies one of these actions. For more information, see Using Task Flow Return Activities .

There are a number of ways to prevent the inadvertent termination of a task flow's transaction by the refresh of an ADF region. All these options disassociate the lifespan of the transaction in the task flow that renders in an ADF region from the lifespan of the ADF region. These options make sure that the region's parent page owns the transaction and/or that the transaction starts before the region renders. Choose the most appropriate option for your application:

- Set the transaction option on the task flow that renders in the ADF region to **Always Use Existing Transaction** (`requires-existing-transaction`). This throws an exception if a transaction does not exist and prevents the ADF region from owning the transaction.
- Set the calling task flow's transaction option to **Always Begin New Transaction** (`new-transaction`) so that the lifespan of the transaction is associated with the lifespan of the parent page.
- Invoke the `DataControlFrame` interface's `beginTransaction()` method in the action that navigates to the parent page that contains the region.
- Configure a method call activity that invokes the `DataControlFrame` interface's `beginTransaction()` method from a managed bean with a request scope before navigating to the parent page of the ADF region.

  The following code begins a transaction:

  ```
  public void beginTransaction()
  {
    BindingContext context = BindingContext.getCurrent();

    String dcFrameName = context.getCurrentDataControlFrame();
  ```

```
        DataControlFrame dcFrame = context.findDataControlFrame(dcFrameName);
        dcFrame.beginTransaction(new TransactionProperties());
      }
```

If you share the data controls of a task flow with a calling task flow, use the `rollback` or `commit` options exposed by the task flow's task flow return activity to commit or rollback changes rather than programmatically use the methods exposed by the ADF Model layer's `DataControlFrame` interface. Using the methods exposed by the `DataControlFrame` interface closes the transaction and may cause the task flow that renders in the ADF region to create a new transaction.

Creating ADF Dynamic Regions

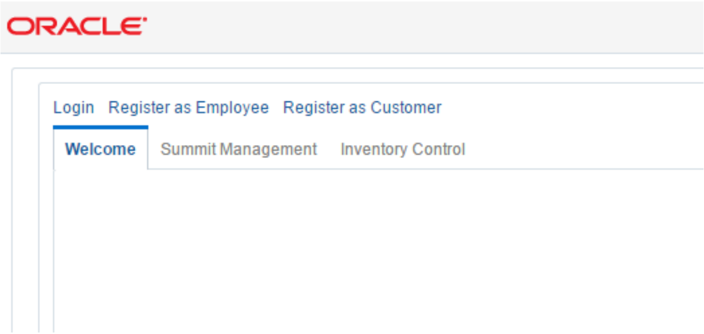
An ADF dynamic region is determined by the taskFlowId attribute of the binding task flow at runtime. This is helpful when you want to run the task flows based on the user choice at runtime. You can add additional bounded task flows to the dynamic region by creating ADF dynamic region links.

An ADF dynamic region is an ADF region where the task flow binding dynamically determines the value of its `taskFlowId` attribute at runtime. This allows the Fusion web application to determine which bounded task flow to execute within the ADF dynamic region based on the result of evaluation of the task flow binding's `taskFlowId` attribute.

Figure 26-13 shows a runtime example from the Summit ADF task flow sample application where the `index.jsf` page renders a different registration task flow in the `showDetailItem` component labeled **Welcome** in response to the link component that the end user clicks. For example, if the end user clicks **Register as Employee**, the application renders the `emp-reg-task-flow-definition.xml` task flow in the ADF dynamic region. If the end user clicks **Register as Customer**, the application renders the `edit-customer-task-flow-definition.xml` task flow in the ADF dynamic region.

At runtime, the ADF dynamic region swaps the task flows that it renders during the Prepare Render lifecycle phase. To give components from a previous task flow the opportunity to participate smoothly in the lifecycle phases, do not navigate off the `regionModel` until the JSF Invoke Application lifecycle phase. It is good practice, therefore, not to navigate a task flow in a dynamic region in response to contextual events. For information about lifecycle phases, see Understanding the Fusion Page Lifecycle , and for information about contextual events, see Using Contextual Events.

Figure 26-13 ADF Dynamic Region in the Summit ADF Task Flow Sample Application



You create an ADF dynamic region by dragging and dropping a bounded task flow to the page where you want to render the ADF dynamic region. The view activities in the bounded task flow must be associated with page fragments (`.jsff`).

If you attempt to drag and drop a bounded task flow that is associated with pages rather than page fragments, JDeveloper does not display the context menu that allows you to create an ADF dynamic region. You can convert a bound task flow that uses pages to use page fragments. See How to Convert Bounded Task Flows .

After you create an ADF dynamic region, you can add additional bounded task flows to the dynamic region by creating ADF dynamic region links, as described in Adding Additional Task Flows to an ADF Dynamic Region.

How to Create an ADF Dynamic Region

Drag and drop bounded task flows to the page where you want to render the ADF dynamic region.

Before you begin:

It may be helpful to have an understanding of the configuration options available to you when creating an ADF dynamic region. For more information, see Creating ADF Dynamic Regions.

You may also find it helpful to understand other functionality that can be added using other task flow and ADF region features. For more information, see Additional Functionality for Task Flows that Render in ADF Regions.

To create an ADF dynamic region:

1. In the Applications window, double-click the JSF page where you want to create the ADF dynamic region.
2. From the Applications window, drag and drop the first bounded task flow onto the JSF page, and from the context menu, choose **Create** > **Dynamic Region**.
3. Choose the appropriate option in the Choose Managed Bean for Dynamic Region dialog:
   - If you want an existing managed bean to store the bounded task flow's ID, select an existing managed bean from the **Managed Bean** dropdown list.

     The managed bean passes the value of the bounded task flow's ID into the task flow binding of the ADF dynamic region. Make sure that the managed bean you select is not used by another ADF dynamic region.
   - If no managed bean exists for the page, click the **Add** icon next to the **Managed Bean** dropdown list to create a new one. Enter values in the dialog that appears as follows:
   - a. **Bean Name**: Enter a name for the new managed bean. For example, enter `DynamicRegionBean`.

     b. **Class Name**: Enter the name of the new managed bean class.

     c. **Package**: Enter the name of the package that is to contain the managed bean or browse to locate it.

     d. **Extends**: Enter the name of the Java class that the managed bean extends. The default value is `java.lang.Object`.

e. **Scope**: This field is read-only and its value is set to `backingBean`. For more information about the memory scope for managed beans, see What You May Need to Know About Memory Scope for Task Flows.

f. Click **OK** to close the dialogs where you configure the managed bean.

4. Choose the appropriate option in the Edit Task Flow Binding dialog:

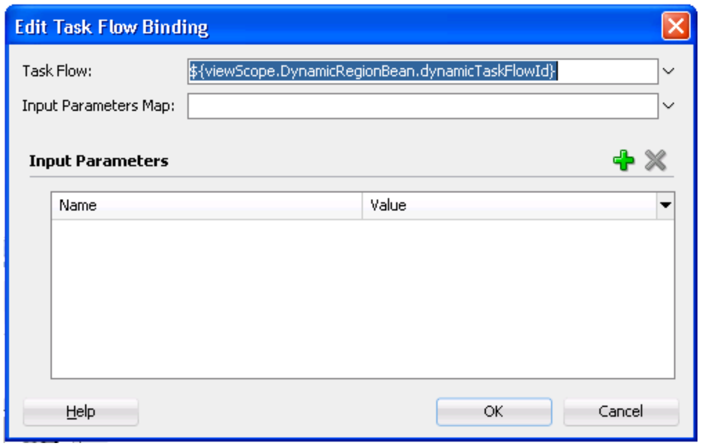
   - Click **OK** if you do not want specify input parameters or an input parameter map for the ADF dynamic region.
   - Specify input parameters for the ADF dynamic region.

     For more information, see Specifying Parameters for an ADF Region.

   Figure 26-14 shows the Edit Task Flow Binding dialog that appears after you configure a managed bean.

   > **Note:**
   >
   > You can write an EL expression that references a list of input parameters specified in a managed bean using the Input Parameters Map field of the Edit Task Flow Binding dialog. For more information about implementing this functionality, see Specifying Parameters for ADF Regions Using Parameter Maps.

Figure 26-14 Edit Task Flow Binding Dialog for an ADF Dynamic Region



Task Flow: ${viewScope.DynamicRegionBean.dynamicTaskFlowId}

Input Parameters Map:

5. Click **OK**.

What Happens When You Create an ADF Dynamic Region

When you drop a bounded task flow onto a JSF page to create an ADF dynamic region, JDeveloper adds an `af:region` tag to the page. The `af:region` tag contains a reference to a task flow binding. The following example shows the `af:region` tag that renders the ADF dynamic region in the `index.jsf` page of the Summit ADF task flow sample application.

```
<af:region value="#{bindings.dynamicRegion1.regionModel}" id="r4"
                                rendered="#{WelcomePageBean.isRegistration}"/>
```

JDeveloper also adds a task flow binding to the page definition file of the page that hosts the ADF dynamic region. The following example shows a sample of the metadata that JDeveloper adds. The task flow binding provides a bridge between the ADF dynamic region and the bounded task flow. It binds the ADF dynamic region to the bounded task flow and maintains all information specific to the bounded task flow.

```
<taskFlow id="dynamicRegion1"
       taskFlowId="${WelcomePageBean.dynamicTaskFlowId}"
       activation="deferred"
       xmlns="http://xmlns.oracle.com/adf/controller/binding"/>
```

The `taskFlowId` attribute in the task flow binding metadata specifies the managed bean that determines at runtime what bounded task flow to associate with the ADF dynamic region. JDeveloper creates this managed bean or modifies an existing managed bean to store this data. The following example shows extracts of the code that JDeveloper writes to the managed bean.

```
import oracle.adf.controller.TaskFlowId;
 ...
    private String taskFlowId = "/directoryPath/toTaskFlow";
 ...
    public TaskFlowId getDynamicTaskFlowId() {
        return TaskFlowId.parse(taskFlowId);
    }
...
```

At runtime, the managed bean stores the value of the bounded task flow's ID (`taskFlowId`) that displays inside the ADF dynamic region. The managed bean swaps the different bounded task flows into the task flow binding of the ADF dynamic region.

When an ADF dynamic region reinitializes, the Fusion web application must reinitialize the task flow binding associated with the ADF dynamic region. This includes evaluating if there are new input parameters and input parameter values to pass to the ADF dynamic region.

Adding Additional Task Flows to an ADF Dynamic Region

The ADF dynamic region helps you to display data from different task flows without navigating away from the current display area. This action is triggered by the end user by clicking a command component. This is possible as the ADF dynamic region has the ability to swap a bounded task flow with another bounded task flow.

An ADF dynamic region link swaps a bounded task flow for another bounded task flow within an ADF dynamic region. An end user clicks a command component (a button or a link) to update the ADF dynamic region with the new bounded task flow.

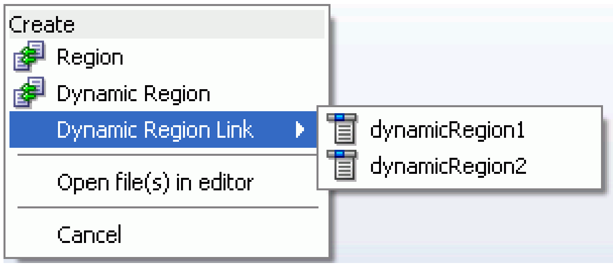
For example, a bounded task flow in the ADF dynamic region displays general information about an employee such as an ID and photo. When the end user clicks a link command component labeled **Details**, the ADF dynamic region updates with a table containing more information about the employee from another bounded task flow. The end user's action (clicking the link) invokes a method on the ADF dynamic region's managed bean. The value of the new bounded task flow is passed to the method, and the ADF dynamic region refreshes with the new bounded task flow. The new bounded task flow now displays within the ADF dynamic region.

By default, a ADF dynamic region link swaps another bounded task flow in for the original, but cannot swap back to the original. To toggle back to the original bounded task flow, you could add a second ADF dynamic region link on the page that, when clicked, swaps the current task flow back to the original one.

You can add an ADF dynamic region link if you already have at least one ADF dynamic region on a page and are adding a new bounded task flow as an ADF dynamic region to the same page. After you drop the bounded task flow on the page and choose to create an ADF dynamic region link, a menu displays all of the dynamic regions currently on the page, as shown in Figure 26-15.

A list of the current dynamic regions in a document appears when you choose **Dynamic Region Link** from the menu that appears after you drag and drop a bounded task that you want to add as an option to an existing dynamic region in the document, as shown in Figure 26-15.

Figure 26-15 ADF Dynamic Region Link Menu



Use this menu to select the ADF dynamic region within which you want to display the contents of the bounded task flow.

Tip:

You can use the values in the dynamic region link in other UI components. For example, you could create a selection list in which each of the items in the list links to a different bounded task flow. All of the linked bounded task flows would display in the same dynamic region. The links perform a method in the class behind the managed bean created to swap the bounded task flow it displays.

How to Create an ADF Dynamic Region Link

You drag and drop a bounded task flow to a page that already contains an ADF dynamic region and you select **Dynamic Region Link** on the context menu that appears to view a list of ADF dynamic regions to which you can create a link.

Before you begin:

It may be helpful to have an understanding of the configuration required before you attempt to create an ADF dynamic region link. For more information, see Adding Additional Task Flows to an ADF Dynamic Region.

You may also find it helpful to understand functionality that can be added using other task flow and ADF region features. For more information, see Additional Functionality for Task Flows that Render in ADF Regions.

You will need to complete this task:

> Add at least one ADF dynamic region to the JSF page where you are going to create an ADF dynamic region link. For information about adding an ADF dynamic region to a JSF page, see Creating ADF Dynamic Regions.

To create an ADF dynamic region link:

1. In the Applications window, double-click the JSF page where you want to create the ADF dynamic region link.
2. From the Applications window, drag and drop a bounded task flow onto the JSF page and, from the context menu, choose **Dynamic Region Link**.

   > **Note:**
   >
   > The view activities of the bounded task flow must be associated with page fragments. You can convert a bounded task flow that uses pages to use page fragments. For more information, see How to Convert Bounded Task Flows .

3. In the context menu that displays a list of the ADF dynamic regions that have already been added to the page, select the name of the ADF dynamic region in which you want to display the contents of the bounded task flow.
4. Click **OK**.

What Happens When You Create an ADF Dynamic Region

JDeveloper adds a link to the page, as shown in the following example. JDeveloper also updates the managed bean for the ADF dynamic region and updates the corresponding task flow binding with any new parameters.

```
<af:link text="region2" action="#{RegionBean.region2}"
  id="dynamicRegionLink1"/>
```

Configuring a Page To Render an Unknown Number of Regions

At times you may not know how many ADF regions might need to be rendered at runtime. This can be handled at design time by configuring the mutiTaskFlow element of the JSF page definition file.

You can configure a page at design time when you do not know the number of regions that render in the page at runtime. For example, you want to add or remove regions to tabs using the `panelTabbed` component. Each tab that you add or remove renders a region that references a bounded task flow.

To implement this functionality, you configure the JSF page definition file's `multiTaskFlow` element to reference a list of task flow bindings contained in a managed bean of the following type:

```
oracle.adf.controller.binding.TaskFlowBindingAttributes
```

The following example shows the code for the class of a managed bean named `MultiBean` that returns a list containing two task flows.

```
package view;

import java.util.ArrayList;
import java.util.List;

import oracle.adf.controller.TaskFlowId;
import oracle.adf.controller.binding.TaskFlowBindingAttributes;

public class MultiBean {
    private List<TaskFlowBindingAttributes> mTaskFlowBindingAttrs = new
                   ArrayList<TaskFlowBindingAttributes>(5);

    public MultiBean() {
        TaskFlowBindingAttributes tfAttr = new TaskFlowBindingAttributes();
        tfAttr.setId("region1");
        tfAttr.setTaskFlowId(new TaskFlowId("/WEB-INF/r1.xml", "r1"));
        mTaskFlowBindingAttrs.add(tfAttr);

        tfAttr = new TaskFlowBindingAttributes();
        tfAttr.setId("region2");
        tfAttr.setTaskFlowId(new TaskFlowId("/WEB-INF/r2.xml", "r2"));
        mTaskFlowBindingAttrs.add(tfAttr);

    }

    public List<TaskFlowBindingAttributes> getTaskFlowList() {
        return mTaskFlowBindingAttrs;
    }
}
```

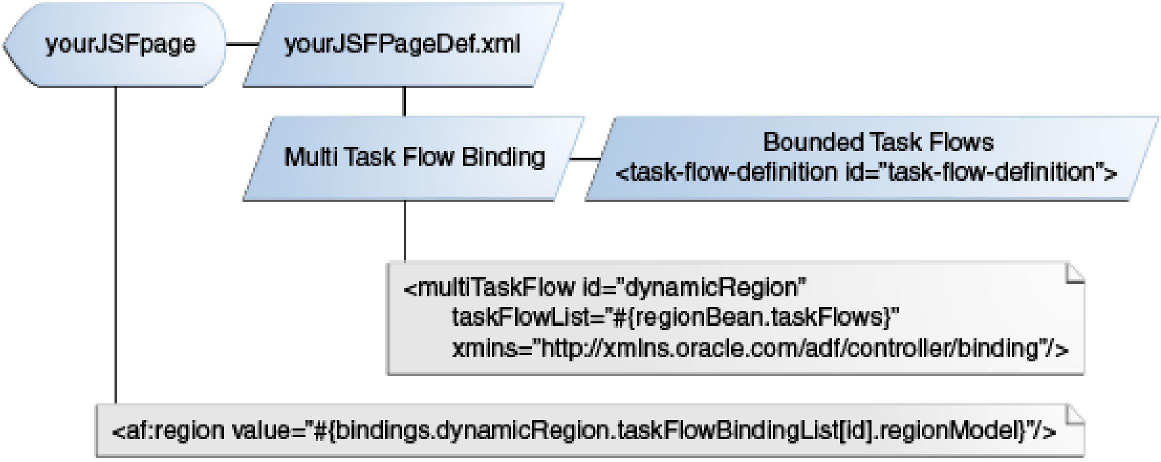
For more information about the `TaskFlowBindingAttributes` class, see the *Java API Reference for Oracle ADF Controller*.

At runtime, the `multiTaskFlow` binding references a list of one or more task flows that an end user can add or remove to the page using command components that you also expose. Figure 26-16 shows the relationships between the different parts that make up this use case.

Note:

The page fragments that the bounded task flows render must use the Facelets document type.

Figure 26-16 ADF Regions Derived from a Multi Task Flow Binding


yourJSFpage
yourJSFPageDef.xml
Multi Task Flow Binding
Bounded Task Flows
<task-flow-definition id="task-flow-definition">
<multiTaskFlow id="dynamicRegion"
taskFlowList="#{regionBean.taskFlows}"
xmlns="http://xmlns.oracle.com/adf/controller/binding"/>
<af:region value="#{bindings.dynamicRegion.taskFlowBindingList[id].regionModel}"/>

How to Configure a Page to Render an Unknown Number of Regions

Configure the JSF page definition file's `multiTaskFlow` element to reference a list of bounded task flows. This list can be modified at runtime by adding or removing elements of type `TaskFlowBindingAttributes`.

Before you begin:

It may be helpful to have an understanding of the configuration options available to you when configuring a page to render an unknown number of ADF regions. For more information, see Configuring a Page To Render an Unknown Number of Regions.

You may also find it helpful to understand other functionality that can be added using other task flow and ADF region features. For more information, see Task Flows and ADF Region Use Cases and Examples.

You will need to complete this task:

> Create a managed bean with a pageFlow scope that returns a list of type `TaskFlowBindingAttributes`. For more information about how to create a managed bean, see How to Use a Managed Bean to Store Information.

To configure a page to render an unknown number of regions:

1. In the Applications window, right-click the JSF page where you want to add or remove ADF regions at runtime and choose **Go to Page Definition**.
2. If a confirmation dialog appears, click **Yes**.
3. In the overview editor, click the **Binding and Executables** tab, and then click the **Add** icon in the Executables section.
4. In the Insert Item dialog, select **ADF Task Flow Bindings** from the dropdown menu, then select **multiTaskFlow** as the item to create and click **OK**.
5. In the Insert multiTaskFlow dialog, enter the following values:
   - **id***: Enter a unique ID for the multi task flow binding.

- **taskFlowList ***: Enter an EL expression that returns the list of `TaskFlowBindingAttributes` to return at runtime. For example, enter an EL expression similar to the following:

  `#{pageFlowScope.multiBean.taskFlowList}`

6. In the Applications window, double-click the JSF page where you want end users to add or remove ADF regions at runtime.
7. In the ADF Faces page of the Components window, from the Operations panel, drag and drop a **For Each** onto the component in the Structure window that you want to enclose the For Each component.
8. In the Properties window for the For Each component, enter values in the following fields:
   - **Items**: Enter an EL expression that references the method that return the list of task flow bindings from the multi task flow binding runtime object.

     For example, enter an EL expression similar to the following:

     `#{bindings.multiRegion1.taskFlowBindingList}`
   - **Var**: Enter a value to identify the list of task flow bindings.
9. In the JSF page that contains the For Each component, insert a Region component inside the For Each component that references the Region component's region model using an EL expression, as shown in the following example.

```
<af:forEach var="tf" items="#{bindings.multiRegion1.taskFlowBindingList}">
  <af:panelBox text="#{tf.name}" id="pb1">
    <f:facet name="toolbar"/>
    <af:region value="#{tf.regionModel}" id="r1"/>
  </af:panelBox>
</af:forEach>
```

What Happens When You Configure a Page to Render an Unknown Number of Regions

Metadata similar to that shown in the following example appears in the JSF page definition file for the multi task flow binding.

```
<multiTaskFlow id="multiRegion1"
    taskFlowList="${pageFlowScope.multiBean.taskFlowList}" activation="deferred"
    xmlns="http://xmlns.oracle.com/adf/controller/binding"/>
```

The `taskFlowList` attribute in the multi task flow binding metadata specifies the managed bean. The managed bean stores the list of objects describing the task flows that can be added to the page at runtime.

What You May Need to Know About Configuring a Page to Render an Unknown Number of Regions

- While there is no limit to the number of remote regions that you can render in a JSF page, use this capability with caution. For simple pages, where tabs are not used, regions may be combined in the page such that the maximum number of regions is determined by the design of the region and the view object queries it executes. Alternatively, for complex pages that use tabs, limit the use of regions to achieve best performance. For complex tabbed pages, ADF will not deactivate task flow transactions once a region is loaded. When switching tabs, the ongoing transaction must be stopped to achieve best performance. For details about the interaction between task flows and regions, see Configuring Transaction Management in an ADF Region.

- Each task flow binding inherits attributes defined for the multi task flow binding unless you override this behavior using the methods exposed by `TaskFlowBindingAttributes`.
- Each task flow binding inherits parameters defined in the multi task flow binding in addition to those defined using the `parametersMap` property, as described in Specifying Parameters for ADF Regions Using Parameter Maps.

Handling Access to Secured Task Flows by Unauthorized Users

When an ADF region is accessed by a user without access permission, a blank area is displayed. You can handle this by invoking a task flow that can display appropriate information or message to the user.

A bounded task flow does not render in an ADF region if it is invoked by a user who does not have the security permissions to access it. The region renders a blank area in place of the requested task flow when this occurs. This may confuse your users as it does not provide feedback as to why the task flow that they attempted to invoke does not appear. To improve the user experience, you can specify an alternative bounded task flow to render when a user attempts to access a bounded task for which they do not have the required security permissions. You can, for example, configure the alternative bounded task flow to display a page to inform users that they do not have the necessary security permissions. A more complex approach may be to render a bounded task flow that takes the users step-by-step through the process to request the necessary permissions from their system administrator.

Configure the level of security for the alternative bounded task flow to make sure that users can view it should they fail to access a bounded task flow for which they do not have security permissions. This may involve making the alternative bounded task flow public, as described in How to Make an ADF Resource Public. For more information about securing your Fusion web application, including bounded task flows, see Enabling ADF Security in a Fusion Web Application.

How to Handle Access to Secured Task Flows by Unauthorized Users

You configure an alternative bounded task flow with the content that you want to display to users when they cannot access a bounded task flow because of a lack of security permissions. You specify the alternative bounded task flow as the value for the Unauthorized Region Taskflow property in your Fusion web application's `adf-config.xml` file.

Before you begin:

It may be helpful to have an understanding of why you may need to display a message to users without security permissions for a task flow. For more information, see Handling Access to Secured Task Flows by Unauthorized Users.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Task Flows that Render in ADF Regions.

You will need to complete these tasks:

1. Create an alternative bounded task flow to display to users when they cannot access a bounded task flow for which they do not have security permissions.

   For more information about creating a bounded task flow, see Creating a Task Flow.

2. Configure this task flow so that users can access it.

   For more information, see How to Make an ADF Resource Public.

To display messages to users without security permissions for task flows:

1. In the Application Resources panel, expand the **Descriptors** and **ADF META-INF** nodes, and then double-click the **adf-config.xml** file.
2. In the overview editor, click the **Controller** navigation tab.
3. In the Controller page, in the Unauthorized Region Taskflow field, enter the name of the bounded task flow to invoke or click **Browse** to select it.

What Happens When You Handle Access to Secured Task Flows by Unauthorized Users

JDeveloper writes the name of the bounded task flow that you selected as a value for the `<unauthorized-region-taskflow>` element in the `adf-config.xml` file, as illustrated in the following example.

```
<?xml version="1.0" encoding="windows-1252" ?>
<adf-config xmlns="http://xmlns.oracle.com/adf/config"
    xmlns:config="http://xmlns.oracle.com/bc4j/configuration"
     xmlns:adf="http://xmlns.oracle.com/adf/config/properties">
...
<adf-controller-config xmlns="http://xmlns.oracle.com/adf/controller/config">
    <unauthorized-region-taskflow>/WEB-INF/task-flow-definition.xml
       </unauthorized-region-taskflow>
  </adf-controller-config>
...
</adf-config>
```

At runtime, the Fusion web application renders this alternative bounded task flow when users attempt to invoke a bounded task flow for which they do not have security permissions.

Creating Remote Regions in a Fusion Web Application

You can create an ADF remote region by invoking a task flow of a Fusion web application outside of your own application. The bounded task flow (producer) producing the remote region must have data-control-scope property set to isolated value. The task flow consuming the remote region is the consumer. A remote region can be consumed by more than one consumer task flows allowing you to reuse the remote region.

You create a remote region by invoking a task flow from another Fusion web application (the producer) in a page of your Fusion web application (the consumer). Creating a remote region allows you to reuse coarse-grained functionality in task flows that have their own data control frame rather than develop multiple task flows that implement the same functionality. It also offers performance, interactivity and ease-of-implementation advantages compared to implementing a JSF portlet in JSF pages that render ADF Faces components. At runtime, your consumer application invokes and renders a remote invocable task flow from the producer application. A remote invocable task flow may be consumed by multiple applications.

A task flow from a producer application that you configure to be invoked remotely must be a bounded task flow that renders page fragments. It must also have a value of `isolated` for the `data-control-scope` property and include the `remote-invocable` property. All of the transaction options that can be used when the `data-control-scope` property has a value of `isolated` can be configured for a task flow from a producer application that you configure to be invoked remotely. The producer and consumer applications do not share the transaction as the option chosen for the remote invocable task flow in the producer application determines the transaction behavior. For more information about the transaction options for task flows, see Managing Transactions in Task Flows.

> **Note:**
>
> The functionality described here, describing how to create a remote region to render a bounded task flow displaying JSF page fragments (`.jsff`) from another Fusion web application, differs to the functionality described in How to Call a Bounded Task Flow Using a URL. The latter section describes how you invoke a bounded task flow that renders JSF pages from another Fusion web application in a JSF page of your Fusion web application.

The producer and consumers application must run on the same version of Oracle ADF. That is, you cannot invoke a remote invocable task flow from a producer application that runs on one version of Oracle ADF and render it in a remote region of a consumer application that runs on a different version of Oracle ADF. If you attempt this type of implementation, the consumer application displays the following message to the user who attempts to launch the producer application:

```
Remote region error: Version of the richclient on the producer at "[producer
URL]" is incompatible with this consumer.
Consumer version: [Consumer Version] Producer version: [Producer Version]
```

The task flow from the producer application that you configure to be invoked remotely must not:

- Render ADF Faces components that use the Active Data Service

  For more information about the Active Data Service, see Using the Active Data Service .

- Invoke a remote invocable task flow from another application that produces remote invocable task flows. That is, you cannot configure a consumer application to invoke a remote invocable task flow from a producer application that invokes a remote invocable task flow from another producer application. A remote region cannot be nested inside another remote region.

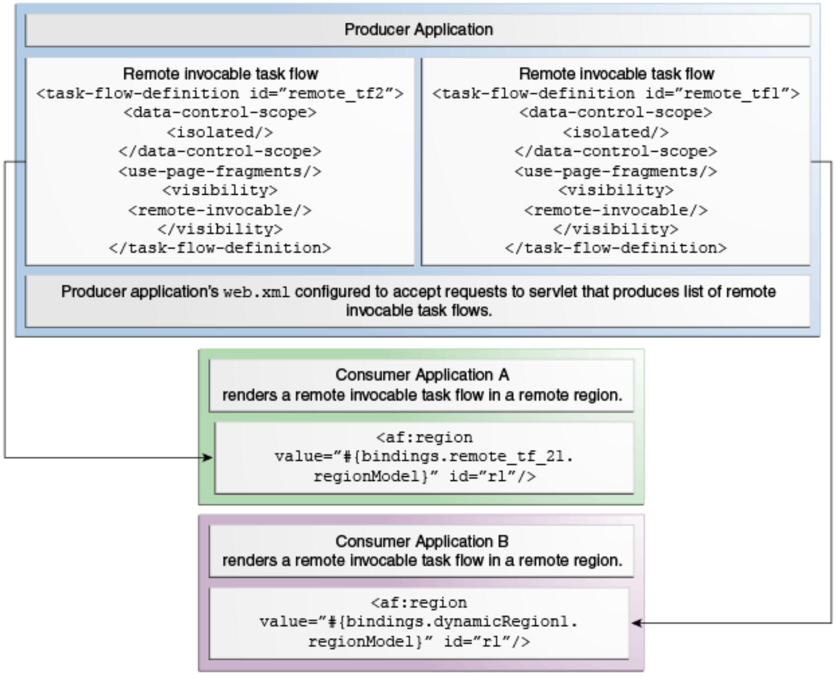
- Invoke external dialogs, as described in:
  - Running a Bounded Task Flow in a Modal Dialog. The bounded task flows that run inside these external modal dialogs must reference pages, not page fragments.
  - Using the ADF Faces Dialog Framework Instead of Bounded Task Flows.
- Use save points, as described in Using Save Points in Task Flows.
- Implement drag and drop functionality, as described in the "Adding Drag and Drop Functionality" chapter in *Developing Web User Interfaces with Oracle ADF Faces*.
- Be invoked through an iFrame. Only use Oracle ADF functionality to create a remote region.
- Assume its input parameters are pass-by-reference and visible to its caller. In a remote region implementation, a copy of all parameter values is sent to the producer of the remote invocable task flow (pass-by-value) because this producer typically resides in a different web application to the consumer application.

In addition, all of the following objects associated with the remote invocable task flow must be serializable:

- Input parameters and return parameter values defined for the task flow
- Exceptions that the task flow throws
- Any payload data that a contextual event passes

Figure 26-17 illustrates an example scenario where consumer applications render remote regions that invoke remote invocable task flows from a producer application.

Figure 26-17   Applications Rendering Remote Regions



Producer Application
Remote invocable task flow
Remote invocable task flow
Producer application's web.xml configured to accept requests to servlet that produces list of remote invocable task flows.
Consumer Application A
renders a remote invocable task flow in a remote region.
Consumer Application B
renders a remote invocable task flow in a remote region.

```
<task-flow-definition id="remote_tf2">
<data-control-scope>
<isolated/>
</data-control-scope>
<use-page-fragments/>
<visibility>
<remote-invocable/>
</visibility>
</task-flow-definition>
```

```
<task-flow-definition id="remote_tf1">
<data-control-scope>
<isolated/>
</data-control-scope>
<use-page-fragments/>
<visibility>
<remote-invocable/>
</visibility>
</task-flow-definition>
```

```
<af:region
value="#{bindings.remote_tf_21.
regionModel}" id="r1"/>
```

```
<af:region
value="#{bindings.dynamicRegion1.
regionModel}" id="r1"/>
```

Implementing the functionality described so far requires the completion of the following tasks:

- In the application that produces the task flows:
  - Configure the application's **Enable Remote Region producer support** property
  - Set the **Remote Invocable** property on the task flows that you want to render in a remote region
  - Configure the following properties for contextual events the application publishes that broadcast events to consumers in the consumer application:
    - Set **dispatchMode** to `remote`
    - Write an EL expression for **customPayload** that references a serializable object
    - Specify the type of the custom payload for the object

    Configuring these properties results in the following entries for the contextual event in the page definition where you publish the contextual event. For more information about contextual events, see Using Contextual Events.

    ```
    <event name="valueChangeEvent" dispatchMode="remote"
           customPayload="<EL to a serializable object>"
    customPayloadType=<type of the custom payload> />
    ```

    For more information about publishing the events to a remote consumer, see How to Publish Contextual Events.
  - Consider setting session timeout to a value greater than the session timeout of the consumer application so that you prevent the display of a session timeout warning from the producer application to the consumer application's end users. For more information about configuring session timeout, see Managing the HttpSession in Fusion Web Applications.
- In the application that consumes the task flows from the producer application:
  - Configure the application's **Enable Remote Region consumer support** property
  - Add the remote invocable task flows to the JSF pages where you want to render remote regions

> **Tip:**
>
> While there is no limit to the number of remote regions that you can render in a JSF page, use this capability with caution. For simple pages, where tabs are not used, regions may be combined in the page such that the maximum number of regions is determined by the design of the region and the view object queries it executes. Alternatively, for complex pages that use tabs, limit the use of regions to achieve best performance. For complex tabbed pages, ADF will not deactivate task flow transactions once a region is loaded. When switching tabs, the ongoing transaction must be stopped to achieve best performance. For details about the interaction between task flows and regions, see Configuring Transaction Management in an ADF Region.

The TaskFlowProducer project in the Summit standalone sample applications for Oracle ADF contains a remote invocable task flow that can be consumed by a consumer application. Run the `RemoteViewer.jsf` page in the TaskFlowProducer project to deploy the sample application that contains the remote invocable task flow. Once you deploy this sample application, you can consume the remote invocable task flow, as described in How to Configure an Application to Render Remote Regions.

For more information about the Summit standalone sample applications for Oracle ADF, see Running the Standalone Samples from the SummitADF_Examples Workspace.

How to Configure an Application to Produce Task Flows for Use in Remote Regions

You configure the project in the application that produces the task flows. You also configure each task flow in the project that you want to render in a remote region so that it can be invoked remotely and does not share data control frames with calling task flows.

Before you begin:

It may be helpful to have an understanding of the configuration required before you attempt to configure an application to produce task flows for use in remote regions. For more information, see Creating Remote Regions in a Fusion Web Application.

You may also find it helpful to understand functionality that can be added using other task flow and ADF region features. For more information, see Additional Functionality for Task Flows that Render in ADF Regions.

To configure an application to produce task flows for remote regions:

1. In the Applications window, right-click the project that you want to configure to produce remote invocable task flows and choose **Project Properties**.
2. Click the **ADF Task Flow** node and select the **Enable Remote Region producer support** checkbox.
3. Select **Allow Unauthenticated Remote Region Query servlet requests** if, at runtime, you want to permit users of the consumer application to query the servlet (`rtfquery`) in the producer application for the list of remote invocable task flows without providing user credentials. Also select this checkbox if you want to view the list of available remote invocable task flows from the producer application while developing the consumer application in JDeveloper (design time).

   Leave this checkbox clear to add a security constraint to the producer application's `web.xml` file that requires users who submit requests to `rtfquery` be authenticated. This is the default option. For more information, see What You May Need to Know About Securing Remote Regions.
4. In the Applications window, double-click the task flow that you want to be invoked remotely.
5. In the overview editor, click the **General** navigation tab and expand the **Visibility** section.
6. Select the **Remote Invocable** checkbox.
7. Click the **Behavior** navigation tab, expand the **Transaction** section, and select **Isolated** from the **Share data controls with calling task flow** dropdown list.

For more information about sharing data controls, see Sharing Data Controls Between Task Flows .

8. Repeat Steps 4 to 7 for each task flow that you want to be remote invocable so that it can render in a remote region.

What Happens When You Produce Task Flows for Use in Remote Regions

When you configure an application to produce remote invocable task flows, JDeveloper creates and/or modifies the following files:

- The application's `web.xml` file
- The application's policy store (the `jazn-data.xml` file)
- The source file of the task flow that you configure to be remote invocable

JDeveloper defines an ADF security policy in your producer application's `jazn-data.xml` file, as shown in the following example. JDeveloper creates this file if it does not exist in the producer application. This ADF security policy specifies a permission grant to grant access to a bootstrap page that acts as a container for the remote invocable task flow. You must define access policies for the individual remote invocable task flows that you want to appear in the bootstrap page by specifying permission grants. For more information, see How to Define Policies for ADF Bounded Task Flows. For more information about ADF Security, see Enabling ADF Security in a Fusion Web Application.

```
...
<jazn-policy>
  <grant>
    <grantee>
      <principals>
        <principal>
          <class>oracle.security.jps.internal.core.principals.JpsAnonymousRoleImpl</class>
          <name>anonymous-role</name>
        </principal>
      </principals>
    </grantee>
    <permissions>
      <permission>
        <class>oracle.adf.share.security.authorization.RegionPermission</class>
        <name>oracle.adfinternal.view.rich.bootstrap.remoteRegionBootstrapPageDef</name>
        <actions>view</actions>
      </permission>
    </permissions>
  </grant>
</jazn-policy>
...
```

JDeveloper also inserts the following properties in the source file of a task flow that you configure to be remote invocable when you set its `remote-invocable` property to `true` and the `data-control-scope` property to `isolated`, as shown in the following example. Setting the `data-control-scope` property to `isolated` means that the task flow will have its own data control frame at runtime. For more information about data controls and task flows, see Sharing Data Controls Between Task Flows .

```
...
<data-control-scope>
```

```
   <isolated/>
</data-control-scope>
...
<visibility>
   <remote-invocable/>
</visibility>
...
```

The entries that JDeveloper adds or configures in the application's `web.xml` file include:

- Filters (and associated filter mappings) named `remoteApplication`, `ServletADFFilter`, `ADFLibraryFilter` and `adfBindings`
- Servlets (and associated servlet mappings) named `Faces Servlet`, `adflibResources` and `rtfqueryServlet`

Example 26-2 shows these entries in a producer application configured to produce task flows for remote regions. In addition to the entries shown in Example 26-2, JDeveloper also generates a `<security-constraint>` and `<login-config>` element when you configure an application to produce remote invocable task flows. These elements are not shown in Example 26-2. For information about the values specified for these elements, see What You May Need to Know About Securing Remote Regions.

Note:

In the `web.xml` file, make sure that the `remoteApplication` filter appears:

- Immediately after the `JpsFilter` filter if this latter filter is configured.

  For more information about the `JpsFilter` filter, see What You May Need to Know About ADF Authentication.
- Before the `ADFLibraryFilter` filter, as shown in Example 26-2.

Example 26-2 Entries in the web.xml File of an Application Producing Remote Invocable Task Flows

```
<filter>
   <filter-name>remoteApplication</filter-name>
   <filter-class>oracle.adfinternal.view.rich.remote.RemoteApplicationFilter</filter-class>
</filter>
...
<filter>
   <filter-name>ADFLibraryFilter</filter-name>
   <filter-class>oracle.adf.library.webapp.LibraryFilter</filter-class>
</filter>
...
<filter-mapping>
   <filter-name>remoteApplication</filter-name>
   <url-pattern>/rr/*</url-pattern>
   <dispatcher>REQUEST</dispatcher>
</filter-mapping>
...
<filter-mapping>
   <filter-name>ADFLibraryFilter</filter-name>
   <url-pattern>/*</url-pattern>
   <dispatcher>FORWARD</dispatcher>
   <dispatcher>REQUEST</dispatcher>
</filter-mapping>
...
<servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
...
<!-- The remote region query servlet. -->
<servlet>
   <servlet-name>rtfqueryServlet</servlet-name>
   <servlet-class>oracle.adfinternal.controller.rtfquery.RemoteTaskFlowQueryServlet</servlet-
class>
   <load-on-startup>1</load-on-startup>
</servlet>
...
<!-- The servlet name here should be the same as the Faces Servlet. -->
<servlet-mapping>
   <servlet-name>Faces Servlet</servlet-name>
   <url-pattern>/rr/*</url-pattern>
</servlet-mapping>
...
<servlet-mapping>
  <servlet-name>rtfqueryServlet</servlet-name>
  <url-pattern>/rtfquery</url-pattern>
</servlet-mapping>
```

What You May Need to Know About Securing Remote Regions

By default, JDeveloper adds the `<security-constraint>` element shown in the following example when you configure an application to produce remote invocable task flows for remote regions. This security constraint restricts access to the remote region query servlet (`rtfquery`) to those users who have the role specified by the `<role-name>` element.

```
<security-constraint>
  <web-resource-collection>
     <web-resource-name>rtfquery</web-resource-name>
     <url-pattern>/rtfquery</url-pattern>
  </web-resource-collection>
  <auth-constraint>
     <role-name>valid-users</role-name>
  </auth-constraint>
</security-constraint>
```

You can remove this constraint from the `web.xml` file by selecting the **Allow Unauthenticated Remote Region Query servlet requests** checkbox when you configure an application to produce remote invocable task flows. For more information about how to access this checkbox, see How to Configure an Application to Produce Task Flows for Use in Remote Regions.

JDeveloper also generates the authentication mechanism shown in the following example when you configure an application to produce remote invocable task flows. This authentication mechanism determines if access to an ADF resource (for example, a task flow) that has been secured is permitted. The value `CLIENT-CERT` must appear here so that the producer application can authenticate clients that request access to a secured resource. The producer application ignores this setting if a client requests access to a non-secured resource.

```
<login-config>
  <auth-method>CLIENT-CERT,FORM</auth-method>
  <realm-name>Identity Assertion</realm-name>
</login-config>
```

In producer applications that define the security constraint (the default behavior), the value for the `<auth-method>` element in the associated authentication mechanism must at a minimum include `CLIENT-CERT`. If you make other changes that modify your `web.xml` file (for example, configuring ADF Security) after you configure an application to produce remote invocable task flows, you might overwrite the values required to make remote invocable task flows accessible to authenticated users. Verify the values specified for the security constraint and the authentication mechanism to make sure they are valid.

For more information about security, see Enabling ADF Security in a Fusion Web Application.

How to Configure an Application to Render Remote Regions

In the Fusion web application that you want to render remote regions, you configure a project so that it can consume remote invocable task flows from the producer application. You also create a connection to the producer application.

Once you have created this connection, you can drag a remote invocable task flow from wherever you defined the connection (Resources window or Application Resources panel) and drop it on the JSF page or page fragment where you want the remote region to render at runtime in your consumer application's page. The producer application of the remote invocable task flow that you want to invoke must be running in order for the remote invocable task flow to appear in the connection.

Before you begin:

It may be helpful to have an understanding of the configuration required before you attempt to configure an application to render remote regions. For more information, see Creating Remote Regions in a Fusion Web Application.

You may also find it helpful to understand functionality that can be added using other task flow and ADF region features. For more information, see Additional Functionality for Task Flows that Render in ADF Regions.

To configure an application to render a remote region:

1. In the Applications window, right-click the project that you want to configure to render a remote region and choose **Project Properties**.
2. Click the **ADF Task Flow** node and select the **Enable Remote Region consumer support** checkbox.
3. In the Resources window, click **New**, select **New Connection** and then **Remote Region Producer**.
4. In the Create Remote Region Connection dialog, select **Application Resources** if you want the connection to be available only within the application. Select **Resources window** if you want the connection to be available to other applications using the Resources window.

5. In the **Name** field, enter a name that the application will use to identify the connection to the producer application of the remote invocable task flows.
6. In the **URL Endpoint** field of the **Query URL** section, enter the URL endpoint for the Fusion web application that produces the remote invocable task flows. Typically, this URL has a format similar to the following:

   ```
   http://<host>:port/<context root>/rtfquery
   ```

   where `rtfquery` is the servlet that provides the list of remote invocable task flows in the producer application.
7. Click **Test Connection** to verify that the URL endpoint is valid.
8. In the **URL Endpoint** field of the **Invoke URL** section, enter the URL endpoint of the Fusion web application that produces the remote invocable task flows. Typically, this URL has a format similar to the following:

   ```
   http://<host>:port/<context root>/rr
   ```

   where `rr` is the `url-pattern` specified for the `remoteApplication` filter in the `web.xml` file of the Fusion web application that produces the remote invocable task flows.
9. If you know that the producer application requires user credentials before it will allow the consumer application access a remote invocable task flow, select the **Always Send User Identity** checkbox. This generates an identity token that the consumer application transmits to the producer application during the initial connection.

   Leave the **Always Send User Identity** checkbox clear if you do not know if the producer application requires user credentials. For more information, see What Happens at Runtime: How an Application Renders Remote Regions.
10. Click **Test Connection** to verify that the URL endpoint is valid and click **OK**.
11. Expand the **Remote Region Producer** node and drag the remote invocable task flow that you want your Fusion web application to invoke and drop it on the JSF page or page fragment where you want it to render in a remote region at runtime.
12. In the context menu that appears, select the type of region that you want to create in the page (**Region** or **Dynamic Region**).

    If you select **Dynamic Region**, you need to choose or create a managed bean that determines at runtime what bounded task flow to render in the ADF dynamic region. For more information, see Creating ADF Dynamic Regions.
13. In the Edit Task Flow Binding dialog, from the **Remote Connection** dropdown list, select the connection to the Fusion web application that produces the remote invocable task flow that you defined in Step 5.

    Typically, this is selected by default.
14. Click **OK**.

What Happens When You Configure an Application to Render Remote Regions

JDeveloper adds a context parameter to the application's `web.xml` file when you enable remote region consumer support in the project that consumes remote invocable task flows, as shown in the following example.

```
<context-param>
  <param-name>oracle.adf.view.rich.remote.AUTH_PROVIDER</param-name>
  <param-value>oracle.adf.view.rich.remote.security.ADFAuthorizationProvider</param-value>
</context-param>
```

This context parameter specifies the default authorization provider that is configured to work with ADF Security. You can add an alternative authorization provider by creating an object that implements the following interface:

`oracle.adf.view.rich.remote.AuthorizationProvider`

When you drag and drop a remote invocable task flow to the JSF page or page fragment in your application, JDeveloper adds an `af:region` tag to the JSF page and a task flow binding to the page definition file of the JSF page. These additions are similar to those generated when you drag and drop a task flow from the Applications window to a JSF page in the same application. One difference is that when you drag and drop a remote invocable task flow to a JSF page, the task flow binding that is added to the page definition file of the JSF page includes a `remoteConnectionName` attribute that references the connection name of the producer application, as shown in Example 26-3.

The task flow binding's `remoteConnectionName` attribute can also reference an EL expression that returns the name of the connection to the producer application at runtime. This is useful, for example, if the name of the producer application changes and you want to refresh the ADF region in response to this event. For more information, see Configuring an ADF Region to Refresh.

For more information about the entries that JDeveloper generates when you create an ADF region or an ADF dynamic region, see What Happens When You Create an ADF Region , and What Happens When You Create an ADF Dynamic Region.

Example 26-3   Metadata Added to Page Definition to Create a Task Flow Binding

```
<taskFlow id="dynamicRegion2"
      taskFlowId="${viewScope.ParentActionRegion.dynamicTaskFlowId}"
      activation="deferred" remoteConnectionName="RemoteAppProducingTFs"
      xmlns="http://xmlns.oracle.com/adf/controller/binding"/>
```

What Happens at Runtime: How an Application Renders Remote Regions

The initial runtime behavior of the applications that you have configured to produce remote invocable task flows and render remote regions depends on the security-related options configured for the producer application:

- It requires user credentials from the consumer application before it permits access to the `rtfquery` servlet to retrieve the list of task flows that can be invoked remotely.

  In this scenario, the producer application responds to the initial request with an error code. In response, the consumer application generates an encrypted identity token that it transmits to the producer application. The producer application uses this identity token to authenticate the consumer application and, if successful, serves the consumer application's request.

  If you know in advance that the producer application requires user credentials, you can make the consumer application submit the encrypted identity token with the initial connection by selecting the **Always Send User Identity** checkbox, as described in How to Configure an Application to Render Remote Regions. This optimizes the connection process.

- It allows unauthenticated requests to the `rtfquery` servlet.

  The consumer application retrieves the list of remote invocable task flows from the `rtfquery` servlet that the producer application configures to be invoked remotely.

After accessing the remote invocable task flow from the producer application, the consumer application renders the content from the task flow in an ADF region (the remote region) that is configured for this purpose. The appearance of this content is defined by the style properties in the ADF skin of the consumer application. The style properties in this ADF skin must match the style properties defined in the producer application's ADF skin. For more information, see What You May Need to Know About ADF Skins and Remote Regions.

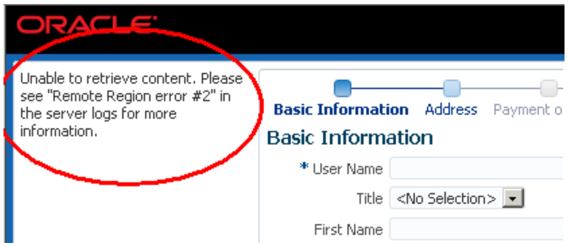
If the consumer application invalidates the session that it established with the producer application, the producer application rolls back any transaction that is in progress at the time that the consumer application invalidates the session. Examples of events that can cause the consumer application to invalidate a session include:

- Reaching the time specified for session timeout in the consumer application
- An end user logging out of the consumer application
- An end user clicking the browser's back button
- Navigation to a different page that does not render the remote region

How to Add Custom Error Messages for Remote Regions

A variety of error messages appear in the consumer application if an error occurs in a remote region implementation. For example, you may see a message similar to that shown in Figure 26-18.

Figure 26-18 Error Message for Remote Region



Unable to retrieve content. Please see "Remote Region error #2" in the server logs for more information.

The following examples show log messages that correspond to the message that appears in Figure 26-18.

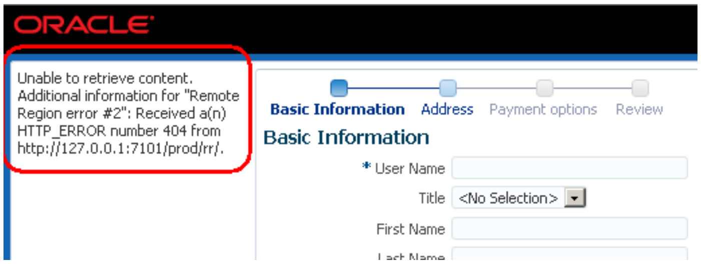
```
...
<oracle.adf.view> <RegionRenderer> <_logError> <RR_HTTP_ERROR>
...
...
<oracle.adf.view> <RegionRenderer> <_logError> <Jan 29, 2014 12:50:51 PM
oracle.adfinternal.view.faces.renderkit.rich.RegionRenderer _logError
ALL: Remote Region error #2: An uncaught exception was thrown on the producer at "http://
127.0.0.1:7101/prod/rr/" while trying to run an ADF Taskflow.
>
```

```
oracle.adf.controller.ControllerException: ADFC-14032: A required remote task flow input
parameter value was not specified. Requested remote taskFlowId
 '/WEB-INF/task-flow-definition.xml#task-flow-definition'; requested from
'localhost.localdomain'.
    at
oracle.adf.controller.internal.binding.TaskFlowRegionModelRemoteProducer.createRegionViewPortCont
ext(TaskFlowRegionModelRemoteProducer.java:173)
    at
oracle.adf.controller.internal.binding.TaskFlowRegionModelViewPort.getViewPort(TaskFlowRegionMode
lViewPort.java:649)
...
```

In many scenarios, the message in Figure 26-18 and log file entries may be sufficient to diagnose and resolve an issue with a remote region implementation. In other scenarios, you may want to present additional information to communicate to end users. Figure 26-19 shows a revised version of the consumer application page in Figure 26-18 that now displays additional information, such as the error type and the location of the producer application. Configuring your consumer application to show additional information, as in Figure 26-19, requires you to add a context parameter to the `web.xml` file of your consumer application and to write a Java class in the same application that extends `RemoteRegionErrorListener` from the following package:

```
oracle.adf.view.rich.model
```

Figure 26-19 Custom Error Message for Remote Region



Unable to retrieve content.
Additional information for "Remote Region error #2": Received a(n) HTTP_ERROR number 404 from http://127.0.0.1:7101/prod/rr/.

To add a custom error message for a remote region:

1. In the user interface project of the consumer application, create a Java class that extends the `oracle.adf.view.rich.model.RemoteRegionErrorListener` class.

   The following example shows a Java class that listens for errors on a remote region and returns a string with details of the error message.

```
package example;

import oracle.adf.view.rich.model.RemoteRegionErrorEvent;
import oracle.adf.view.rich.model.RemoteRegionErrorListener;
import oracle.adf.view.rich.model.RemoteRegionHttpErrorEvent;

public class MyRemoteRegionErrorListener extends RemoteRegionErrorListener {

    @Override
    public String getAdditionalUserInformation(RemoteRegionErrorEvent event)
{

        RemoteRegionErrorEvent.Type type = event.getErrorType();
        StringBuilder sb =
            new StringBuilder().append("Received a(n) ").append(type);
        if (RemoteRegionErrorEvent.Type.HTTP_ERROR.equals(type)) {
            sb.append(" number
").append(((RemoteRegionHttpErrorEvent)event).getHttpErrorCode());
        }

        sb.append(" from ").append(event.getProducerURL());

        String viewId = event.getViewID();
        if (null != viewId) {
            sb.append(" for viewId ").append(viewId);
        }

        return sb.toString();
    }
}
```

2. In the Applications window, in the user interface project, expand the **Web Content** and **WEB-INF** nodes, and then double-click **web.xml**.
3. In the overview editor for the `web.xml` file, click the **Application** navigation tab and in the Context Initialization Parameters section, click the **Create** icon to add a parameter that references the Java class you created in Step 1 as follows:

   Name: Enter the context parameter `oracle.adf.view.rich.REMOTE_REGION_ERROR_LISTENER`.

   Value: Enter the path to the Java class that you created in Step 1. For example, to reference the class in Step 1, enter `example.MyRemoteRegionErrorListener`.

What You May Need to Know About ADF Skins and Remote Regions

The style properties in the ADF skins of the consumer and producer applications must match in order for the remote region to render. If the ADF skins do not match, the remote region does not render its content. If the ADF skins match, the remote region renders and uses the consumer application's ADF skin to determine the look and feel.

If mismatches occur between the ADF skins for the consumer and producer applications, you need to modify one or both ADF skins so no mismatches occurs. Where mismatches occur the consumer application's client displays a message similar to the following:

```
Unable to retrieve content. Please see "Remote Region Error #1" in the
server logs for more information.
```

The server logs refers to messages that appear in the log files of the application server for the consumer application. These log messages may provide information that you can use to identify mismatches between ADF skins or to diagnose issues with the ADF skin. The following list describes issues that result in the consumer application generating log messages:

- `ADF_FACES-30209` identifies a mismatch between the ADF skins. It means that while the producer application can find the skin that the consumer application uses, the ADF skins are not the same. A message similar to the following appears:

  ```
  Remote Region Error #1: Producer at "http://myproducer.com/rr" does
  not have a skin that matches this consumer. Consumer skin information:
  {...} Producer skin information: {...}.
  ```

The ADF skin information provided contains a checksum of the ADF skins and lists all of the skinning extensions (including checksum) and features employed by each ADF skin. The consumer and producer applications should have the same ADF skins and extensions. Using this data may help identify if there are differences between the two applications.

- `ADF_FACES-30210` indicates that the producer application does not contain the ADF skin that the consumer application uses. A message similar to the following appears:

  ```
  Remote Region Error #1: Producer at "http://myproducer.com/rr" does
  not have the following skin: "simple"
  ```

- Producer application does not have a skin factory. This means that the producer application is not configured properly and cannot find an instance object that extends `SkinFactory` from the following package:

  ```
  org.apache.myfaces.trinidad.skin
  ```

  A message similar to the following appears:

  ```
  Remote Region Error #1: No skin factory available on the producer at
  "http://yourproducer.com/rr"
  ```

The consumer application logs the skin debug information using a `VERBOSE` setting.

> **Note:**
>
> The context parameter `oracle.adf.view.rich.remote.PRODUCER_STYLESHEET_DEBUG_LEVEL` that determines the level of log output is deprecated in release 12.2.1.4.0.

For a general overview of ADF skins, see the "Customizing the Appearance Using Styles and Skins" chapter in *Developing Web User Interfaces with Oracle ADF Faces*.

27

Creating Complex Task Flows

This chapter describes how to use advanced features of ADF task flows that you create in Fusion web applications. These include the ability to define transaction options for a task flow, configure reentry options to a task flow for a user that previously exited, handle exceptions, configure save points for your task flow, use ADF Faces components such as `train`, and create task flows from task flow templates.

This chapter includes the following sections:

About Creating Complex Task Flows

ADF task flows allow you to add features such as handling exceptions, reentry to the bounded task flow, using save points, and so on. Such task flows can be used to perform advance and complex activities.

After creating a task flow, adding activities to it, and configuring control flow between the activities, you can extend the task flow´s functionality by adding some of the features described in the following list:

- Bounded task flows can be configured to include additional transactional management facilities beyond those provided by the underlying data controls so that a Fusion web application can commit or roll back the data controls associated with the task flow's activities as a group.
- Reentry options can be configured for bounded task flows to determine the response if a user clicks the browser's back button.
- Task flows contain a number of features that allow you to proactively handle exceptions. In addition, you can write custom code to handle exceptions thrown by a task flow.
- Task flows can also be configured to capture the state of a Fusion web application at a particular instance using save points. These allow you to save

the application's state if, for example, a user leaves a page without finalizing a task.

- Train components are ADF Faces components that render navigation items to guide a user through a multistep process. You can configure a bounded task flow to render these navigation items by default for the view activities in a task flow.
- Task flow templates can serve as a starting point for the creation of task flows that share common elements.
- Create a page hierarchy from the view activities that the default unbounded task flow in your application (and additional task flows in the application) reference.
- Invoke initializers and finalizers when a bounded task flow is entered and exited. An **initializer** is custom code that is invoked when a bounded task flow is entered. A **finalizer** is custom code that is invoked when a bounded task flow is exited via a task flow return activity or because an exception occurred. The finalizer is a method on a managed bean. Common finalizer tasks include releasing all resources acquired by the bounded task flow and performing cleanup before exiting the task flow.

  You specify both the initializer and the finalizer as an EL expression for a method on a managed bean, for example:

  ```
  #{pageFlowScope.modelBean.releaseResources}
  ```

  There are two techniques for running initializer code at the beginning of the bounded task flow, depending on whether or not the task flow may be reentered via a browser back button:

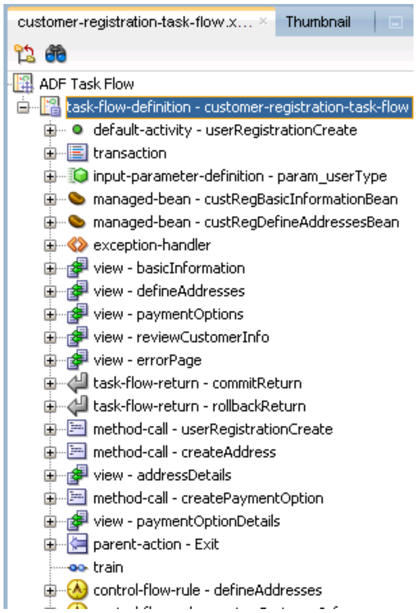
  – No reentry via back button expected: Designate a method call activity as the default activity (first to execute) in the bounded task flow. The method call activity calls a custom method containing the initializer code. See Using Method Call Activities and What You May Need to Know About the Browser Back Button and Navigating Through Records.

  – Back button reentry possible: Specify an initializer method using an option on the bounded task flow metadata. See Creating a Task Flow. Use this technique if you expect that a user may reenter the task flow using a browser back button. In such a case, the designated default activity for the bounded task flow may never be called, but a method designated using the Initializer method will.

Complex Task Flows Use Cases and Examples

Figure 27-1 shows the Structure window for a task flow that implements many of the use cases discussed in About Creating Complex Task Flows. For example, it specifies an error page to handle errors, makes use of the ADF Faces `train` component to render navigation items, and specifies two task flow return activities that either commit or roll back changes that the task flow makes as part of transaction management.

Figure 27-1 Task Flow in the Structure Window



```
ADF Task Flow
  task-flow-definition - customer-registration-task-flow
    default-activity - userRegistrationCreate
    transaction
    input-parameter-definition - param_userType
    managed-bean - custRegBasicInformationBean
    managed-bean - custRegDefineAddressesBean
    exception-handler
    view - basicInformation
    view - defineAddresses
    view - paymentOptions
    view - reviewCustomerInfo
    view - errorPage
    task-flow-return - commitReturn
    task-flow-return - rollbackReturn
    method-call - userRegistrationCreate
    method-call - createAddress
    view - addressDetails
    method-call - createPaymentOption
    view - paymentOptionDetails
    parent-action - Exit
    train
    control-flow-rule - defineAddresses
```

Additional Functionality for Complex Task Flows

You may find it helpful to understand other **Oracle ADF** features before you configure or use task flows. Additionally, you may want to read about what you can do with your task flows. Following are links to other functionality that may be of interest.

- You can specify exception handlers to manage errors that occur during execution of task flows. For more information about error handling, see Customizing Error Handling.
- Task flow save points can save application state. For more information about **application state management**, see Using State Management in a Fusion Web Application.
- Bounded task flows can be secured by defining the privileges that are required for someone to use them. For more information, see Enabling ADF Security in a Fusion Web Application.
- You can use Business Process Execution Language (BPEL) with task flows to:
  - Invoke a BPEL process from an unbounded or bounded task flow to perform a function or use services
  - Call a bounded task flow from a BPEL process in order to model user interactions with a web interface

  You can use a task flow method call activity to invoke a managed bean method or a web service that interacts with a BPEL process. For more information about the task flow method call activity, see Using Method Call Activities.
- You can write custom code that extends your task flows. For example, you can write custom code to invoke when a task flow throws an exception, as described in How to Designate Custom Code as an Exception Handler. For information about the APIs that you can use to write custom code, see the following reference documents:

- – *Java API Reference for Oracle ADF Controller*
- – *Java API Reference for Oracle ADF Faces*

Sharing Data Controls Between Task Flows

ADF task flows allow you to share data controls between one or more task flows. Either an instance of the data control or two separate, isolated, data controls can be shared. This can be used for making transaction management options when committing or rolling back. To share the data controls you must set a data-control-scope value to shared or isolated on the called bounded task flow.

When one task flow calls another, the task flows can either share an instance of a data control, or create two separate isolated instances of the same data control so the task flows can maintain independent state. The internal object that task flows use to share their data controls or to store their own isolated data control is known as a **data control frame**.

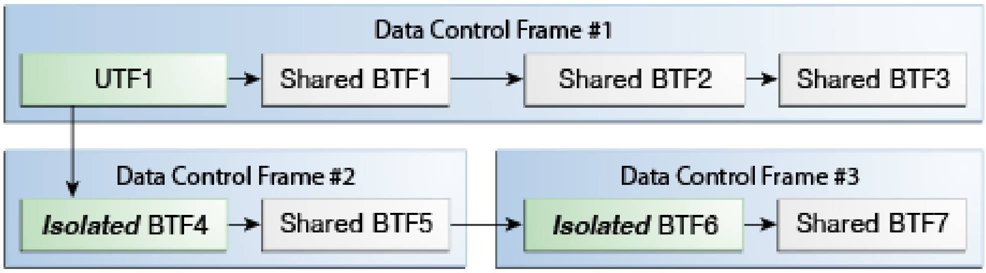
Task flows making use of the task flow transaction management options when committing or rolling back use the data control frame to know which data controls to perform the transaction operations on. A data control frame is created at runtime for your application's unbounded task flow and any bounded task flow with a `data-control-scope` value of `isolated`. When a task flow specifies a `data-control-scope` value of `shared`, the called task flow uses the data control frame of the calling task flow rather than creating its own. This allows the called task flow to share data control instances attached to the data control frame. Alternatively, if a called task flow specifies a `data-control-scope` value of `isolated`, a new data control frame is created and a new instance of any data controls used by the bounded task flow will be attached to the newly-created data control frame.

To specify whether data controls are shared between the calling and called task flows, you must set a `data-control-scope` value of either `shared` or `isolated` on the called bounded task flow. The default value is `shared`.

If `data-control-scope` is set to `shared` on both the calling and called bounded task flow, the data control frame used will be the one for the calling task flow.

In Figure 27-2, the bounded task flows BTF1, BTF2, and BTF3 use the data control frame created for the application's unbounded task flow (UTF1) because these bounded task flows all have a `data-control-scope` value of `shared`. In contrast, the bounded task flow BTF4 uses a different data control frame because it has a `data-control-scope` value of `isolated`. It shares this data control frame with the bounded task flow that it calls (BTF5) because this latter bounded task flow has a `data-control-scope` value of `shared`.

Figure 27-2 Sharing and Isolating Data Control Frames



The value that you set for the `data-control-scope` element of a called task flow determines if a calling task flow and a called task flow share flow data controls. Other factors, such as whether a task flow renders in a page or page fragment, do not change this behavior.

How to Share a Data Control Between Task Flows

You share data controls between task flows by specifying a value for the `data-control-scope` element of the called task flow.

Before you begin:

It may be helpful to have an understanding of the properties that determine how you share data controls between task flows. For more information, see Sharing Data Controls Between Task Flows .

You may also find it helpful to understand functionality that can be added using other task flow features. For more information, see Additional Functionality for Complex Task Flows.

To share a data control between task flows:

1. In the Applications window, double-click the called task flow.
2. In the Properties window expand the **Behavior** section and select **Shared** from the **Share data controls with calling task flow** list.

> **Note:**
>
> After you select **Shared** from the **Share data controls with calling task flow** list, you may need to configure transaction options for the task flow. For more information, see What You May Need to Know About Sharing Data Controls and Managing Transactions.

What Happens When You Share a Data Control Between Task Flows

JDeveloper writes an entry in the source file for the called task flow when you select **Shared** from the **Share data controls with calling task flow** list, as illustrated in the following example.

```
<task-flow-definition id="task-flow-definition">
    ...
    <data-control-scope id="__1">
      <shared/>
    </data-control-scope>
    ...
  </task-flow-definition>
```

What Happens at Runtime: How Task Flows Share Data Controls

An end user can open the same application more than once in different browser windows or browser tabs. If an end user does this, each window or tab has its own view port, and each window or tab is isolated from all other windows or tabs within the same HTTP session. As a result, the windows or tabs do not share data controls and, therefore, state. The windows or tabs start with a new data control frame as if they were running with an isolated data control scope. The exception to this rule is if an end user opens a modal dialog in a secondary browser window, then the primary and secondary browser windows have the same server-side state and share data controls. For more information about using modal dialogs, see Running a Bounded Task Flow in a Modal Dialog. For more information about view ports, see About View Ports and ADF Regions.

Data controls are not instantiated until needed. One of two things can occur when a parent task flow calls a child task flow that specifies `data-control-scope` as `shared` (the default) and references a data control named D:

1. If a data control named D already exists in the parent task flow's data control frame, then the child task flow's reference to D resolves to the existing data control named D that is in scope.
2. If a data control named D does not exist in the parent task flow's data control frame, then Oracle ADF instantiates the first data control named D that it finds when performing a top-down search through the `DataBindings.cpx` files.

The above rules are unlikely to be applied because most applications define data controls with unique names. However, if a task flow from an application calls a task flow in an ADF Library JAR, the application and the ADF Library JAR task flows may reference data controls that use the same names.

If, when the parent task flow calls the child task flow, the data control D from the parent task flow's ADF Library JAR is already instantiated, then the child task flow uses it. If it is not already instantiated, then the data control D from the parent task flow will be instantiated using the first reference for this data control name found during a top-down search through the `DataBindings.cpx` files. In particular, in this task flow calling scenario, the child task flow's data control D will not be instantiated.

Another scenario to consider is where a task flow in a JSF page calls a task flow that renders in a region in the JSF page. If the calling task flow in the JSF page instantiates a data control D before the region renders in the page, the calling task flow's data control D is used. However, if the region is the first databound component in the JSF page, the called task flow's data control D will be instantiated and used.

For more information about data controls, see Exposing Application Modules with ADF Data Controls. For more information about the `DataBindings.cpx` file, see Working with the DataBindings.cpx File .

Managing Transactions in Task Flows

The transactions on the ADF bounded task flows can be managed by configuring the transaction type. If No Controller Transaction is chosen, then no transaction such as begin, commit, or roll back is performed. If one of these options: Always Begin New Transaction, Always Use Existing Transaction, or Use Existing Transaction if Possible is chosen then the transaction will either create a new transaction, participate in the exiting transaction, or if a transaction is in progress, then the task flow participates in this existing transaction or the task flow will create a new transaction when the bounded task flow is initiated.

In the context of systems relating to databases, a **transaction** is a persisted collection of work items that can be committed or rolled back together as a group. ADF, through technologies such as **ADF Business Components**, offers all the advantages that transactions provide. However, support for transactions does not stop at the ADF

Model layer. In the ADF Controller layer, bounded task flows optionally provide their own abstract implementation of transactions over the underlying ADF Model layer. This implementation allows you to control transactions from the task flow and also declaratively manage the transaction boundaries.

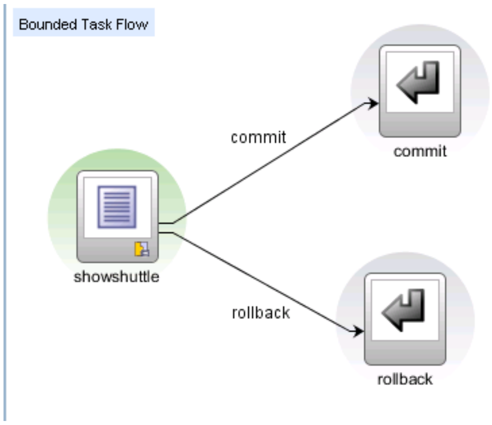
The transaction options that you can configure on a called bounded task flow can be grouped into two categories:

- You choose the **No Controller Transaction** option so the called bounded task flow does not perform a transaction operation (begin, commit, or roll back a transaction).
- You use one of the following ADF task flow transaction management options:
  - **Always Begin New Transaction**: A new transaction starts when the bounded task flow is entered, regardless of whether or not a transaction is in progress. The new transaction completes when the bounded task flow exits.
  - **Always Use Existing Transaction**: When called, the bounded task flow participates in an existing transaction already in progress.
  - **Use Existing Transaction If Possible**: When called, the bounded task flow either participates in an existing transaction if one exists, or starts a new transaction upon entry of the bounded task flow if one does not exist.

If a called bounded task flow starts a new ADF task flow transaction (based on the `transaction` option that you selected), you can specify whether the transaction commits or rolls back when the task flow returns to its caller. The commit and rollback options are set on the task flow return activity that returns control back to the calling task flow. The same task flow that starts a transaction must also resolve the transaction.

In a called bounded task flow, you can specify two different return task flow activities that result in either committing or rolling back a transaction in the called bounded task flow. Each of the task flow return activities pass control back to the same calling task flow. The difference is that one task flow return activity specifies the commit option, while the other specifies the rollback option. Figure 27-3 shows the `orders-select-many-items.xml` task flow from the Summit ADF task flow sample application. If transaction processing successfully completes, control flow passes to the commit task flow return activity, which specifies options to commit the transaction. If the transaction is cancelled before completion, the rollback task flow activity specifies options to roll back the transaction.

Figure 27-3 Task Flow Return Activities in Called Bounded Task Flow

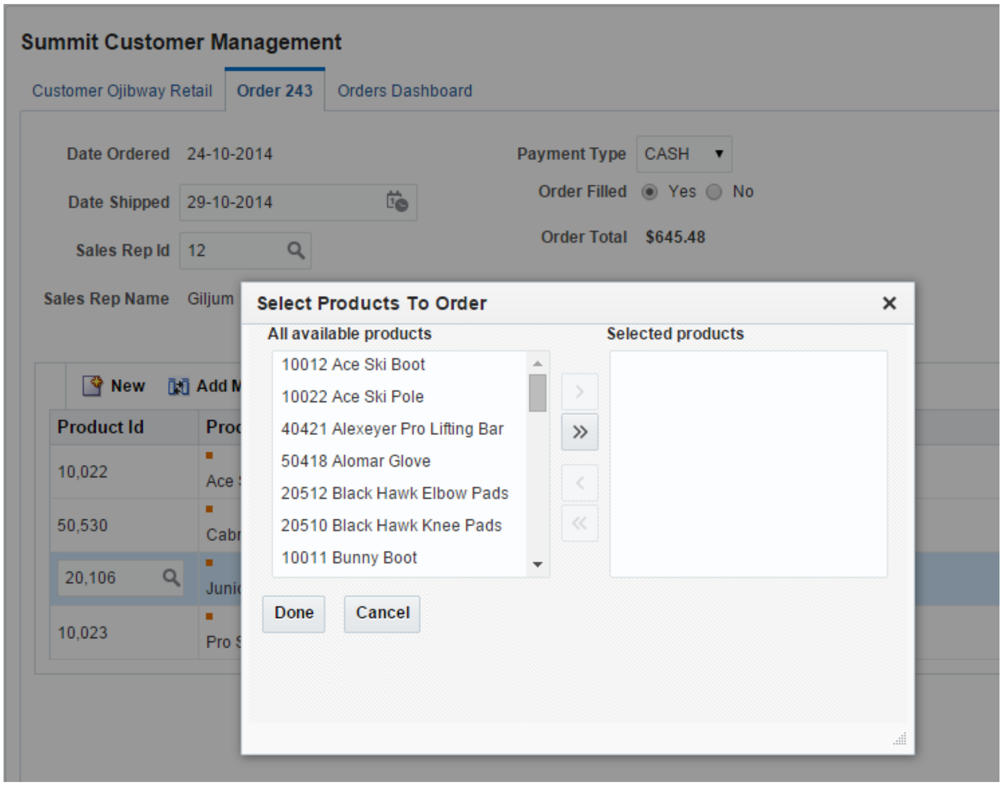


Use the `restore-save-point` option on the task flow return activity if you want to discard the changes an end user makes within a called bounded task flow when the called bounded task flow exits. ADF Controller rolls back to the previous ADF Model save point that was created when the bounded task flow was entered. The `restore-save-point` option applies only to cases when a bounded task flow is entered by joining an existing transaction (either the `requires-existing-transaction` or `requires-transaction` option is also specified) and a save point is created upon entry.

If you use the ADF task flow transaction management features that commit and rollback the data controls associated with the data control frame of the current task flow, you must use task flow return activities with their End Transaction property set to `commit` or `rollback`, or programmatically commit the associated data control frame. Alternatively if you use the No Controller Transaction setting or you only want to commit or rollback one data control, use the associated commit or rollback operations from the Data Controls panel or programmatically execute the associated commit and rollback bindings.

In the Summit sample application for ADF task flows, a number of task flows create transactions. For example, the `orders-select-many-items.xml` task flow uses the ADF task flow transaction management's **Use Existing Transaction if Possible** option. This task flow allows an end user to add multiple items to an order and commit or rollback this change when the end user interacts with the `showshuttle.jsf` page shown in Figure 27-4.

Figure 27-4   Task Flow Transaction



Summit Customer Management

Customer Ojibway Retail | Order 243 | Orders Dashboard

Date Ordered 24-10-2014
Date Shipped 29-10-2014
Sales Rep Id 12
Sales Rep Name Giljum

Payment Type CASH ▾
Order Filled ● Yes ○ No
Order Total $645.48

New | Add M

Product Id	Proc
10,022	Ace
50,530	Cabr
20,106	Juni
10,023	Pro S

Select Products To Order ×

All available products
- 10012 Ace Ski Boot
- 10022 Ace Ski Pole
- 40421 Alexeyer Pro Lifting Bar
- 50418 Alomar Glove
- 20512 Black Hawk Elbow Pads
- 20510 Black Hawk Knee Pads
- 10011 Bunny Boot

Selected products

How to Enable Transactions in a Bounded Task Flow

You define the transaction options on a bounded task flow that is called by another task flow. Add a task flow return activity on the called bounded task flow that returns control to the task flow that calls the bounded task flow.

Before you begin:

It may be helpful to have an understanding of what a transaction is and how you can configure it. For more information, see Managing Transactions in Task Flows.

You may also find it helpful to read about additional functionality that you can add using other task flow features. For more information, see Additional Functionality for Complex Task Flows.

To enable a bounded task flow to run as a transaction:

1. In the Applications window, double-click the called bounded task flow.
2. In the editor window, click the **Overview** tab
3. In the overview editor, click the **Behavior** navigation tab.
4. In the Behavior page, expand the **Transaction** section and choose one of the following from the dropdown list:
   - **No Controller Transaction**: The called bounded task flow does not perform a transaction operation (begin, commit, or roll back a transaction).
   - **Always Use Existing Transaction**: When called, the bounded task flow participates in an existing transaction already in progress.
   - **Use Existing Transaction If Possible**: When called, the bounded task flow either participates in an existing transaction if one exists, or starts a new transaction upon entry of the bounded task flow if one does not exist.
   - **Always Begin New Transaction**: A new transaction starts when the bounded task flow is entered, regardless of whether or not a transaction is in progress. The new transaction completes when the bounded task flow exits.

   > **Note:**
   >
   > After choosing a transaction option, you may also need to configure the **Share data controls with calling task flow** option (`data-control-scope`) for the bounded task flow to determine whether there are any interactions between the options. For more information, see What You May Need to Know About Sharing Data Controls and Managing Transactions.

5. Optionally, select **Isolated** from the **Share data controls with calling task flow** checkbox (`data-control-scope`) so that data controls are not shared with the calling task flow if you chose **Always Begin New Transaction** in Step 4.

   The default behavior is to share data controls. For more information, see What You May Need to Know About Sharing Data Controls and Managing Transactions.

6. Optionally, select the **No save point on task flow entry** checkbox to prevent the creation of an ADF Model save point on task flow entry if you chose one of the following options in Step 4:

   - **Always Use Existing Transaction**
   - **Use Existing Transaction If Possible**

   An ADF Model save point is a saved snapshot of ADF Model state. Selecting the **No save point on task flow entry** checkbox means that overhead associated with a save point is not created for the transaction.

7. Select the task flow return activity in the called bounded task flow.

8. In the Properties window, expand the **Behavior** section.

9. If the called bounded task flow supports creation of a new transaction (bounded task flow specifies **Use Existing Transaction If Possible** or **Always Begin New Transaction** options), select one of the following in the **End Transaction** dropdown list:

   - **commit**: Select to commit the existing transaction to the database.
   - **rollback**: Select to roll back a new transaction to its initial state on task flow entry. This has the same effect as cancelling the transaction.

10. In the **Restore Save Point** dropdown list, select **true** if you want changes the user makes within the called bounded task flow to be discarded when the task flow exits. The save point that was created upon task flow entry will be restored.

What Happens When You Specify Transaction Options

Example 27-1 shows the metadata for transaction options on a called bounded task flow. The <new-transaction> element indicates that a new transaction always starts when the called bounded task flow is invoked. The <new-transaction> element is generated when you select **Always Begin New Transaction** value in the Transaction section's dropdown list.

Example 27-1 also shows the metadata for transaction options on the task flow return activity on the called task flow. The <commit/> element commits the existing transaction to the database. The <outcome> element specifies a literal outcome, for example, success, that is returned to the caller when the bounded task flow exits. The calling ADF task flow can define control flow rules based on this outcome. For more information about defining control flow upon return, see Using Task Flow Return Activities .

Example 27-1 Called Bounded Task Flow Metadata

```
<task-flow-definition id="trans-taskflow-definition">
   <default-activity>taskFlowReturn1</default-activity>
   <transaction>
     <new-transaction/>
   </transaction>
   <task-flow-return id="taskFlowReturn1">
     <outcome>
       <name>success</name>
       <commit/>
     </outcome>
   </task-flow-return>
 </task-flow-definition>
```

What You May Need to Know About Sharing Data Controls and Managing Transactions

Data controls cannot be shared across more than one transaction at the same time. If your task flow is involved in managing transactions, the value you select for the `data-control-scope` option may affect the `transaction` option settings for a bounded task flow. Table 27-1 describes how these options interact.

The ADF Model layer exposes the `DataControlFrame` interface to manage a transaction in which the data controls within the frame participate. The `DataControlFrame` interface exposes methods such as:

- `beginTransaction()`
- `commit()`
- `rollback()`

Similarly, ADF Controller allows a task flow to demarcate a transaction boundary, to begin a transaction at task flow entry, and to either commit or roll back the transaction on task flow exit. It does this by invoking methods exposed by ADF Model layer's `DataControlFrame` interface.

ADF Controller supports the transaction options listed in Table 27-1. The behavior of these transaction options depends on whether you select **Shared** or **Isolated** from the **Share data controls with calling task flow** list (XML element: `<data-control-scope>`) in the overview editor for a task flow.

Table 27-1 Transaction Settings Behavior

Transaction Setting	Share Data Control Scope	Isolate Data Control Scope
No Controller Transaction	The `DataControlFrame` is shared without the bounded task flow performing a transaction operation on it (begin, commit, or roll back a transaction). The existing transaction state of the data control frame remains and may be altered by other bounded task flows that share the same data control frame.	A new `DataControlFrame` is created without an open transaction.
Always Begin New Transaction XML element: `<new-transaction/>`	Begins a new transaction if one is not already open and throws an exception if one is already open.	Always begins a new transaction.
Always Use Existing Transaction XML element: `<requires-existing-transaction/>`	Throws an exception if the transaction is not already open.	Invalid. The checkbox cannot be selected.
Use Existing Transaction if Possible XML element: `<requires-transaction/>`	Begins a new transaction if one is not already open.	Always begins a new transaction.

A number of examples illustrate how bounded task flows interact when you configure different combinations of values for the data control scope and transaction properties described in Table 27-1:

- Isolate two bounded task flow transactions completely so that they can be committed or rolled back separately. For more information, see Creating Completely Separate Transactions.
- Guarantee that two bounded task flows share a transaction to save connection resources and stop the transaction if the called bounded task flow cannot join the transaction of the calling bounded task flow. For more information, see Guaranteeing a Called Bounded Task Joins an Existing Transaction.
- Call a bounded task flow that joins the transaction of the calling bounded task flow if one exists and, if not, creates its own transaction. For more information, see Using an Existing Transaction if Possible.

The above list describes common use cases rather than all possible configuration options.

Creating Completely Separate Transactions

Figure 27-5 shows a number of task flows that are configured to create separate transactions. Two bounded task flows are chained together in Figure 27-5 yet maintain completely separate transactions. This implementation may be appropriate where you have a call center agent taking an order from a customer and entering the order details in your application. At some point near the end of the order process, the customer realizes that the delivery address that they have provided is incorrect. To assist the call center agent, you designed your application so that it updates and commits the delivery address in a separate transaction to the transaction for the order details (order number, quantity, and so on). In the scenario where the customer wishes to modify the delivery address, the call center agent does not need to roll back changes to the order details in order to modify the delivery address because a separate transaction manages the order details.

In Figure 27-5, the unbounded task flow calls Bounded Task Flow #1. This results in the creation of Data Control Frame #1 because Bounded Task Flow #1 is configured as follows:

- A `data-control-scope` value of `isolated`
- **Always Begin New Transaction**

Bounded Task Flow #1 exposes attributes from two different **view objects** as data control fields (viewObject1.X and viewObject2.Y). X and Y are initially set to 10 and 20 respectively. The end user, while accessing Bounded Task Flow #1, updates X from 10 to 30 and leaves Y unchanged. This marks the transaction as changed or "dirty". That is, there are changes in the transaction that must be committed or rolled back.

Bounded Task Flow #1 calls Bounded Task Flow #2. The application creates Data Control Frame #2 because Bounded Task Flow #2 is configured as follows:

- A `data-control-scope` value of `isolated`
- **Always Begin New Transaction**

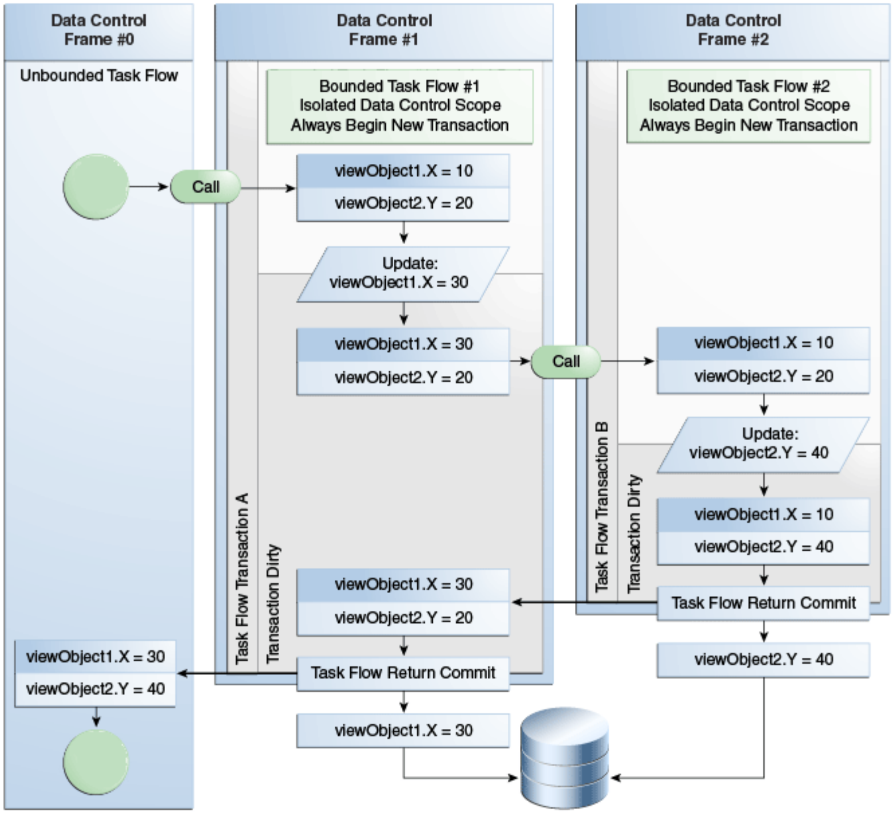
Note that the transaction of Data Control Frame #1 has yet to be committed and that Data Control Frame #2 has a separate database connection and a separate task flow transaction. As a result, the initial values of the data control fields X and Y in Bounded Task Flow #2 are 10 and 20. The end user, while accessing Bounded Task Flow #2, updates Y to 40 and leaves X unchanged. This marks the transaction for Bounded Task Flow #2 as dirty. Bounded Task Flow #2 next calls a task flow return activity that commits the change to Y to the database and returns control to Bounded Task Flow

#1. As Bounded Task Flow #2 did not modify the value of X, no change for this data control field occurs.

Because Bounded Task Flow #1 has a separate data control frame (Data Control Frame #1) and transaction, the values of the data control fields X and Y in Bounded Task Flow #1 remain at 30 and 20, their values when Bounded Task Flow #1 called Bounded Task Flow #2. Bounded Task Flow #1 now calls a task flow return activity (set to commit), saving the value of X to the database. The value of Y remains unchanged because Bounded Task #1's transaction did not modify the value of Y.

On returning to the unbounded task flow and refreshing the values of X and Y from the database, the updated values of 30 and 40 are returned.

Figure 27-5   Completely Separate Transactions



Guaranteeing a Called Bounded Task Joins an Existing Transaction

Figure 27-6 shows two bounded task flows in a configuration that guarantees that the second bounded task flow joins the transaction of the first bounded task flow that calls it. An example scenario where such an implementation may be appropriate is if your application creates invoices and, separately, the individual lines of the invoice. When creating an invoice line, it does not make sense that an invoice line is created without the parent invoice that is to contain the invoice line. If you were to build two separate task flows to create the invoice and invoice lines, the invoice line task flow needs to enforce that it wants to join a transaction of the calling task flow, in this case the invoice task flow, rather than creating a separate transaction.

Figure 27-6 illustrates how the above use case can be addressed by configuring an unbounded task flow that calls a bounded task flow (Bounded Task Flow #1). The unbounded task flow has its own data control frame (Data Control Frame #1). Bounded Task Flow #1 starts a transaction and also has a data control frame (Data Control Frame #0) because it is configured as follows:

- A `data-control-scope` value of `isolated`
- **Always Begin New Transaction**

Bounded Task Flow #1 exposes attributes from two different view objects as data control fields (viewObject1.X and viewObject2.Y). X and Y are initially set to 10 and 20 respectively. The end user, while accessing Bounded Task Flow #1, updates X from 10 to 30 and leaves Y unchanged. This marks Task Flow Transaction A associated with Data Control Frame #1 as changed or "dirty". That is, there are changes in the transaction that must be committed or rolled back.

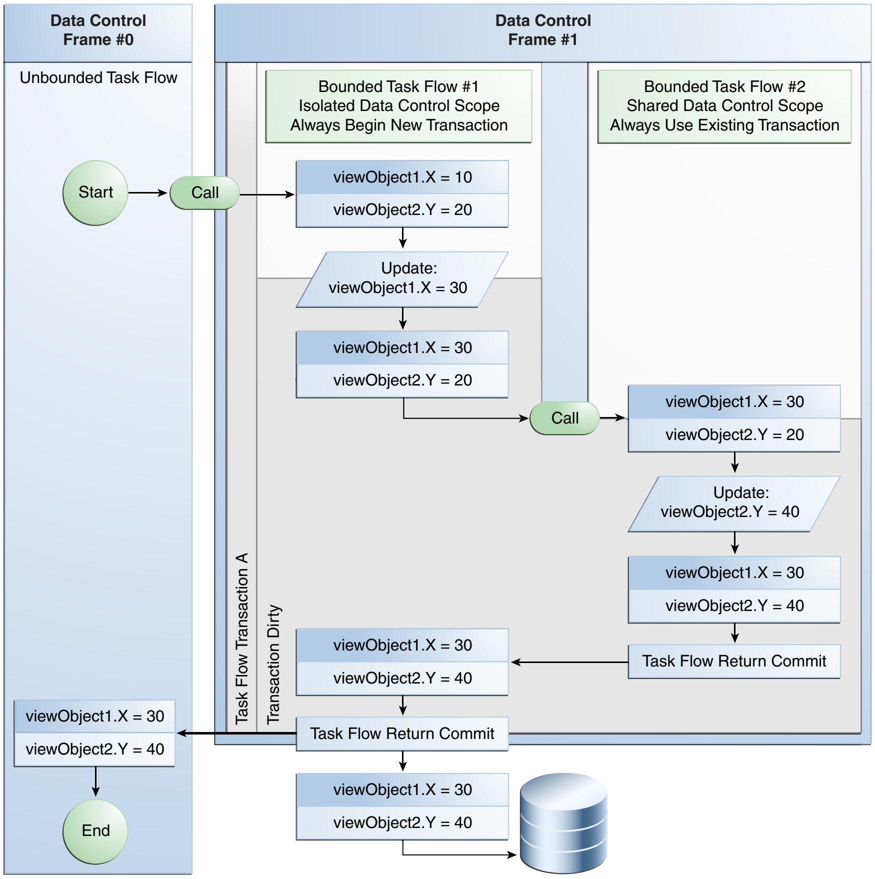
Bounded Task Flow #1 calls Bounded Task Flow #2. Bounded Task Flow #2 is configured as follows:

- A `data-control-scope` value of `shared`
- **Always Use Existing Transaction**

Because of this configuration, no new data control frames are created. Bounded Task Flow #2 uses the same data control frame (Data Control Frame #1) as Bounded Task Flow #1. The changes applied to X in Bounded Task Flow #1 are visible to Bounded Task Flow #2. Updates made to Y in Bounded Task Flow #2 are visible to Bounded Task Flow #1 when it receives control back from Bounded Task Flow #2 after it executes a task flow return activity that specifies a commit. This update to Y is visible in Bounded Task Flow #1 because of the shared data control scope, not because of the commit option that the task flow return activity of Bounded Task Flow #2 specifies.

Remember that the commit and rollback options of a task flow return activity are only valid for the bounded task flow that starts a transaction. That is why the commit of Bounded Task Flow #1´s task flow return activity finalizes changes in the database and not Bounded Task Flow #2.

Figure 27-6 Guaranteed Joined Transactions



Using an Existing Transaction if Possible

Guaranteeing a Called Bounded Task Joins an Existing Transaction, described how to configure a bounded task flow to call a second bounded task flow and guarantee that the called bounded task flow joins the transaction of the calling bounded task flow. The effect of changing the configuration of the calling bounded task flow to have:

- A `data-control-scope` value of `isolated`
- **No Controller Transaction**

while leaving the called bounded task flow unchanged is to stop execution of the bounded task flows and throw the following error message at runtime:

```
ADFC-00006: Existing transaction is required when calling task flow ''{0}''
```

The above scenario illustrates the benefit of configuring the called bounded task flow as follows:

- A `data-control-scope` value of `shared`
- **Use Existing Transaction if Possible**

This latter configuration means that the called bounded task flow creates a transaction if the calling bounded task flow has not created a transaction.

Another configuration option for a called bounded task that guarantees it creates its own transaction is to configure it as follows:

- A `data-control-scope` value of `isolated`
- **Use Existing Transaction if Possible**

This causes the called bounded task flow to create its own transaction regardless of the transaction options that you configure for the calling bounded task flow. The effect is the same as if you set the called bounded task flow to use Always Begin New Transaction.

What You May Need to Know About ADF Business Component Database Connections and Task Flow Transaction Options

ADF Controller shares the ADF Business Component application modules' database connections in an application that has more than one **root application module** (and each application module has a database connection) where task flow transaction options combine to share connections. That is, you configured a set of chained task flows in your application that have a have a `data-control-scope` value of `shared` in combination with the following transaction options:

- **Always Begin New Transaction**
- **Always Use Existing Transaction**
- **Use Existing Transaction if Possible**

When a user configures an ADF Controller to share a single transaction between task flows, by default Oracle ADF manages the application module instances of the various data controls as separate root application modules. These application modules share a single database connection and each application module is considered as an individual root application module that has its own unique configuration as defined during design time.

This implementation helps to reduce the number of database connections that each user creates and, as a result, makes the application more scalable. This is particularly useful when you add task flows and root application modules to your application through ADF Library JARs.

Users can override this default behavior and enforce nesting of the application modules under a single root application module by setting the `oracle.adfm.useSharedTransactionForFrame` property of the root application module to `false` in the `adf-config.xml` file. In such case, the nested application modules configuration settings will be based on the root application module configuration. Since it is difficult to determine the root application module, the configuration settings may be arbitrary. Hence, users must have a valid use case to set the `oracle.adfm.useSharedTransactionForFrame` property to `false`.

A set of chained task flows is where task flows call other task flows so that your application contains a chain with two or more task flows. For example, in Figure 27-2 the unbounded task flow UTF1 calls the bounded task flow BTF1 that in turn calls the bounded task flow BTF2, and so on.

> **Note:**
>
> Remember to configure your application's chained task flows to use a `data-control-scope` value of `isolated` if you do not want root application modules to share database connections.

However, you may want to configure different behavior where you want to have separate root application modules connect to different data sources within the same data model project. For example, you may want to show data from two different database systems or you need to connect to two different schemas in the same database.

To implement this last scenario (have the root application modules maintain their own database connections), you set the `transaction` property's value to No Controller Transaction (the default value) for the task flows in your application. This makes sure that each root application module manages its own database connection in isolation and that the ADF Controller does not manage the transaction behavior of the ADF Business Components.

Reentering Bounded Task Flows

ADF task flows allow you to control the reentry to the bounded task flow. By setting the value of the task-flow-reentry to reentry-allowed, reentry-not-allowed, or reentry-outcome-dependent you can either allow users, not allow users, or allow users based on the previous outcome of the bounded task flow to reenter the bounded task flow when users click on the back button of the navigation window.

To deal with cases in which the end user clicks the back button to navigate back into a bounded task flow that was already exited, you can specify `task-flow-reentry` options for the bounded task flow. These options specify whether a page in the bounded task flow can be reentered.

Upon reentry, bounded task flow input parameters are evaluated using the current state of the application, not the application state existing at the time of the original bounded task flow entry.

How to Set Reentry Behavior

You can set reentry behavior on a bounded task flow by specifying `task-flow-reentry` options.

Before you begin:

It may be helpful to have an understanding of what reentry options you can configure for a bounded task flow. For more information, see Reentering Bounded Task Flows.

You may also find it helpful to read about additional functionality that you can add using other task flow features. For more information, see Additional Functionality for Complex Task Flows.

To set reentry behavior:

1. In the Applications window, double-click the bounded task flow.

2. In the editor window, click the **Overview** tab.
3. In the overview editor, click the **Behavior** navigation tab.
4. In the Behavior page, choose one of the following from the **Task Flow Reentry** dropdown list:
   - **reentry-allowed**: Reentry is allowed on any view activity within the bounded task flow.
   - **reentry-not-allowed**: Reentry of the bounded task flow is not allowed. If you specify `reentry-not-allowed` on a bounded task flow, an end user can still click the browser back button and return to a page within the bounded task flow. However, if the user does anything on the page such as clicking a button, an exception (for example, `InvalidTaskFlowReentry`) is thrown indicating the bounded task flow was reentered improperly. The actual reentry condition is identified upon the submit of the reentered page.

     You can set up an exception handler to display the exception and route control flow in order to navigate to the default activity of the called bounded task flow. If the bounded task flow was not called from another bounded task flow, a normal web error is posted and handled as specified in the `web.xml` file.
   - **reentry-outcome-dependent**: Reentry of a bounded task flow using the browser back button is dependent on the outcome that was received when the same bounded task flow was previously exited via task flow return activities. If specified, any task flow return activities on the called bounded task flow must also specify either `reentry-allowed` or `reentry-not-allowed` to define outcome-dependent reentry behavior.

     If you choose this option, the user can navigate to a task flow using a back button based solely on how the user originally exited the task flow. For example, a task flow representing a shopping cart can be reentered if the user exited by canceling an order, but not if the user exited by completing the order.

How to Set Outcome-Dependent Options

You can set outcome-dependent options on bounded task flows that have specified the `reentry-outcome-dependent` option, as described in How to Set Reentry Behavior .

Before you begin:

It may be helpful to have an understanding of what reentry options you can configure for a bounded task flow. For more information, see Reentering Bounded Task Flows.

You may also find it helpful to read about additional functionality that you can add using other task flow features. For more information, see Additional Functionality for Complex Task Flows.

To set outcome-dependent options:

1. In the Applications window, double-click the bounded task flow.
2. In the task flow diagram for the bounded task flow, select the task flow return activity.

   For information about adding a task flow return activity, see Using Task Flow Return Activities .
3. In the Properties window, expand the **General** section and enter the name of a literal outcome, for example, `success` or `failure` in the **Name** field.

4. Expand the **Behavior** section and choose one of the following options in the **Reentry** dropdown list:
   - **reentry-allowed**: Reentry is allowed on any view activity within the bounded task flow.
   - **reentry-not-allowed**: Reentry of the bounded task flow is not allowed. If you specify `reentry-not-allowed` on a bounded task flow, an end user can still click the browser back button and return to a page within the bounded task flow. However, if the user does anything on the page such as clicking a button, an exception (for example, `InvalidTaskFlowReentry`) is thrown indicating the bounded task flow was reentered improperly. The actual reentry condition is identified upon the submit of the reentered page.

     You can set up an exception handler to display the exception and route control flow in order to navigate to the default activity of the called bounded task flow. If the bounded task flow was not called from another bounded task flow, a normal web error is posted and handled as specified in the `web.xml` file.

What Happens When You Configure a Bounded Task Flow for Reentry

JDeveloper updates the task flow's metadata with the configuration options that you specify for reentry. The following example shows a task flow that has been configured to allow reentry based on a specific outcome.

```
<task-flow-definition id="task-flow-definition1">
    <default-activity>view1</default-activity>
    <task-flow-reentry>
      <reentry-outcome-dependent/>
    </task-flow-reentry>
    <view id="view1">
      <page>/page1.jsf</page>
    </view>
    <task-flow-return id="taskFlowReturn1">
      <outcome>
        <name>taskFlowReturn1</name>
        <reentry-allowed/>
      </outcome>
    </task-flow-return>
</task-flow-definition>
```

At runtime, the following events occur to a task flow that is configured to allow reentry:

- The task flow is reentered if the user invokes a component that, in turn, invokes a HTTP `POST` or `GET` method.
- The task flow's managed bean values in pageFlow scope are not restored to the values they had when the task flow exited.
- If the reentered task flow defines input parameters, as described in Using Parameters in Task Flows, their values are recreated using the current state of the parent task flow at the time that the task flow is reentered.

Handling Exceptions in Task Flows

Exceptions to the normal flow of application activities can occur during ADF application runtime. ADF task flows allow you to handle such exceptions. You can designate one activity to handle exceptions in a bounded or unbounded task flow as an exception handler.

During execution of a task flow, exceptions can occur that may require some kind of exception handling, for example:

- A method call activity throws an exception.
- A custom method you have written as a task flow initializer or finalizer throws an exception.
- A user is not authorized to execute the activity.

To handle exceptions thrown from an activity or caused by some other type of ADF Controller error, you can designate one activity in a bounded or unbounded task flow as an exception handler.

When a task flow throws an exception, control flow passes to the designated exception handling activity. For example, the exception handling activity might be a view activity that displays an error message. Alternatively, the activity might be a router activity that passes control flow to a method based on an EL expression that evaluates the type of exception. For example:

```
#{controllerContext.currentViewPort.exceptionData.class.name ==
'oracle.adf.controller.ControllerException'}
```

After control flow passes to the exception handling activity, flow from the exception handling activity uses standard control flow rules. For example, you designate a router activity as the exception handling activity. At runtime, the task flow passes control to the exception handling activity (in this example, a router activity) in response to an exception. In addition to designating the router activity as an exception handler, you can define task flow control cases that the router invokes based on the type of exception that it has to handle. This allows you to manage your end user's application session gracefully when an exception occurs. See About Control Flows.

You can optionally specify an exception handler for both bounded and unbounded task flows. Each task flow can have only a single exception handler. However, a task flow called from another task flow can have a different exception handler from that of the caller. In addition, a region on a page can have a different exception handler from that of the task flow containing the page. The exception handler activity can be any supported activity type, for example, a view or router activity.

If a bounded task flow does not have a designated exception handler activity, control passes to an exception handler activity in a calling bounded task flow, if there is a calling task flow and if it contains an exception handler activity. The exception is propagated up the task flow stack until either an exception handler activity or the top-level unbounded task flow is reached. If no exception handler is found, the exception is propagated to the web container.

If a bounded task flow does have a designated exception handler activity, make sure the exception handler activity leaves the application in a valid state after it handles an exception. One way to do this is to redirect to a view activity in the same task flow after the exception handler activity.

Other modules, such as ADF Model, also provide exception handling capabilities. In certain scenarios this can determine the way that your application handles exceptions. For example, a databound method activity is a method activity that has a page definition and an EL expression with the following format:

```
#{bindings.somebindingname.execute}
```

where *somebindingname* is a method binding defined in the page definition.

An exception thrown by any type of binding is caught by ADF Model which calls the `reportException()` method and stores the exception in the binding container. Later, when the page renders, the error displays in the page.

When a method activity invokes a method binding, there is no page to display an error that the exception raises because the exception occurs during navigation between two pages. To make the application aware of an error, Oracle ADF rethrows the exception so that it is caught by ADF Controller's exception handler mechanism that navigates to an exception handler activity if one exists.

Keep this in mind, particularly if you decide to override methods, such as the `reportException()` method, described in Customizing Error Handling, because in that scenario both ADF Controller and ADF Model exception handlers will be called.

How to Designate an Activity as an Exception Handler

You can designate an exception handler activity for a bounded task flow running as an ADF region. If an exception occurs in the bounded task flow and it is not handled by the task flow's exception handler, the exception is not propagated up the task flow stack of the parent page. Instead, it becomes an unhandled exception.

Before you begin:

It may be helpful to have an understanding of what exception handling options you can configure for a task flow. For more information, see Handling Exceptions in Task Flows.

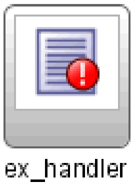
You may also find it helpful to read about additional functionality that you can add using other task flow features. For more information, see Additional Functionality for Complex Task Flows.

To designate an activity as an exception handler for a task flow:

1. In the Applications window, double-click the task flow where you want to designate an activity as an exception handler.
2. In the task flow diagram, right-click the activity and choose **Mark Activity** > **Exception Handler**.

   A red exclamation point is superimposed on the activity in the task flow to indicate that it is an exception handler. Figure 27-7 shows an example.

   Figure 27-7 Example of an Activity Designated as an Exception Handler

3. To unmark the activity, right-click the activity in the task flow diagram, and choose **Unmark Activity** > **Exception Handler**.

   If you mark an activity as an exception handler in a task flow that already has a designated exception handler, the old handler is unmarked.

What Happens When You Designate an Activity as an Exception Handler

After you designate an activity to be the exception handling activity for a task flow, JDeveloper updates the task flow metadata with an `<exception-handler>` element that specifies the ID of the activity, as shown in Example 27-2.

Example 27-2 <exception-handler> element

```
<exception-handler>activityID</exception-handler>
```

How to Designate Custom Code as an Exception Handler

Rather than designate a task flow activity as the activity to invoke, you can write custom code to invoke when a task flow throws an exception. This requires you to:

- Write a Java class that extends the class `ExceptionHandler` from the following package:

  `oracle.adf.view.rich.context.ExceptionHandler`
- Register the Java class that you write as a service in the `.adf\META-INF` directory of your Fusion web application

The following example shows custom code that checks if an exception thrown by a task flow corresponds to a particular type of error message (`ADF_FACES-30108`). If it does, the custom code redirects the task flow to the `faces/SessionExpired.jsf` page.

```
package oracle.summit.frmwkext;

import javax.faces.context.ExternalContext;
import javax.faces.context.FacesContext;
import javax.faces.event.PhaseId;
import oracle.adf.view.rich.context.ExceptionHandler;

public class CustomExceptionHandler extends ExceptionHandler {

    public CustomExceptionHandler() {
        super();
    }

    public void handleException(FacesContext facesContext, Throwable throwable,
                                PhaseId phaseId) throws Throwable {

        String error_message;
        error_message = throwable.getMessage();

        if (error_message != null &&
            error_message.indexOf("ADF_FACES-30108") > -1) {
            ExternalContext ectx = facesContext.getExternalContext();
            ectx.redirect("faces/SessionExpired.jsf");
        }

        else {
            //Code to execute if the if condition is not met
            throw Throwable
        }
    }
}
```

For information about the APIs that you can use to write custom code, see the following reference documents:

- *Java API Reference for Oracle ADF Controller*
- *Java API Reference for Oracle ADF Faces*

Before you begin:

It may be helpful to have an understanding of what exception handling options you can configure for a task flow. For more information, see Handling Exceptions in Task Flows.

You may also find it helpful to read about additional functionality that you can add using other task flow features. For more information, see Additional Functionality for Complex Task Flows.

To designate custom code as an exception handler:

1. In the `.adf\META-INF` directory of your application, create a directory named **services** so that you have the following directory path:

   ```
   application_root\.adf\META-INF\services
   ```

   where `application_root` refers to the root directory of application.
2. Create a text file named **oracle.adf.view.rich.context.ExceptionHandler** in the **services** folder.
3. Write the package name and class name of the custom code that you wrote to handle exceptions in the text file named **oracle.adf.view.rich.context.ExceptionHandler**.

   For example, write code similar to the following:

   ```
   oracle.summit.frmwkext.CustomExceptionHandler
   ```
4. Save and close the text file.

What Happens When You Designate Custom Code as an Exception Handler

At runtime, the task flow passes control to the custom code that you specified if the task flow throws an exception.

What You May Need to Know About Handling Exceptions During Transactions

Designate an exception handling activity for a bounded task flow that is enabled to run as a transaction. A Fusion web application attempts to commit a transaction if you set `commit` as the value for a task flow return activity's End Transaction property on a bounded task flow that runs as a transaction. If an exception occurs when the Fusion web application attempts to commit a transaction, the exception handling activity receives control and provides the end user with an opportunity to correct the exception. You can use the exception handling activity (for example, a view activity) to display a warning message to an end user with information about how to correct the exception and how to recommit the transaction. For information about enabling a bounded task flow as a transaction and setting commit as a value for the End Transaction property, see How to Enable Transactions in a Bounded Task Flow .

What You May Need to Know About Handling Validation Errors

For validation errors on a JSF page, you can rely on standard JSF to attach validator error messages to specific components on a page or to the whole page. A component-level validator typically attaches an error message inline with the specific UI component. There is no need to pass control to an exception handler activity.

In addition, your application should define validation logic on data controls that are executed during the Validate Model Updates phase of the JSF lifecycle. In this way, data errors are found as they are submitted to the server without waiting until attempting the final commit.

Validations done during the Validate Model Updates phase typically do not have direct access to the UI components because the intention is to validate the model after the model has been updated. These validations are often things like checking to see whether dependent fields are synchronized. In these cases, the error message is usually attached to the whole page, which this logic can access.

You should attach errors detected during the Validate Model Updates phase to the JSF page, and call `FacesContext.renderResponse()`. This signals that following this phase, the current (submitting) page should be rendered showing the attached error messages. There is no need to pass control to an exception handler activity.

For more information, see Implementing Validation and Business Rules Programmatically.

Configuring Your Application to Use Save Points

To use save points with ADF task flows, you need to define a value for the <savepoint-datasource> element in the adf-config.xml file. This element is used to specify the JNDI name of the data source containing the database table of the save points. To create this database table, you need to run the SQL script adfc_create_save_point_table.sql. To delete the expired save points, you need to run the SQL script adfc_cleanup_save_point_table.sql.

Before you can add save points to a task flow, as described in Using Save Points in Task Flows, and configure related functionality, you need to make sure that the Fusion web application allows save points. To do this, you define a value for the `<savepoint-datasource>` element in the `adf-config.xml` file to specify the JNDI name for the data source that contains the save points' database table. You may also need to run a SQL script (`adfc_create_save_point_table.sql`), as described in What You May Need to Know About the Database Table for Save Points, to create the database table that stores save points. Once your Fusion web application starts using save points, you can use another SQL script (`adfc_cleanup_save_point_table.sql`) to delete expired save points.

How to Configure Your Fusion Web Application to Use Save Points

You define a value for the `<savepoint-datasource>` element in your application's `adf-config.xml` file to specify the JNDI name for the data source that contains the save points' database table. Optionally, you can also specify an expiration time for save points.

Before you begin:

It may be helpful to have an understanding of what save point options you can configure for a task flow. For more information, see Configuring Your Application to Use Save Points.

You may also find it helpful to read about additional functionality that you can add using other task flow features. For more information, see Additional Functionality for Complex Task Flows.

To configure your Fusion web application to allow save points:

1. In the Application Resources panel, expand the **Descriptors** and **ADF META-INF** nodes, and then double-click **adf-config.xml**.
2. In the overview editor, click the **Controller** navigation tab and then expand the **Savepoints** section to write a value for the Data Source property that specifies the JNDI name for the data source that contains the database table for save points.

   For example, write the following:

   `java:comp/env/jdbc/`*`Connection1`*`DS`

   where `Connection1` is the JDeveloper connection name.
3. Optionally, write a value in seconds for the Expiration property to specify the time between when a task flow creates a save point and when the save point manager removes it. The default value is 86400 seconds.

   For more information, see What You May Need to Know About the Time-to-Live Period for a Save Point .
4. Save the `adf-config.xml` file.

What Happens When You Configure a Fusion Web Application to Use Save Points

JDeveloper generates an entry, similar to the following example, in the `adf-config.xml` file to specify the JNDI name for the data source that contains the save points' database table.

```
<adf-controller-config xmlns="http://xmlns.oracle.com/adf/controller/config">
    ...
    <savepoint-datasource>
      java:comp/env/jdbc/Connection1DS
    </savepoint-datasource>
    <savepoint-expiration>
      86399
    </savepoint-expiration>
  </adf-controller-config>
```

For more information about the `adf-config.xml` file, see adf-config.xml.

What You May Need to Know About the Database Table for Save Points

A database table named `ORADFCSAVPT` stores save points. If this database table does not exist, it is created the first time that a save point is created if your Fusion web application has the necessary permissions to create a database table. If your Fusion web application does not have the necessary permissions, you or an administrator with the necessary permissions can use SQL scripts to create and maintain the `ORADFCSAVPT` database table. These SQL scripts are:

- `adfc_cleanup_save_point_table.sql`

  Each save point in the `ORADFCSAVPT` database table has an expiration date. Use this script to delete save points that have passed their expiration date.
- `adfc_create_save_point_table.sql`

  Use this script to create the `ORADFCSAVPT` database table that stores save points.

You can find these SQL scripts in the following directory of your JDeveloper installation:

```
jdev_install\oracle_common\common\sql
```

Using Save Points in Task Flows

The ADF task flow allows you to save the current state of the application and restore the state for a later use by using save point. To invoke these save points you need to add a method call activity in the bounded task flow.

You can configure a task flow to capture the state of a Fusion web application at a particular instance creating what is called a **save point**. This allows you to save application state if, for example, a user leaves a page without finalizing it. The application state can be restored at a later point.

Table 27-2 describes what information a save point captures.

Table 27-2 Saved Application State Information

Saved State Information	Description
User Interface State	UI state of the current page, including selected tabs, selected checkboxes, selected table rows, and table column sort order. This state assumes the end user cannot select the browser back button on save point restore.

Table 27-2 (Cont.) Saved Application State Information

Saved State Information	Description
Managed Beans	State information saved in several possible memory scopes, including session and page flow scope. The managed beans must be serializable in order to be saved. If you have page flow scope beans that are not serializable and you attempt to create a save point, a runtime exception occurs. Request scope is not supported since its lifespan is a single HTTP request and its lifespan cannot be used to store cross request application state. Save points will not save and restore application-scoped managed beans since they are not passivated in failover scenarios. Therefore, the application is always responsible for making sure that all required application-scoped state is available. Potential naming conflicts for managed beans already existing within the session scope at restore time will not occur because multiple managed beans using the same name should not be implemented within an application.
Navigation State	Task flow call stack, which ADF Controller maintains as one task flow calls another at runtime. The task flow call stack tracks where the end user is in the application and the navigation path for getting there. The task flow stack also identifies the starting point of any persisted data transactions originated for the end user.
ADF Model State	Fusion web applications use ADF Model to represent the persisted data model and business logic service providers. The ADF Model holds any data model updates made from when the current bounded task flow begins. The model layer determines any limits on the saved state lifetime. For more information, see SeeUsing State Management in a Fusion Web Application.

You add a method call activity to a bounded task flow that invokes a `createSavePoint` method to create save points. Later, you use a save point restore activity to restore application state and data associated with the created save points.

The same save point can be used if a user repeatedly performs a save for later on a task flow instance that executes in one session within the same browser window. The new save point overwrites the existing save point when a user performs a save for later following navigation from page to page in a task flow. For information about restoring a save point, see How to Restore a Save Point.

You can specify the `createSavePoint` method exposed by the `currentViewPort` node of ADF Controller Objects in the Expression Builder. Alternatively, you can write a custom method that updates the save point with the values of attributes you specify in your custom method, as illustrated in the following example.

```
package viewController;

import java.io.Serializable;

import oracle.adf.controller.ControllerContext;
import oracle.adf.controller.savepoint.SavePointManager;

public class SaveForLater implements Serializable {
    public SaveForLater() {
        super();
    }

    public String saveTaskFlow() {
        ControllerContext cc = ControllerContext.getInstance();
        if (cc != null) {
            SavePointManager mgr = cc.getSavePointManager();
        if (mgr != null) {
            String id = mgr.createSavePoint();
            System.out.println("Save point is being set " + id);
        ...
```

The SavePointListener interface exposes methods that notify clients when save point events occur. The following package contains the SavePointListener interface:

oracle.adf.controller.savepoint

> **Note:**
>
> All save points created inside a bounded task flow are deleted when the bounded task flow exits.

How to Add a Save Point to a Task Flow

You drag and drop a method call activity to the task flow diagram and configure it to invoke the createSavePoint method or to invoke a custom method if you created one.

Before you begin:

It may be helpful to have an understanding of what save point options you can configure for a task flow. For more information, see Using Save Points in Task Flows.

You may also find it helpful to read about additional functionality that you can add using other task flow features. For more information, see Additional Functionality for Complex Task Flows.

You will need to complete this task:

Confirm that your Fusion web application leaves the value of the jbo.locking.mode property set to the default value optimistic. This is required if you use a save point in a Fusion web application. The value pessimistic causes an old session to lock until the session has timed out. In pessimistic mode, if you run an application and change data without committing changes to the database, you may get an error when you create a save point and try to restore it at a later point. For information about the jbo.locking.mode property, see How to Confirm That Fusion Web Applications Use Optimistic Locking.

To add a save point to a task flow:

1. In the Applications window, double-click the bounded task flow where you want to add a save point.
2. In the ADF Task Flow page of the Components window, from the Component panel, in the Activities group, drag and drop a **Method Call** onto the diagram.
3. In the Properties window, expand the **General** node and write an EL expression in the **Method** field to specify the save point method that the method call activity invokes. If needed, select **Method Expression Builder** in the dropdown menu that you invoke from the icon that appears when you hover over the property field.

   If you use the Expression Builder to specify the createSavePoint method exposed by the currentViewPort node of ADF Controller Objects, the resulting EL expression is similar to the following:

```
#{controllerContext.currentViewPort.createSavePoint}
```

4. Use a control flow to connect the method call activity with other activities in the bounded task flow.

   For more information, see How to Add a Control Flow Rule to a Task Flow.

5. Optionally, configure save point options in the Fusion web application's `adf-config.xml` file to determine, for example, if implicit save points can be created for the application.

   For more information, see How to Enable Implicit Save Points.

What Happens When You Add Save Points to a Task Flow

JDeveloper generates entries similar to those shown in the following example in the task flow's source file when you configure a method call activity to invoke the `createSavePoint` method.

```
<method-call id="methodCall1">
      <method id="__3">#{controllerContext.currentViewPort.createSavePoint}</
method>
</method-call>
```

How to Restore a Save Point

Use the save point restore activity to restore a previously persisted save point for an application. The save point restore activity uses the save point that was originally created by invoking the `createSavePoint` method to identify the save point to restore.

You can obtain a list of the current persisted save points with `createSavePoint`. However, ADF Controller does not determine which save points to restore. A user must select the save point from a list or the application developer must select it programmatically. The savepoint ID is then passed to a save point restore activity to perform the restore.

Before you begin:

It may be helpful to have an understanding of what save point options you can configure for a task flow. For more information, see Using Save Points in Task Flows.

You may also find it helpful to read about additional functionality that you can add using other task flow features. For more information, see Additional Functionality for Complex Task Flows.

To add a save point restore activity to a bounded or unbounded task flow:

1. In the Applications window, double-click the task flow where you want to add the save point restore activity.
2. In the ADF Task Flow page of the Components window, from the Components panel, in the Activities group, drag and drop a **Save Point Restore** onto the diagram.
3. In the Properties window, expand the **General** section and write an EL expression in the **Save Point ID** field that, when evaluated, retrieves the save point that was originally created when the `createSavePoint` method was invoked.

   If you use the Expression Builder to specify the `getSavePoint` method of ADF Controller Objects, the resulting EL expression is similar to the following:

```
#{SessionScope.myBean.savepointID}
```

What Happens When You Restore a Save Point

JDeveloper generates entries similar to those in the following example in the task flow's source file when you add a save point restore activity that gets a save point ID.

```
<save-point-restore id="savePointRestore1">
      <save-point-id id="__4">#{sessionScope.myBean.savepointID}</save-point-id>
</save-point-restore>
```

How to Use the Save Point Restore Finalizer

When using the save point restore activity, you may need to invoke application-specific logic as part of restoring the application state. You can write an EL expression for the Save Point Restore Finalizer property of a bounded task flow that specifies a finalizer method. The bounded task flow invokes the specified method after the task flow's state has been restored. It performs any necessary logic to make sure that the application's state is correct before proceeding with the restore.

Before you begin:

It may be helpful to have an understanding of what save point options you can configure for a task flow. For more information, see Using Save Points in Task Flows.

You may also find it helpful to read about additional functionality that you can add using other task flow features. For more information, see Additional Functionality for Complex Task Flows.

To use the save point restore finalizer:

1. In the Applications window, double-click the bounded task flow that you want to use a save point restore finalizer.
2. In the Structure window, right-click the node for the bounded task flow (`task-flow-definition`) and choose **Go to Properties**.
3. In the Properties window, expand the **General** section, and select **Expression Builder** in the dropdown list that you invoke from the icon that appears when you hover over the **Save Point Restore Finalizer** property field.
4. Write an EL expression that specifies the finalizer method to invoke.

What Happens When a Task Flow Invokes a Save Point Restore Finalizer

JDeveloper generates entries similar to those in the following example in the task flow's source file when you write an EL expression for the Save Point Restore Finalizer property.

```
<task-flow-definition id="task-flow-definition1">
    <save-point-restore-finalizer id="__2">#{sessionScope.MyBean.invokeFinalizer}
      </save-point-restore-finalizer>
  </task-flow-definition>
```

How to Enable Implicit Save Points

A save point in a task flow can be categorized as *implicit* or *explicit*. An explicit save point requires an end user action before a bounded or unbounded task flow creates a save point. For example, an end user clicks a button that invokes a method call activity that, in turn, creates a save point.

An implicit save point can only originate from a bounded task flow. It includes everything from when the originating task flow creates a save point. It occurs when data is saved automatically because:

- A session times out due to end user inactivity.
- An end user logs out without saving the data.
- An end user closes the only browser window, thus logging out of the application.
- An end user navigates away from the current application using control flow rules (for example, uses a `link` component to go to an external URL) and having unsaved data.

Enabling implicit save points requires you to add an element to your Fusion web application's `adf-config.xml` file and to make the bounded task flow critical.

You configure the `adf-config.xml` file in your application and the bounded task flow(s) for which you want to create implicit save points. Enabling implicit save points involves a performance cost because your Fusion web application has to do extra work that it would otherwise not do. This is why you have to explicitly enable implicit save points in your application and specify the task flows to which it applies.

Before you begin:

It may be helpful to have an understanding of what save point options you can configure for a task flow. For more information, see Using Save Points in Task Flows.

You may also find it helpful to read about additional functionality that you can add using other task flow features. For more information, see Additional Functionality for Complex Task Flows.

To enable implicit save points:

1. In the Application Resources panel, expand the **Descriptors** and **ADF META-INF** nodes, and then double-click **adf-config.xml**.
2. In the overview editor, click the **Controller** navigation tab and then expand the **Savepoints** section and select the **Enable Implicit Savepoints** checkbox.

   JDeveloper generates the following entry in the `adf-config.xml` file:

   ```
   <adf-controller-config xmlns="http://xmlns.oracle.com/adf/controller/config">
       ...
       <enable-implicit-savepoints>true</enable-implicit-savepoints>
   </adf-controller-config>
   ```

   For more information about the `adf-config.xml` file, see adfc-config.xml.
3. In the Applications window, double-click the bounded task flow.
4. In the editor window, click the **Overview** tab.

5. In the overview editor, click the **Behavior** navigation tab and select the **Critical** checkbox.

What You May Need to Know About Enabling Implicit Save Points

If multiple windows are open when the implicit save point is created, a different save point is created for each browser window. This includes everything from the root view port of the browser window on down. You can write an EL expression for the Method property of a method call activity to retrieve the list of implicit save points using the `savePointManager` node under ADF Controller Objects. The resulting EL expression is similar to the following:

```
ControllerContext.savePointManager.listSavePointIds
```

Implicit save points are generated only if a critical task flow is present in any of the page flow stacks for any view port under the current root view port. An implicit save point is not generated if the request is for an ADF Controller resource, such as:

- Task flow call activity
- Task flow return activity
- Save point restore activity
- A dialog

Implicit save points are deleted when the task flow at the bottom of the stack completes or a new implicit save point is generated, whichever comes earlier.

What You May Need to Know About the Time-to-Live Period for a Save Point

An application-level property (`savepoint-expiration`) that is defined in the `adf-config.xml` file determines the period between when a task flow creates a save point and when the save point manager removes it (time-to-live period). The default value is 86400 seconds (24 hours).

You can change the time-to-live period for individual save points by calling the `setSavePointTimeToLive` method on an instance of `SavePointManger` from the following package:

```
oracle.adf.controller.savepoint
```

An instance of `SavePointManager` can be obtained as follows:

```
SavePointManager mgr = ControllerContext.getInstance().getSavePointManager();
```

The following example shows the syntax for the `setSavePointTimeToLive` method.

```
    public void setSavePointTimeToLive(long timeInSeconds) {
       }
```

If you supply a value for the `setSavePointTimeToLive` method argument (`timeInSeconds`) equal to or less than zero, the default value is used (`86400`).

The `SavePointManger` defines methods that help you manage save points. For example, it defines `getSavePoint` and `removeSavePoint` methods that can retrieve and remove save points. Note that the `removeSavePoint` method does not get called automatically when a save point expires. You must explicitly call the `removeSavePoint` method to remove save points (including expired save points) from the `ORADFCSAVPT` database table. Alternatively, Oracle ADF provides a SQL script (`adfc_cleanup_save_point_table.sql`) that removes expired save points. For more information, see What You May Need to Know About the Database Table for Save Points.

Consider calling the `setSavePointTimeToLive` method at the same time that you call the method to create save points. For more information about the `SavePointManger`, see the *Java API Reference for Oracle ADF Controller*.

Minimizing the Number of Active Root View Ports in an Application

The ADF task flow allows you to limit the number of active root view ports in an application by setting a value for the Maximum Root View Ports property (<max-root-view-ports>) in the adf-config.xml of the application. Setting the limit may prevent the denial-of-service attacks. Setting the value to -1 allows unlimited view ports; however, the default value is 20.

You can minimize the number of active root view ports in an application by specifying a value for the Maximum Root View Ports property (`<max-root-view-ports>`) in the application's `adf-config.xml` file. This limits the memory footprint of the application and can help to prevent denial-of-service attacks.

The default value is `20`. The lowest permissible value is `5`. To prevent accidental misuse, the application ignores any value below `5` and enforces a value of `5`.

Specify a value of `-1` if you want to disable this feature and allow an unlimited number of active root view ports in the application.

How to Minimize the Number of Active Root View Ports

You specify a value for the `<max-root-view-ports>` property in the application's `adf-config.xml` file.

Before you begin:

It may be helpful to have an understanding of what `<max-root-view-ports>` options you can configure. For more information, see Using Save Points in Task Flows.

You may also find it helpful to read about additional functionality that you can add using other task flow features. For more information, see Additional Functionality for Complex Task Flows.

To minimize the number of active root view ports:

1. With the Fusion web application open in JDeveloper, open the Application Resources pane in the Applications window.
2. Expand **Descriptors**, then expand **ADF META-INF**.
3. Right-click `adf-config.xml` and choose **Open** from the context menu.
4. On the Controller page of the overview editor, write a value for the Maximum Root View Ports property to specify the maximum number of active root view ports for the application.

5. Save the `adf-config.xml` file.

What Happens When You Minimize the Number of Active Root View Ports

JDeveloper generates an entry, similar to that illustrated in the following example, in the `adf-config.xml` file to specify the maximum number of active root view ports for the application.

```
<?xml version="1.0" encoding="UTF-8" ?>
<adf-config xmlns="http://xmlns.oracle.com/adf/config"
            xmlns:config="http://xmlns.oracle.com/bc4j/configuration"
            xmlns:adf="http://xmlns.oracle.com/adf/config/properties">
    ...
  <adf-controller-config xmlns="http://xmlns.oracle.com/adf/controller/config">
    <max-root-view-ports>
      19
    </max-root-view-ports>
  </adf-controller-config>
</adf-config>
```

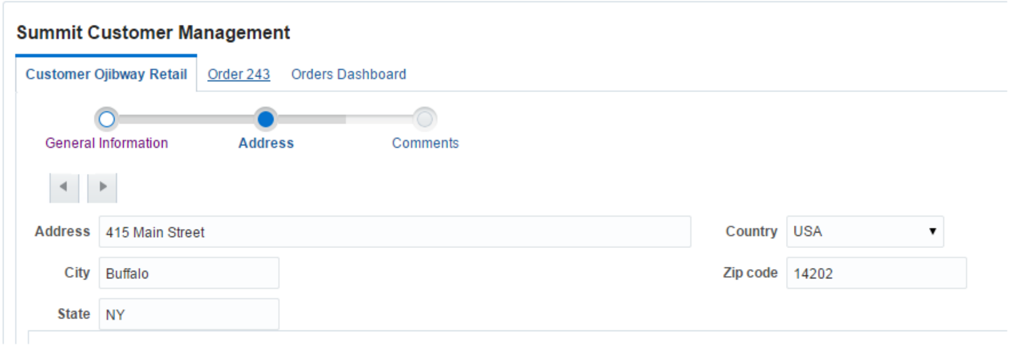
At runtime, once the specified root view port limit is reached, any request to create a new root view port automatically expires the least-recently-used root view port first. Attempts to subsequently use the expired root view port result in a `ViewExpiredException`.

Using Train Components in Bounded Task Flows

By integrating a train with an ADF task flow, you can allow users to go through a multistep process. The train and trainButtonBar components allow you to create a user interface that can render the train functionality in the task flow view activities. This feature allows you to branch out using router activities and control flow cases in the task flow.

You can create a task flow as a train. This allows you to create a user interface where you navigate end users through a multistep process. ADF Faces provides the user interface components that render the train functionality in the task flow view activities that appears to the end user. These components are the `train` and `trainButtonBar` components. Figure 27-8shows a runtime view of the `edit-customer-task-flow-definition.xml` task flow from the Summit ADF task flow sample application. This task flow uses the `train` and `trainButtonBar` components to navigate an end user through the task of editing a customer's data.

Figure 27-8 Train and Train Button Bar Components in a Task Flow



Address 415 Main Street
Country USA
City Buffalo
Zip code 14202
State NY

The `train` component in Figure 27-8 renders three train stops. Each stop corresponds to a page fragment in the `edit-customer-task-flow-definition.xml` task flow where the end user can enter and review information to complete the task of editing a customer's data. Figure 27-8 shows the Address train stop where the end user enters an address for postage. The other stops are:

- General Information
- Comments

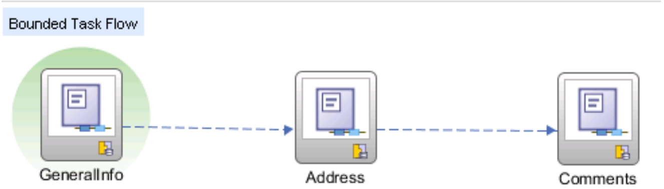
The Train Button Bar component, also shown in Figure 27-8, is optional and provides additional controls to navigate between the train stops. This component can be used in conjunction with the train component to provide multiple ways to navigate through train stops.

Bounded task flows and task flow templates can make use of these train components if you select the **Create Train** checkbox on the dialogs provided by JDeveloper to create a bounded task flow or a task flow template. A bounded task flow or a task flow template can render one train only. If you want to use multiple trains, create a separate bounded task flow or task flow template for each train.

Figure 27-9 displays an extract from the design-time view of the task flow where you can see the view activities that render at runtime as train stops in Figure 27-8.

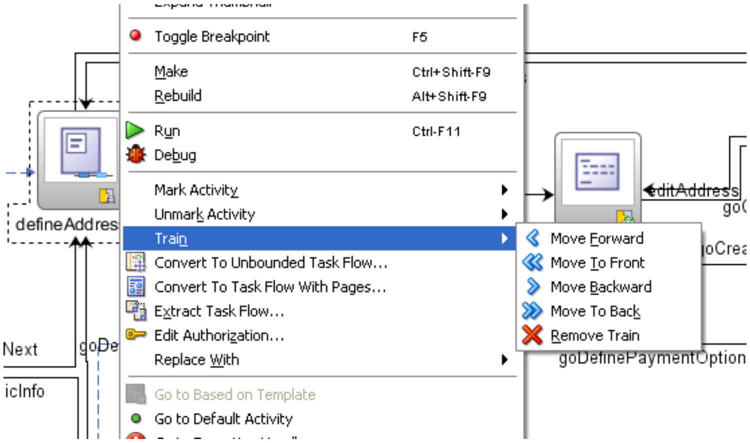
The dotted lines connecting the view activities indicates the sequence in which the end user navigates between the view activities when the view activities render as train stops.

Figure 27-9 Task Flow Configured to Render as a Train



JDeveloper displays a context menu with options to change the position of the activity when your right-click a view activity or task flow call activity that is configured as a train stop, as illustrated in Figure 27-10.

Figure 27-10   Context Menu to Edit a Train Stop



Configure a task flow call activity as a train stop when you want to group a number of activities as a train stop or call a child train stop. For example, there are cases when other task flow activities, such as router and method call activities, should be considered part of a train stop along with the corresponding view activity. A method call activity might need to precede a view activity for initialization purposes. When grouped this way, the activities can be performed as a set each time the train stop is visited. See Grouping Task Flow Activities to Execute Between Train Stops .

Branching using router activities and control flow cases is supported on the task flow diagram containing the train, as well as in child bounded task flows called from the train.

Creating a Task Flow as a Train

You need to configure a bounded task flow to use train components before you can implement the type of functionality discussed in Using Train Components in Bounded Task Flows. Use one of the methods outlined in the following list to configure a bounded task flow to use train components:

- Select the **Create Train** checkbox in the dialog that appears when you create a new bounded task flow or task flow template.

  For more information, see Creating a Task Flow or Creating Task Flow Templates .
- Right-click an existing bounded task flow in the diagram and choose **Train** > **Create Train**.
- Open an existing bounded task flow in the overview editor, click the **Behavior** navigation tab and select the **Train** checkbox.

Once you have configured the bounded task flow to use train components, you drag and drop the task flow activities that you want to render in the train to the diagram in the bounded task flow.

How to Create a Train in a Bounded Task Flow

You drag and drop the task flow activities that you want to render in the train to the diagram for the bounded task flow.

Before you begin:

It may be helpful to have an understanding of the configuration options that you can configure for a train. For more information, see Creating a Task Flow as a Train .

You may also find it helpful to read about additional functionality that you can add using other task flow features. For more information, see Additional Functionality for Complex Task Flows.

You will need to complete this task:

> Configure the bounded task flow to use train components, as described in Using Train Components in Bounded Task Flows.

To create a train in a bounded task flow:

1. In the Applications window, double-click the bounded task flow or task flow template where you want to render train stops.
2. Drag each view activity or JSF page you want to include in the train to the diagram for the bounded task flow.
   - If you drag a JSF page, JDeveloper automatically adds a view activity to the diagram. You drag a JSF page that you want to include from the Applications window to the diagram for the bounded task flow.
   - If you drag a view activity to the diagram, you can double-click it to invoke the dialog to create a new JSF page. To drag a view activity, in the ADF Task Flow page of the Components window, from the Component panel, in the Activities group, drag and drop a **View** onto the diagram.

   You can reorder the train stop sequence at a later point. For more information, see Disabling the Sequential Behavior of Train Stops in a Train.
3. In the diagram for the bounded task flow, double-click each view activity that you want to define as a train stop.

   A dialog appears to create a new JSF page if the view activity is not yet associated with a JSF page. Use the dialog to create a new JSF page.

   If the view activity is already associated with a JSF page, the JSF page opens.
4. In the ADF Faces page of the Components window, from the General Controls panel, in the Location group, drag a **Train**, and, optionally, a **Train Button Bar** component and drop it on the JSF page.

   You must manually add each Train and Train Button Bar component that you want to appear in a task flow view activity. Consider using a page template. For more information, see Using Page Templates.

What Happens When You Create a Task Flow as a Train

JDeveloper writes the `<train/>` element to the source file for the bounded task flow or task flow template that you enable as a train, using one of the methods outlined in Creating a Task Flow as a Train .

JDeveloper writes entries to the JSF page where you add `train` and `trainButtonBar` components. The following example shows code snippets from the `GeneralInfo.jsff` page fragment in the Summit ADF task flow sample application. Both types of train component bind to instances of the train model object (`trainModel`).

```
 <af:train
value="#{controllerContext.currentViewPort.taskFlowContext.trainModel}"
         id="t1"/>
  ...
  <af:trainButtonBar
value="#{controllerContext.currentViewPort.taskFlowContext.trainModel}"
        id="tbb1"/>
```

Dotted lines appear between each view activity that you add to the bounded task flow and define as a train stop, as illustrated in Figure 27-9. The dotted lines indicate the order in which the bounded task flow navigates through the train stops. JDeveloper defines the sequence in the order that you add the view activities to the bounded task flow. You can change the sequence. For more information, see Disabling the Sequential Behavior of Train Stops in a Train. You can also skip a train stop. For more information, see Configuring a Train to Skip a Train Stop.

Invoking a Child Bounded Task Flow from a Train Stop

An alternative approach to grouping task flow activities in one train stop, as described in Grouping Task Flow Activities to Execute Between Train Stops , is to create another bounded task flow as a train that you invoke from your current bounded task flow. This bounded task flow is referred to as a **child bounded task flow**. You configure the parent bounded task flow to invoke it using a task flow call activity. To implement this functionality, you:

- Create a child bounded task flow as a train
- Add the task flow activities that you want end users to access from the train stop on the parent bounded task flow to the child bounded task flow
- Designate a task flow call activity as a train stop in the parent bounded task flow to invoke the child bounded task flow
- Add a task flow return activity to the child bounded task flow that returns control to the parent bounded task flow once the child bounded task flow finishes execution

Task flow activities in the child bounded task flow execute together regardless of whether the end user visits the train stop for the first time or returns at a later point. Nonview task flow activities in the child bounded task flow typically precede the view activity in the child bounded task flow's flow control. You can invoke a child bounded task flow from a bounded task flow that itself is a child to another bounded task flow.

How to Invoke a Child Bounded Task Flow From a Train Stop

You create a bounded task flow as a train, add task flow activities to it, and configure a task flow call activity on the parent bounded task flow to invoke it.

Before you begin:

It may be helpful to have an understanding of the configuration options that you can configure for a child bounded task flow that you invoke from a train. For more information, see Invoking a Child Bounded Task Flow from a Train Stop.

You may also find it helpful to read about the additional functionality that you can add using other task flow features. For more information, see Additional Functionality for Complex Task Flows.

You will need to complete this task:

Create a bounded task flow as a train. This is a child bounded task flow that is to be invoked from a task flow call activity in the parent task flow. Add the task flow activities that you require to it. For more information, see Creating a Task Flow as a Train .

To invoke a child bounded task flow from a train stop:

1. In the Applications window, double-click the bounded task flow that is the child bounded task flow configured as a train.
2. In the ADF Task Flow page of the Components window, from the ADF Task Flow panel, in the Activities group, drag and drop a **Task Flow Return** onto the diagram.
3. Configure the task flow return activity to return control to the parent task flow when the child bounded task flow finishes execution.

   For more information, see Using Task Flow Return Activities .
4. In the Applications window, double-click the parent bounded task flow that is going to invoke the child bounded task flow.

   For more information, see Creating a Task Flow as a Train .
5. In the ADF Task Flow page of the Components window, from the ADF Task Flow panel, in the Activities group, drag and drop a **Task Flow Call** onto the diagram.
6. Configure the task flow call activity to invoke the child bounded task flow that you configured in Steps 1 to 3.

   For more information, see Using Task Flow Call Activities.

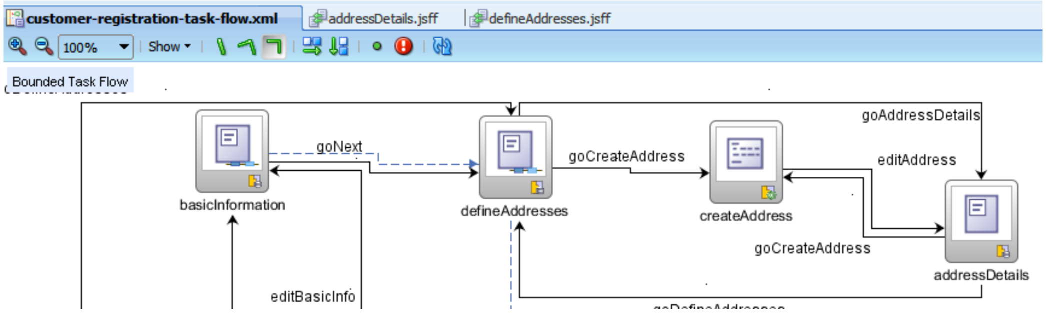
Grouping Task Flow Activities to Execute Between Train Stops

You can group task flow activities together to form a single train stop. For example, a train stop that renders as **Address** at runtime might group together the following task flow activities:

- `defineAddress` view activity
- `createAddress` method call activity
- `addressDetails` view activity

Each time a user visits the runtime **Address** train stop, the task flow would provide access to the task flow activities listed previously. The `defineAddresses` view activity renders the train stop and the associated `defineAddresses.jsff` page fragment. The `defineAddresses.jsff` page fragment, in turn, exposes a button to invoke the `createAddress` method call activity. This method call activity validates user input in the Address page and defines an outcome that (optionally) passes control to the `addressDetails` view activity. The `addressDetails` view activity renders the Address Information page at runtime where an end user may choose to enter additional addresses.

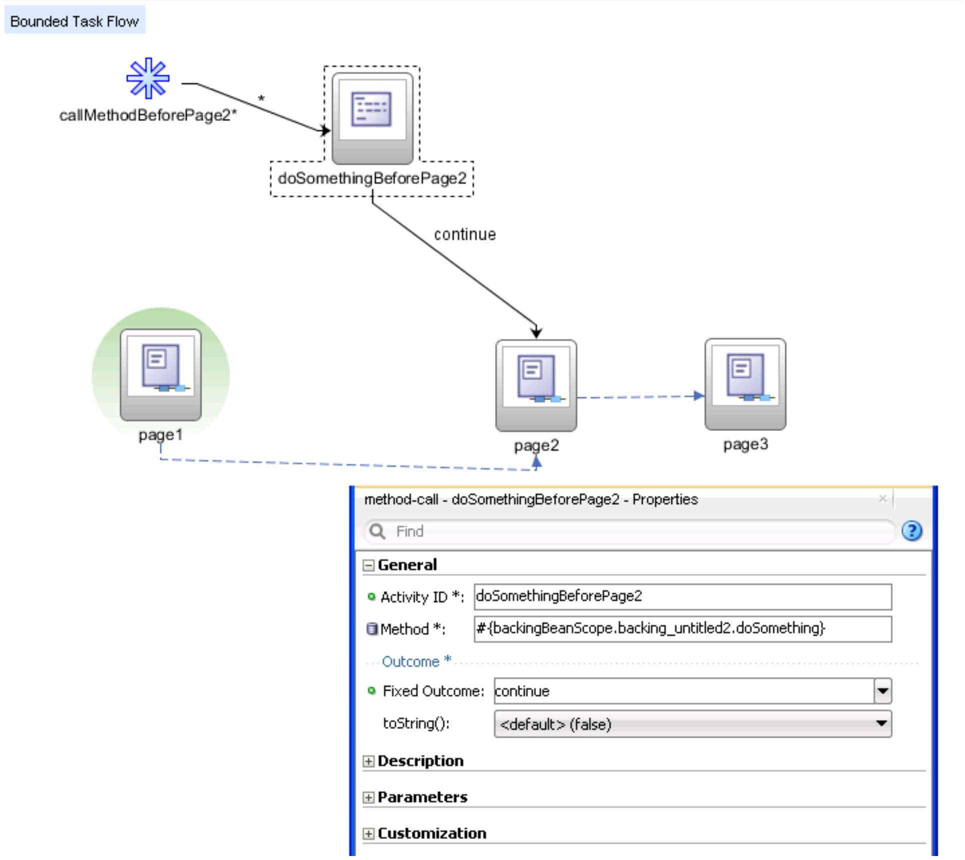
Figure 27-11   Task Flow Activities Grouped into One Train Stop



The train considers all task flow activities that lead from the first nonview activity through the next view activity to be part of the train. Make sure that all task flow activities, except for view and task flow call activities, for the train stop follow the view or task flow call activity that you define as a train stop.

You can also group task flow activities to execute between train stops. Figure 27-12 shows a possible configuration where a method call activity (`doSomethingBeforePage2`) invokes before control flow passes to `page2`. The `callMethodBeforePage2` wildcard control flow rule passes control to the `doSomethingBeforePage2` method call activity. This method call activity defines a fixed outcome (`continue`) that passes control to the next train stop (`page2`) in the train.

Figure 27-12   Method Call Activity Between Train Stops



Activity ID *: doSomethingBeforePage2

Method *: #{backingBeanScope.backing_untitled2.doSomething}

Outcome *

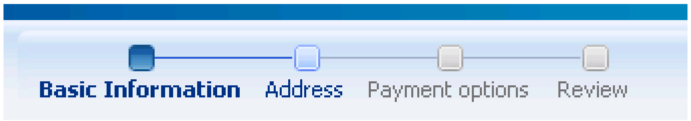
Fixed Outcome: continue

toString(): <default> (false)

Disabling the Sequential Behavior of Train Stops in a Train

By default, train stops are sequential. This means end users can select a train stop only after visiting the previous train stop in the train. Figure 27-13 shows a train component that a registration task flow renders. End users cannot visit the **Payment** options and **Review** train stops until they first visit the **Address** train stop. All train stops in this train are sequential.

Figure 27-13 Sequential Train Stops in a Task Flow



You can change this default behavior (make a train stop nonsequential) by configuring the view activity that renders the train stop to make it nonsequential.

How to Disable the Sequential Behavior of a Train

You write a value for the task flow view activity's `sequential` property that evaluates to `false` at runtime.

Before you begin:

It may be helpful to have an understanding of the configuration options that you can configure for a train. For more information, see Using Train Components in Bounded Task Flows.

You may also find it helpful to read about additional functionality that you can add using other task flow features. For more information, see Additional Functionality for Complex Task Flows.

To disable the sequential behavior of a train stop:

1. In the Applications window, double-click the bounded task flow containing the view activity that renders the train stop.
2. In the diagram for the bounded task flow, select the view activity.
3. In the Properties window, expand the **Train Stop** section and write a value in the **Sequential** field to determine if the train stop is sequential or nonsequential.

   Write `false` or write an EL expression that resolves to `false` at runtime to make the train stop nonsequential. For example, write an EL expression similar to the following:

   ```
   #{myTrainModel.isSequential}
   ```

What Happens When You Disable the Sequential Behavior of a Train Stop

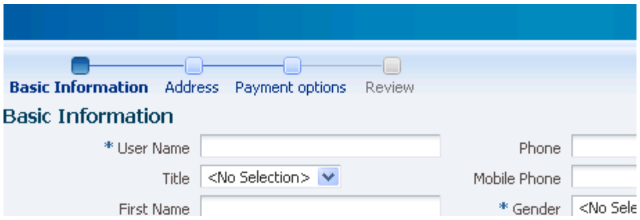
JDeveloper writes an entry to the source file for the bounded task flow, as illustrated in the following example.

```
<view id="paymentOptionDetails">
      <page>/account/paymentOptionDetails.jsff</page>
```

```
      <train-stop id="__1">
        <sequential>#{myTrainModel.isSequential}</sequential>
      </train-stop>
</view>
```

Figure 27-14 shows the corresponding runtime view when the value specified for `sequential` evaluates to `false`. End users can now navigate directly to payment options by clicking the **Payment options** train stop.

Figure 27-14   Train with a Nonsequential Train Stop



Changing the Label of a Train Stop

You can configure a train stop to display a label that you define. The label that you define can be a static value or a value in a resource bundle that you reference using an EL expression, for example. For more information about using localized strings, see the "Internationalizing and Localizing Pages" chapter of *Developing Web User Interfaces with Oracle ADF Faces*.

How to Change the Label of a Train Stop

You change the label of a train stop by configuring a value for the view activity's Display Name property.

Before you begin:

It may be helpful to have an understanding of the configuration options that you can configure for a train. For more information, see Using Train Components in Bounded Task Flows.

You may also find it helpful to read about additional functionality that you can add using other task flow features. For more information, see Additional Functionality for Complex Task Flows.

To change the label of a train stop:

1. In the Applications window, double-click the bounded task flow containing the view activity that renders the train stop.
2. In the diagram for the bounded task flow, select the view activity.
3. In the Properties window, expand the **Description** section and write a value in the **Display Name** to specify the label that the train stop renders at runtime.

   Write a literal value or write an EL expression that references a value in a resource bundle to render at runtime.

What Happens When You Change the Label of a Train Stop

JDeveloper writes an entry to the source file for the bounded task flow, as illustrated in the following example from the `edit-customer-task-flow-definition.xml` bounded task flow in the Summit ADF task flow sample application.

```
<view id="GeneralInfo"> <page>/customers/GeneralInfo.jsff</page>
  <train-stop>
    <display-name>General Information</display-name>
    <sequential>false</sequential>
  </train-stop></view>
```

At runtime, the Fusion web application displays the value you specified for the `display-name` property as the label for the train stop.

Configuring a Train to Skip a Train Stop

You can configure a train so that it skips an individual train stop. At runtime, the Fusion web application disables the train stop that you configure to skip. The end user can only navigate to the next train stop in the train.

Implement this functionality if you want the train to execute at a train stop other than the first train stop. For example, configure train stops 1 and 2 to skip if you want to execute train stop 3 first. Use this approach rather than defining the view activity associated with the train stop as the default activity, as discussed in What You May Need to Know About the Default Activity in a Bounded Task Flow.

How to Configure a Train to Skip a Train Stop

You write a value for the view activity's `skip` property that evaluates to `true` at runtime.

Before you begin:

It may be helpful to have an understanding of the configuration options that you can configure for a train. For more information, see Using Train Components in Bounded Task Flows.

You may also find it helpful to read about additional functionality that you can add using other task flow features. For more information, see Additional Functionality for Complex Task Flows.

To configure a train to skip a train stop:

1. In the Applications window, double-click the bounded task flow containing the view activity that renders the train stop.
2. In the diagram for the bounded task flow, select the view activity.
3. In the Properties window, expand the **Train Stop** and write a value in the **Skip** field to determine if the train navigates to the train stop or skips it.

   Write `true` or write an EL expression that resolves to `true` at runtime to make the train skip the train stop. For example, write an EL expression similar to the following:

   ```
   #{myTrainModel.shouldSkip}
   ```

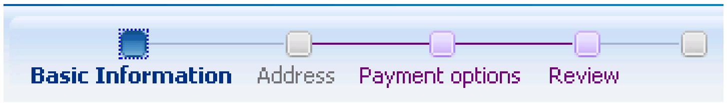
What Happens When You Configure a Train to Skip a Train Stop

JDeveloper writes an entry to the source file for the bounded task flow, as illustrated in the following example.

```
 <view id="defineAddresses">
      <display-name>Address</display-name>
      <page>/account/defineAddresses.jsff</page>
      <train-stop>
        <display-name>Address</display-name>
        <skip>#{myTrainModel.shouldSkip}</skip>
      </train-stop>
</view>
```

Figure 27-15 shows the corresponding runtime view when the value specified for `skip` evaluates to `true`. End users must now navigate to the next train stop in the train (**Payment** options) if the train is sequential.

Figure 27-15 Train Configured to Skip a Train Stop

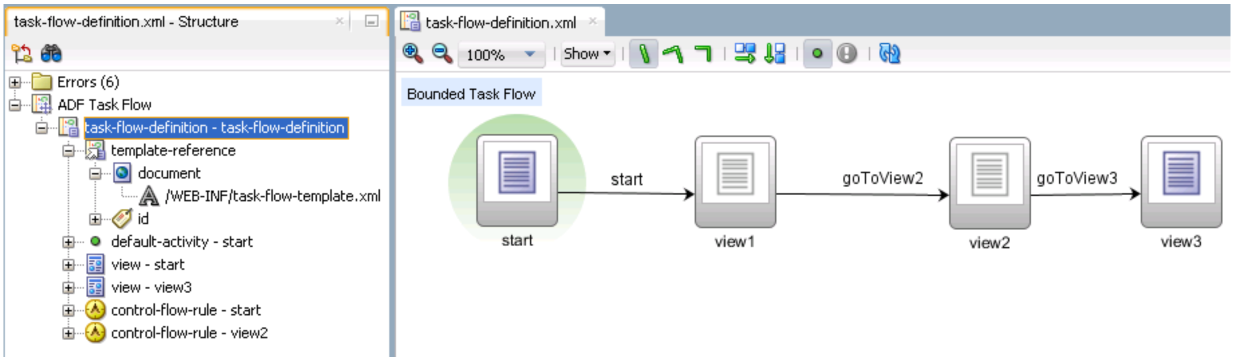


Creating Task Flow Templates

When you have a repeated set of ADF task flow activities, control flows, input parameters, and managed beans, you can create ADF task flow templates for reuse. A template serves as a starting point when creating a new bounded task flow. However, you can create templates for unbounded task flows.

You can create task flow templates for yourself or other application developers to use as a starting point when creating new bounded task flows. Unbounded task flows cannot be created using task flow templates. A bounded task flow created from a task flow template will have definitions for the same set of task flow activities, control flows, input parameters, and managed beans as the task flow template. For example, Figure 27-16 shows a bounded task flow that has been created from a task flow template. The `view1` and `view2` view activities plus the `goToView2` control flow in this bounded task flow have been defined in the task flow template while the `start` and `view3` view activities have been defined in the bounded task flow itself.

Figure 27-16 Bounded Task Flow Displaying Activities Defined in Task Flow Template



An example use case for a task flow template is where you define a task flow activity as an exception handler on a bounded task flow. The task flow activity could be a view activity associated with a page that you display to end users when an exception occurs. Rather than define this view activity in each bounded task flow, you define it in a task flow template and select the task flow template when you create new bounded task flows. For information about exception handling, see Handling Exceptions in Task Flows.

You can base a new task flow template on an existing task flow template. You can also refactor an existing bounded task flow to create a new task flow template. See How to Convert Bounded Task Flows .

When you create a bounded task flow or another task flow template based on a task flow template, you can specify that subsequent changes you make to a task flow template be automatically propagated to bounded task flows and templates. To do this, you select the **Update the Task Flow When the Template Changes** checkbox in the dialog that you use to create a bounded task flow or a template. Subsequent changes that you make to the task flow template (add new view activities, for example) get propagated to the bounded task flows. You can change, update, or disassociate the parent task flow template of a child bounded task flow or task flow template at any point during development of the child.

At runtime, the contents of a child bounded task flow or a child task flow template combine with the contents of the parent task flow template if you selected the **Update the Task Flow When the Template Changes** checkbox when creating the child. The child task flow and task flow template override the parent task flow template when conflict occurs, even if you selected the **Update the Task Flow When the Template Changes** checkbox when creating the child task flow or task flow template. For example, you create a parent task flow template to use as a train, as described in Using Train Components in Bounded Task Flows, and then create a bounded task flow based on this task flow template. Later, you disable the train component on the parent task flow template. The child task flow overrides the parent task flow template and continues to execute as a train.

Table 27-3 describes in more detail how ADF Controller resolves conflicts between parent task flow templates and child task flows and child task flow templates.

Table 27-3 Conflict Resolution between Parent Templates and Child Task Flows

Bounded Task Flow Metadata	Combination Algorithm
Default activity	Child bounded task flow or child task flow template overrides parent task flow template.
Transaction	Child bounded task flow or child task flow template overrides parent task flow template as an entire block of metadata, including all subordinate elements.
Task flow reentry	Child bounded task flow or child task flow template overrides parent task flow template as an entire block of metadata, including all subordinate elements.
Control flow rules	Combination algorithm occurs at the control flow case level, not the control flow rule level. Control flow cases fall into the following categories: • Both from action and from outcome specified • Only from action specified • Only from outcome specified • Neither from action nor from outcome specified Each of these categories is merged additively. The child bounded task flow or template overrides parent task flow template for identical matches within each of the four categories.
Input parameter definitions	Child bounded task flow or child task flow template overrides parent task flow template for identical input parameter definition names.
Return value definitions	Child bounded task flow or child task flow template overrides parent task flow template for identical return value definition names.
Activities	Child bounded task flow or child task flow template overrides parent task flow template for identical activity IDs.
Managed beans	Child bounded task flow or child task flow template overrides parent task flow template for identical managed bean names.
Initializer	Child bounded task flow or child task flow template overrides parent task flow template.
Finalizer	Child bounded task flow or child task flow template overrides parent task flow template.
Critical	Child bounded task flow or child task flow template overrides parent task flow template.
Use page fragments	Child bounded task flow or child task flow template overrides parent task flow template.
Exception handler	Child bounded task flow or child task flow template overrides parent task flow template.
Security - permission	Child bounded task flow or child task flow template overrides parent task flow template. Privilege maps are additive. Child bounded task flow or child task flow template overrides parent task flow template for identical privilege map operations.
Security - transport guarantee	Child bounded task flow or child task flow template overrides parent task flow template.

Table 27-3 (Cont.) Conflict Resolution between Parent Templates and Child Task Flows

Bounded Task Flow Metadata	Combination Algorithm
Train	Child bounded task flow or child task flow template overrides parent task flow template.

Validations at both design time and runtime verify that the resulting parent-child extension hierarchy does not involve cycles of the same task flow template.

How to Create a Task Flow Template

You create a task flow template by selecting ADF Task Flow Template from the New Gallery.

Before you begin:

It may be helpful to have an understanding of what task flow template options you can configure. For more information, see Creating Task Flow Templates .

You may also find it helpful to read about additional functionality that you can add using other task flow features. For more information, see Additional Functionality for Complex Task Flows.

To create a task flow template:

1. In the Applications window, right-click the project where you want to create the task flow template and choose **New** > **From Gallery**.
2. In the New Gallery, expand **Web Tier**, select **JSF/Facelets** and then **ADF Task Flow Template**, and click **OK**.
3. In the Create ADF Task Flow Template dialog, choose the appropriate options:
   - **File Name**: Accept the default value proposed by JDeveloper or enter a new file name. The value is used to name the XML source file for the task flow template you create. The source file includes the activities and control flow rules that are in the task flow template. The default name for the XML source file is `task-flow-template.xml`.
   - **Create with Page Fragments**: Leave this checkbox selected (the default) if you expect that a bounded task flow based on the template will be used as an ADF region. Clear the checkbox if you want to add JSF pages instead of JSF page fragments to a task flow based on the task flow template.
4. Click **OK**.

What Happens When You Create a Task Flow Template

As shown in the following example, an XML file is created each time you create a new task flow template using JDeveloper. You can find the XML file in the Applications window in the location that you specified in the **Directory** field of the Create ADF Task Flow Template dialog, for example,`.../WEB-INF`.

The contents of the XML source file for the task flow template can be similar to those of a bounded task flow. One difference is the inclusion of the `<task-flow-template>` tag.

```
<?xml version="1.0" encoding="windows-1252" ?>
<adfc-config xmlns="http://xmlns.oracle.com/adf/controller" version="1.2"
id="__1">
  <task-flow-template id="task-flow-template">
     <default-activity>view1</default-activity>
     <view id="view1">view1.jsff</view>
  </task-flow-template>
</adfc-config>
```

What You May Need to Know About Task Flow Templates

If you use a task flow template that contains bindings, you must change the component IDs of task flows based on the task flow template. Doing this makes sure that the IDs are unique. Task flows generated from the template inherit the same ID as the template. This may cause an exception at runtime. For more information, see How to Use ADF Data Binding in ADF Page Templates.

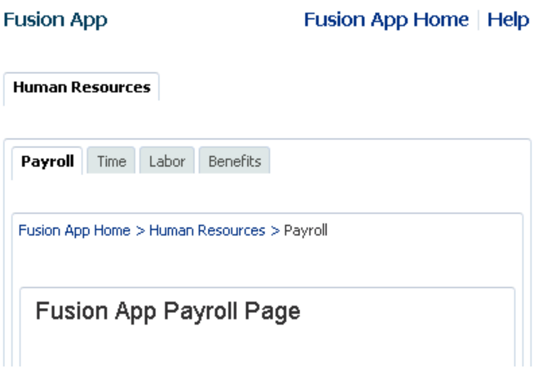
If your Fusion web application has configured ADF security, you have to make grants on both the task flow template and task flows that use the template. Usually, the grant on the task flow template is to the `anonymous-role` application role so that the grant that really matters is on the task flow, but not always. For more information about ADF security, see Enabling ADF Security in a Fusion Web Application.

Creating a Page Hierarchy Using Task Flows

The page hierarchy is a useful way of organizing the JSF pages in your Fusion web application. You can create a page hierarchy for ADF task flows using ADF Controller or using an XMLMenuModel implementation. You can allow users to access information on the pages by navigating a path of links.

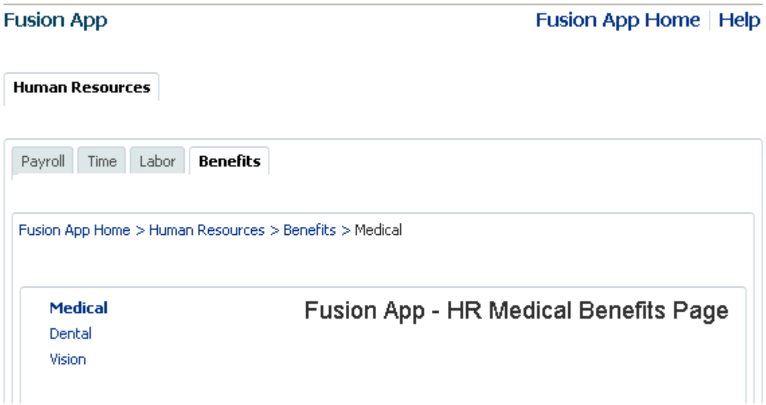
Creating a page hierarchy is a useful way of organizing the JSF pages in your Fusion web application so that end users can more easily navigate the application. End users access information on the pages by navigating a path of links. Figure 27-17 shows a sample page hierarchy that allows the user to navigate between the home page of a Fusion application, a Human Resource page, and a Payroll page.

Figure 27-17   Page Hierarchy



To navigate this hierarchy, an end user clicks links on each page to drill down or up to another level of the hierarchy. For example, clicking **Human Resources** on the Fusion App home page displays the Human Resources page hierarchy shown in Figure 27-17. Clicking the link on the **Benefits** tab displays the page hierarchy shown in Figure 27-18.

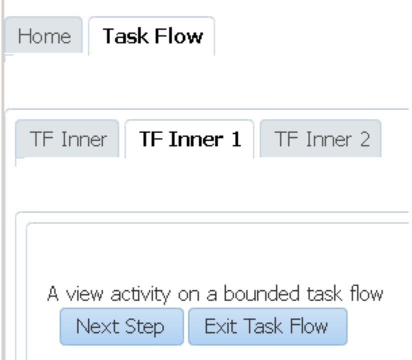
Figure 27-18 Benefits Page


Fusion App
Fusion App Home | Help
Human Resources
Payroll
Time
Labor
Benefits
Fusion App Home > Human Resources > Benefits > Medical
Medical
Dental
Vision
Fusion App - HR Medical Benefits Page

The user can click links on the Benefits page to display other pages, for example, the Medical, Dental or Vision pages. The breadcrumbs on each page indicate where the current page fits in the hierarchy. The user can click each node in a breadcrumb to navigate to other pages in the hierarchy. The bold tab labels match the path to the current page shown by the breadcrumbs.

Pages referenced by view activities in a bounded task flow can also be included in any page hierarchy that you generate. Figure 27-19 shows the runtime view of a page hierarchy that renders view activities referenced by a bounded task flow.

Figure 27-19 Runtime Menu Hierarchy Including a Bounded Task Flow



You can use ADF Controller with an `XMLMenuModel` implementation to create the previously discussed page hierarchies. If you do, JDeveloper generates the following for you:

- Control flow metadata that determines the view or page to display when an end user selects a menu item

- An `XMLMenuModel` metadata file
- Default navigation widgets such as **Previous** and **Next** buttons
- Breadcrumbs
- Managed bean configuration

If you decide not to use ADF Controller, you can create the page hierarchy using an `XMLMenuModel` implementation. For information about this method of building a page hierarchy, see Using a Menu Model to Create a Page Hierarchy.

How to Create a Page Hierarchy

Create an unbounded task flow or open an existing one. Add view activities or bounded task flows to the unbounded task flow. Each view activity or bounded task flow that you add to the unbounded task flow contains references to pages to appear in the proposed page hierarchy. Use JDeveloper's Create ADF Menu Model dialog to generate an `XMLMenuModel` metadata file. Organize the item nodes in the generated `XMLMenuModel` metadata file to create the page hierarchy you want. Connect submenus to parent menus to finalize the hierarchy.

Figure 27-20 shows an example page hierarchy that consists of view activities:

- The top-level menu (**Home Page**) is the root parent page. It contains a single tab that links to the **Human Resources** submenu.

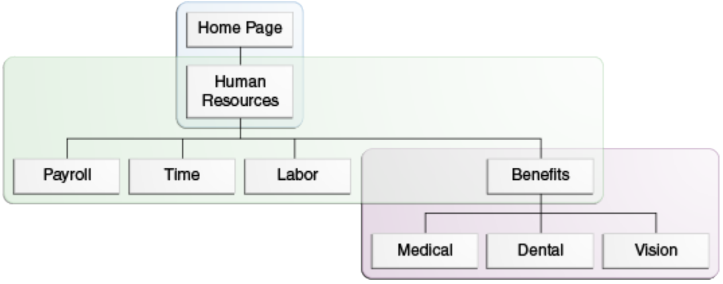
  In JDeveloper, **Home Page** page is represented as an item node and **Human Resources** page as a shared node.

- **Human Resources** has four tab links to **Payroll**, **Time**, **Labor**, and **Benefits** pages.

  In this menu, **Human Resources** is a group node that references child item nodes (**Payroll**, **Time**, and **Labor**) and a shared node (**Benefits**) that references the Benefits submenu.

- **Benefits** is a group node that references child item nodes (**Medical**, **Dental**, and **Vision**) pages.

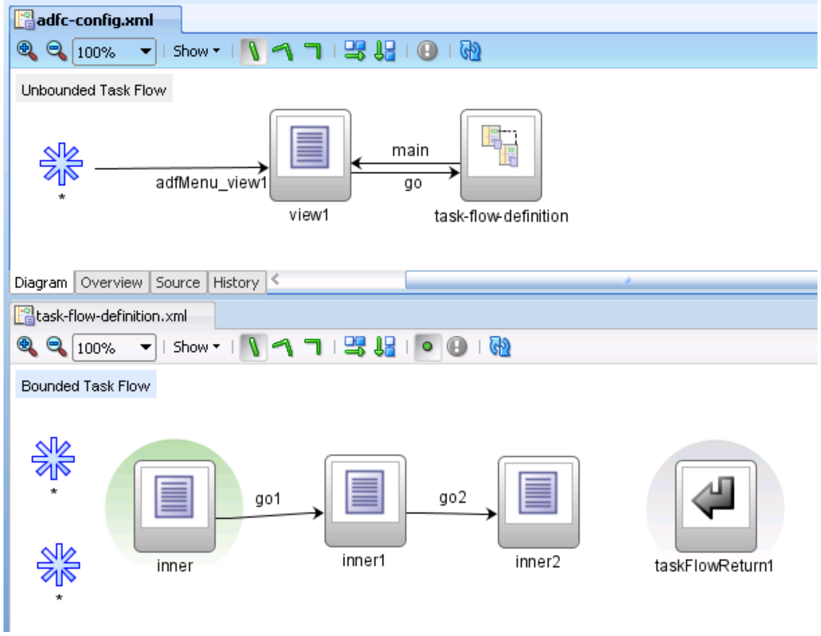
Figure 27-20 Menu Hierarchy



> **Note:**
>
> It is possible to create the entire menu hierarchy in one menu model. However, breaking a menu hierarchy into submenus makes maintenance easier. In addition, breaking the menu hierarchy into smaller submenu models enables each separate development organization to develop its own menu. These separate menus can later be combined using shared nodes to create the complete menu hierarchy.

Figure 27-21 shows the corresponding design-time view in JDeveloper of the unbounded and bounded task flows that render the page hierarchy shown in Figure 27-19. The unbounded task flow (`adfc-config.xml`) contains a view activity (**view1**) and a task flow call activity (**task-flow-definition**) that invokes the bounded task flow (`task-flow-definition.xml`) shown in the lower part of Figure 27-21.

Figure 27-21 Design Time Menu Hierarchy Including a Bounded Task Flow



How to Create an XMLMenuModel Metadata File

You use JDeveloper's Create ADF Menu Model dialog to generate an `XMLMenuModel` metadata file once you have defined what menus (unbounded task flows) and nodes (pages) you want to appear in the final page hierarchy.

Before you begin:

It may be helpful to have an understanding of what page hierarchy options you can configure. For more information, see Creating a Page Hierarchy Using Task Flows.

You may also find it helpful to read about additional functionality that you can add using other task flow features. For more information, see Additional Functionality for Complex Task Flows.

You will need to complete these tasks:

1. Create an unbounded task flow for each menu in the final page hierarchy.

   For example, to achieve the page hierarchy illustrated in Figure 27-20, you create two unbounded task flows (Human Resources menu and Benefits menu).

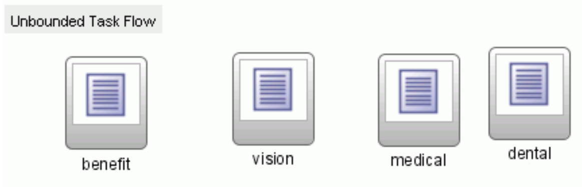
   For more information about creating an unbounded task flow, see How to Create a Task Flow.

   > **Tip:**
   >
   > Prefix the name of the file for unbounded task flows that you create with `adfc-` to help you to identify the file as the source of an unbounded task flow, as opposed to a bounded task flow.

2. Add view activities that reference pages to each unbounded task flow. The pages referenced by the view activities correspond to the menu nodes in the menu.

   For example, the Benefits menu contains one group node (benefits) and three item nodes (medical, dental and vision) so you add four view activities to the unbounded task flow for the Benefits menu, as illustrated in Figure 27-22.

   Figure 27-22 View Activities on a Task Flow



   Do not add view activities for menus that include other menus using shared nodes. For example, the Human Resources menu in Figure 27-20 has a tab called **Benefits** that references the Benefits menu using a shared node. The bounded task flow for the Benefits menu already includes a view activity for Benefits so there is no need to add a view activity to the bounded task flow for the Human Resources menu.

   For more information about view activities, see Using View Activities.

   > **Note:**
   >
   > If the page hierarchy includes pages referenced by a bounded task flow, add a task flow call activity to the unbounded task flow that calls the bounded task flow.

To create the XMLMenuModel metadata file:

1. In the Applications window, right-click the file(s) for each of the unbounded task flows you created and choose **Create ADF Menu Model**.

2. In the Create ADF Menu Model dialog, enter a file name for the `XMLMenuModel` metadata file and a directory to store it.
3. Click **OK**.

How to Create a Submenu with a Hierarchy of Group and Child Nodes

You open the `XMLMenuModel` metadata file you created and convert the item nodes that you want to make group nodes to group nodes. You then create a hierarchy where a group node is a parent to one or more item nodes.

Before you begin:

It may be helpful to have an understanding of what page hierarchy options you can configure. For more information, see Creating a Page Hierarchy Using Task Flows.

You may also find it helpful to read about additional functionality that you can add using other task flow features. For more information, see Additional Functionality for Complex Task Flows.

To create a submenu with a hierarchy of group and item nodes:

1. In the Applications window, double-click the `XMLMenuModel` metadata file.

   An item node appears in the Structure window for each view activity in the unbounded task flow. By default, no hierarchy is defined.

2. In the Structure window, drag and drop the item nodes to become child nodes of the item node that you are going to convert to a group node.

   Each item node that you convert to a group node must have at least one child item node. For example, to create the menu hierarchy in Figure 27-20, you convert the item node for Benefits to a group node, you drag and drop the item nodes for Medical, Dental, and Vision so that they become child nodes of the Benefit item node.

3. In the Structure window, right-click the parent item node and choose **Convert To groupNode**.
4. In the groupNode Properties dialog, in the **id** field, enter a new identifier or accept the default value.

   The identifier must be unique among all of the nodes in all of the `XMLMenuModel` metadata files. It is good practice to specify a value that identifies the node. For example, if you change the Benefits node to a group node, you can update its default ID, `itemNode_benefits`, to `groupNode_benefits`.

5. In the **idref** field, enter the ID of one of the other nodes in the menu, for example, `itemNode_Medical`.

   The value you enter can be an ID of a child item node that is a group node or an item node.

6. Enter or change the existing default value in the **label** field to match what you want to appear at runtime.

   For example, you might change `label_benefits` to `Benefits`.

7. Accept the rest of the default values in the fields and click **Finish**

   A Confirm Convert dialog asks if you want to delete the `action` and `focusViewID` attributes on the `groupNode` element. Group nodes do not use these attributes, always click **OK**

8. Click **OK**

How to Attach a Menu Hierarchy to Another Menu Hierarchy

You use a shared node element to link two menus together. For example, the Human Resources menu shown in the menu hierarchy in Figure 27-20 contains four submenus (Payroll, Time, Labor, and Benefits). The Benefits submenu is itself a menu with submenu entries. In the `XMLMenuModel` metadata file for the Human Resources menu, you convert the item node for the Benefits submenu to a shared node. You write an EL expression for an attribute (`ref`) of the newly created shared node that references the `XMLMenuModel` metadata file for the Benefits menu.

Before you begin:

It may be helpful to have an understanding of what page hierarchy options you can configure. For more information, see Creating a Page Hierarchy Using Task Flows.

You may also find it helpful to read about additional functionality that you can add using other task flow features. For more information, see Additional Functionality for Complex Task Flows.

To attach a menu hierarchy to another hierarchy using a shared node:

1. In the Applications window, double-click the `XMLMenuModel` metadata file for the menu that is going to reference the other menu.
2. In the Structure window, select a node, right-click and select the appropriate menu options to insert a `sharedNode` element.
3. In the Insert sharedNode dialog, in the **ref** field, enter an EL expression to reference the `XMLMenuModel` metadata file for the other menu.
4. Click **OK**.

> **Note:**
>
> If your page hierarchy has more than one unbounded task flow, make sure that the file name for each additional unbounded task flow appears as a value for the `<metadata-resources>` element in the `adfc_config.xml` file. For more information, see What Happens When You Create a Page Hierarchy.

What Happens When You Create a Page Hierarchy

Changes occur in a number of different files when you create a page hierarchy.

Changes to the adfc-config.xml File

When you create a new unbounded task flow, JDeveloper automatically adds a reference in the `adfc-config.xml` file to the source file for the newly created unbounded task flow. In the following example, `adfc-unbounded_tflow.xml` is the name of the source file for a newly created unbounded task flow.

```
<?xml version="1.0" encoding="windows-1252" ?>
<adfc-config xmlns="http://xmlns.oracle.com/adf/controller" version="1.2"
             id="__1">
```

```
  <metadata-resource id="__2">
      /WEB-INF/adfc-unbounded_tflow.xml
  </metadata-resource>
</adfc-config>
```

For more information about `adfc-config.xml`, see adfc-config.xml.

At runtime, the Fusion web application loads the `adfc-config.xml` file when it first starts. The `adfc-config.xml` file can contain:

- ADF navigation metadata for an unbounded task flow
- ADF activity metadata for an unbounded task flow
- Managed bean definitions used by ADF activities

XMLMenuModel Metadata File

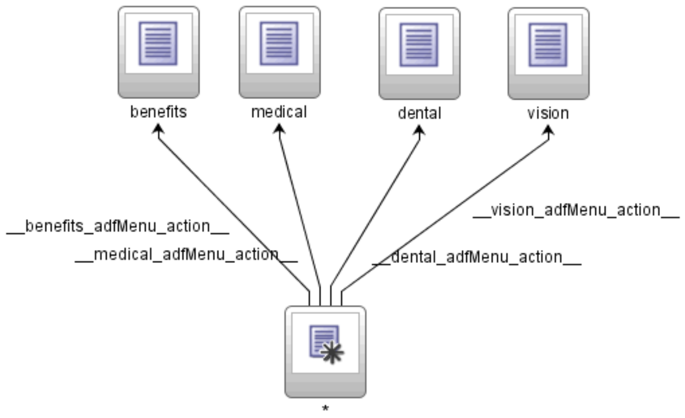
JDeveloper generates an `XMLMenuModel` metadata file with nodes for each of the view activities that you added to the unbounded task flow, as illustrated in the following example.

```
<?xml version="1.0" encoding="windows-1252" ?>
<menu xmlns="http://myfaces.apache.org/trinidad/menu">
  <groupNode id="groupNode_benenfits" label="Benefits" idref="itemNode_medical">
    <itemNode id="itemNode_medical" label="label_medical"
              action="adfMenu_medical" focusViewId="/medical"/>
    <itemNode id="itemNode_dental" label="label_dental" action="adfMenu_dental"
              focusViewId="/dental"/>
    <itemNode id="itemNode_vision" label="label_vision" action="adfMenu_vision"
              focusViewId="/vision"/>
  </groupNode>
</menu>
```

Diagram for an Unbounded Task Flow

JDeveloper updates the file for the unbounded task flow with the control flow rules and managed beans used to navigate the page hierarchy. Figure 27-23 shows the updated unbounded task flow in the diagrammer that corresponds to the unbounded task flow in Figure 27-22.

Figure 27-23 Updated Unbounded Task Flow


benefits
medical
dental
vision
__benefits_adfMenu_action__
__medical_adfMenu_action__
__vision_adfMenu_action__
__dental_adfMenu_action__
*

Reporting Incidents to the Oracle Fusion Middleware Diagnostic Framework

An Oracle Fusion Middleware Diagnostic Framework enables you to collect and retrieve information about incidents that occur in Fusion web applications. You can use the collected information to troubleshoot problems or to send it to Oracle Support for resolution. ADF Controller can report incidents, such as version information, hierarchy of the ADF binding container, metadata and so on, to this framework.

The Diagnostic Framework collects and manages information about incidents that occur in Fusion web applications. ADF Controller can report incidents to this framework. You can use this information to resolve problems or send it to Oracle Support for resolution. Examples of the information that the ADF Controller reports to the Diagnostic Framework when an incident occurs include the following:

- Version information
- Information about the view port hierarchy, including page flow stack entries
- Hierarchy of the ADF binding container
- Metadata information, for example the list of files that contribute to the unbounded task flow

Furthermore, information about recent events within the runtime of the ADF Controller can also be sent to the Diagnostic Framework. Examples of these events includes permission checks, region refreshes, navigation between activities, and the retrieval of metadata from the Oracle Metadata Services (MDS) framework. Sending this latter category of information (events within the runtime of ADF Controller) requires you to enable QuickTrace, a mechanism that Oracle Fusion Middleware provides to enable fine-grained logging to memory. For information about QuickTrace, including how to specify the history (number) of recent events that ADF Controller sends to the Diagnostic Framework, see Configuring and Using QuickTrace. For information about using the Diagnostic Framework, see Diagnosing Problems.

28

Using Dialogs in Your Application

This chapter describes how to create dialogs in your Fusion web application. It describes how you can use ADF Controller and task flows to create dialogs or, alternatively, use the ADF Faces dialog framework.
This chapter includes the following sections:

About Using Dialogs in Your Application

ADF dialogs are the secondary windows that can display additional information external to the browser window displaying the current page. You can also use modal dialogs that allows users to work on both primary and secondary windows at the same time.

Use dialogs if you want to show information to end users in a secondary browser window external to the browser window that displays the end user's current page. For example, you want to display help information to end users to assist them with a task in the primary browser window or you want end users to choose a value from a list of values. The help information example is a use case where a modeless dialog is appropriate. A **modeless dialog** allows end users work in both the primary window and the dialog at the same time. For the use case where you want an end user to choose a value, a modal dialog is more appropriate. A **modal dialog** prevents an end user accessing the page that invoked the dialog until they execute an action requested by the dialog (for example, choose a value).

Use the ADF Faces dialog framework if you want to configure modeless dialogs for your end users. If you plan to configure modal dialogs for your end users, configure an ADF Controller bounded task flow to invoke one or more dialogs. You can define view activities that reference either a JSF page or a page fragment file (`.jsff`) to invoke a dialog.

Using Dialogs in Your Application Use Cases and Examples

Running a Bounded Task Flow in a Modal Dialog describes how you can run a bounded task flow in a modal dialog to retrieve input from an end user, and return to a view activity that called the bounded task with the retrieved input. If your application does not use ADF Controller task flows or you want to use modeless dialogs, Using the ADF Faces Dialog Framework Instead of Bounded Task Flows, describes how you can use the ADF Faces dialog framework to render one or more pages in a dialog.

Additional Functionality for Using Dialogs in Your Application

You may find it helpful to understand other **Oracle ADF** features before you configure or use dialogs in your application. Additionally, you may want to read about what you can do with the dialogs you configure. Following are links to other functionality that may be of interest.

- If your application uses ADF Controller task flows, you may find it helpful to understand more about the features that task flows offer. For more information, see Getting Started with ADF Task Flows .
- For more information about ADF Faces components, see *Developing Web User Interfaces with Oracle ADF Faces*.
- You can also write custom code for your dialogs using the APIs provided by Oracle ADF. Make sure when you write custom code that you do not import packages that are marked internal, as in this example:

  ```
  import oracle.adfinternal.controller.*;
  ```

  For information about the APIs that you can use to write custom code, see the following reference documents:

  - *Java API Reference for Oracle ADF Controller*
  - *Java API Reference for Oracle ADF Faces*

Running a Bounded Task Flow in a Modal Dialog

In the cases where you want to accept the values from a user and return the application to an activity, running an ADF bounded task flow in a modal dialog is appropriate to be used. The modal dialogs can be used in the ADF Controller and task flows by specifying the dialog:syntax in the control flow rules.

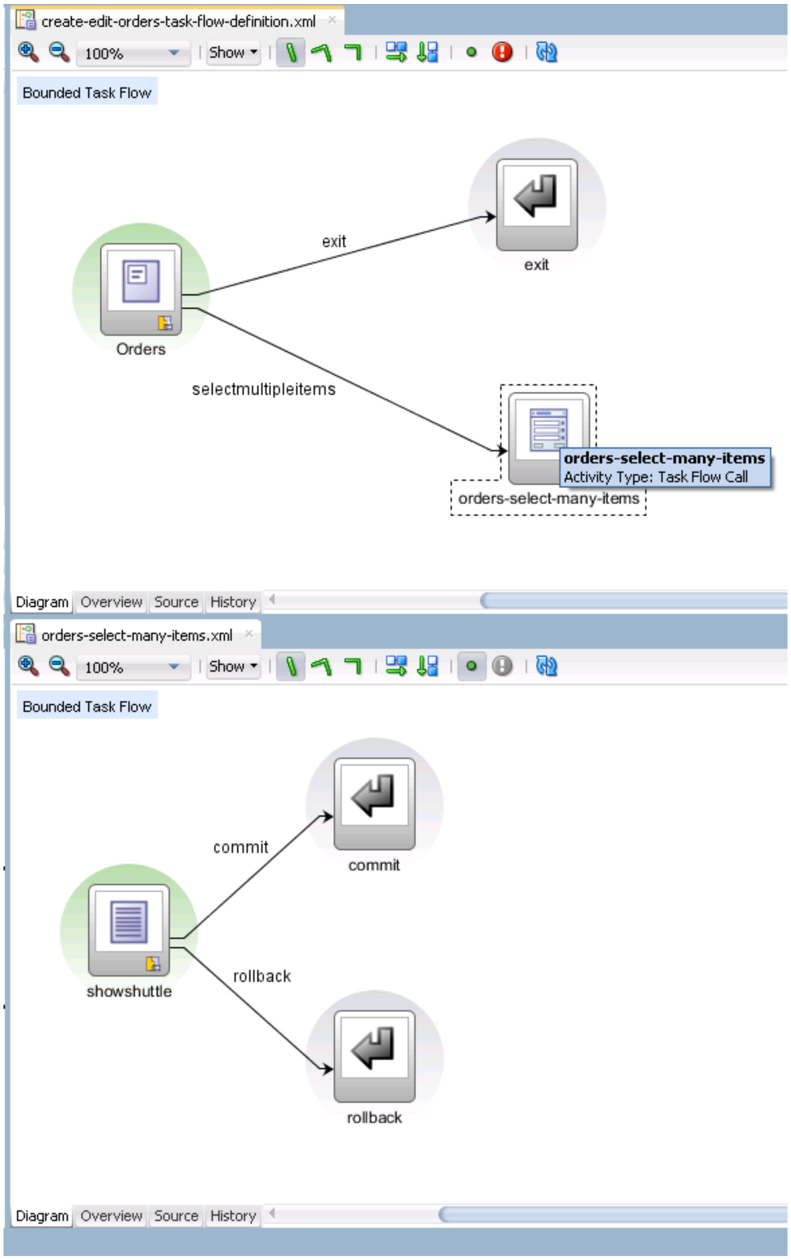
You can configure a bounded task flow to run in a modal dialog, retrieve input from an end user, and return to the view activity that called the bounded task flow with the retrieved input. The bounded task flow that you configure must reference pages, not page fragments.

Figure 28-1 shows an example of the configuration required from the Summit sample application for ADF task flows. The task flow in `create-edit-orders-task-flow-definition` contains a view activity (**Orders**) from where the end user invokes the `orders-select-many-items.xml` task flow that renders a modal dialog. For more information, see How to Run a Bounded Task Flow in a Modal Dialog.

When the end user closes the dialog, control and any modified values return to the launch page in the calling task flow in `create-edit-orders-task-flow-definition`. For more information, see How to Return a Value From a Modal Dialog.

In addition, you can configure the application to refresh part of the launch page after the modal dialog returns control. For more information, see How to Refresh a Page After a Modal Dialog Returns.

Figure 28-1 Task Flow Activities to Invoke a Modal Dialog in Summit ADF Sample Application



How to Run a Bounded Task Flow in a Modal Dialog

You add a view activity and a task flow call activity to an existing task flow. The view activity invokes a page where an end user can invoke an action that, in turn, invokes the bounded task flow to appear in a modal dialog.

Before you begin:

It may be helpful to have an understanding of how the attributes you configure affect the functionality of a bounded task flow in a modal dialog. For more information, see Running a Bounded Task Flow in a Modal Dialog.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Using Dialogs in Your Application.

To run a bounded task flow in a modal dialog box:

1. In the Applications window, double-click the existing task flow.
2. In the diagram for the task flow, double-click the view activity to open the associated page.
3. Select the command component that the end user clicks at runtime to invoke the bounded task flow as a modal dialog box (for example, a `button` component).
4. In the Properties window, expand the **Common** section and enter the control flow case to invoke the bounded task flow in the **Action** field.

   For example, `selectmultipleitems` in Figure 28-1.
5. From the **UseWindow** dropdown list, select **true** to invoke the bounded task flow in a popup dialog.
6. In the diagram for the existing task flow, select the task flow call activity.
7. In the Properties window, expand the **Behavior** section and select **true** from the **Run As Dialog** dropdown list to run the bounded task flow as a dialog.

   The bounded task flow that the task flow activity calls can now behave as a dialog; it can be invoked by the command component you configured in Steps 3 to 5 and can return a value to the view activity that renders the command component, as described in How to Return a Value From a Modal Dialog. Note that you need to make these other changes to run a bounded task flow in a modal dialog. Setting the `run-as-dialog` attribute to `true` is not sufficient.
8. Select **external-window** (the default value) from the **Display Type** dropdown list if you want to render the dialog in an external browser window or **inline-popup** if you want to render the dialog in the same browser window.

How to Return a Value From a Modal Dialog

You can configure a bounded task flow that renders in a modal dialog to return a value to the view activity that invoked the bounded task flow when the end user dismisses the modal dialog. The returned value can, for example, be displayed in an input component on the page associated with the view activity.

You must configure the bounded task flow that is called by the task flow call activity to declare input parameters and return values. For more information, see Passing Parameters to a Bounded Task Flow.

You specify a method binding for a method with one argument (a return event) as the value for the `returnListener` attribute of the command component (for example, a `button` component). The `returnListener` attribute sets this value in the input component on the page associated with the view activity. Specify a backing bean for the input component and set the input component's `partialTrigger` attribute to the ID of the command component.

You also need to specify:

- A return value definition on the called bounded task flow to indicate where to take the return value from upon exit of the called bounded task flow.

- Return values on the task flow call activity in the existing task flow to indicate where the existing task flow can find return values.

  For more information, see Configuring a Return Value from a Bounded Task Flow.

For more information about creating backing beans, input components, and command components, see the "Using Input Components and Defining Forms" chapter in *Developing Web User Interfaces with Oracle ADF Faces*.

Before you begin:

It may be helpful to have an understanding of how the attributes you configure affect the functionality of bounded task flows in modal dialogs. For more information, see Running a Bounded Task Flow in a Modal Dialog.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Using Dialogs in Your Application.

You will need to complete this task:

> Configure a bounded task flow to run in a modal dialog. For more information, see How to Run a Bounded Task Flow in a Modal Dialog.

To specify a return value:

1. In the Applications window, double-click the task flow that contains the task flow call activity to invoke the bounded task flow to render in a modal dialog.
2. In the diagram editor, select the task flow call activity.
3. In the Properties window, expand the **Behavior** section and select **true** from the **Run As Dialog** dropdown list.
4. In the **Dialog Return Value** field, enter the name of the return value definition you specified for the target bounded task flow.

   For information about how to specify the return value definition on a bounded task flow, see Configuring a Return Value from a Bounded Task Flow.
5. In the Applications window, double-click the page that launches the modal dialog.
6. In the design editor, select the input component and, in the Properties window, expand the **Behavior** section to specify an EL expression in the **PartialTriggers** field.

   The EL expression you specify identifies the command component, accepts the return value of the command component, and specifies a backing bean. For example, enter an EL expression with syntax similar to the following:

   ```
   #{pageFlowScope.backingBean.gotoModalDialog}
   ```

   where `gotoModalDialog` identifies the command component.
7. In the design editor, select the command component and, in the Properties window, expand the **Behavior** section and enter an EL expression that references a return listener method in the page's backing bean as a value in the **ReturnListener** field.

   The return listener method you specify processes the return event that is generated when an end user dismisses the modal dialog. In the Summit ADF task flow sample application, the command component in the `Orders.jsff` page fragment specifies the following EL expression:

```
#{viewScope.OrdersBackingBean.onShuttleTaskFlowReturn}
```

How to Refresh a Page After a Modal Dialog Returns

You can configure the page from where end users invoke a modal dialog to refresh after end users close the modal dialog. You may want to configure this behavior if the modal dialog returns a value or end users edit existing data in the page through controls in the modal dialog.

You implement this functionality by configuring the command component that invokes the modal dialog to listen for a return event. When it receives the return event, it invokes a backing bean method that executes a partial page render event on your page. The following example shows a `button` component that invokes a modal dialog and listens for a return event.

```
<af:button text="Edit"
    id="b1" binding="#{backingBeanScope.backing_launch_page.b1}"
    action="edit" useWindow="true"
    returnListener="#{backingBeanScope.backing_launch_page.backFromPopup}"/>
```

The `button` component's `returnListener` attributes listens for the return event from invoking the modal dialog. When it receives the return event, it invokes a backing bean method similar to the following example.

```
public void backFromPopup(ReturnEvent returnEvent) {
        AdfFacesContext adfFacesContext;
        adfFacesContext = AdfFacesContext.getCurrentInstance();
        adfFacesContext.addPartialTarget(this.getF1());
}
```

The backing bean method (`backFromPopup`) takes the return event as an argument and, in our example, invokes a partial page render event on a form in the page.

Before you begin:

It may be helpful to have an understanding of how you configure other options for modal dialogs that you invoke from a page associated with a view activity in a task flow. For more information, see Running a Bounded Task Flow in a Modal Dialog.

You may also find it helpful to understand other functionality that can be added using other task flow and dialog framework features. For more information, see Using Dialogs in Your Application Use Cases and Examples.

To refresh a page after a modal dialog returns:

1. In the Applications window, double-click the task flow that invokes the bounded task flow that renders in a modal dialog.
2. In the diagram, double-click the view activity that references the page where your end user invokes a command component to launch the modal dialog.
3. In the design editor, select the command component.
4. In the Properties window, expand the **Behavior** section and, from the **ReturnListener** field under the **Secondary Window** label, select **Edit** from the icon that appears when you hover over the property field to specify the name of the backing bean method that you want JDeveloper to generate.
5. In the Edit Property: ReturnListener dialog, click **New** beside the **Managed Bean** field.

6. In the Create Managed Bean dialog, name the bean and the class, set the bean's scope, and click **OK**.
7. In the Edit Property: ReturnListener dialog, click **New** beside the **Method** field.
8. In the Create Method dialog, name the method and click **OK**.
9. In the Edit Property: ReturnListener dialog, click **OK**.
10. In the Applications window, expand **Application Sources** and then expand the package that contains the backing bean class and method that you created in Steps 3 to 9.
11. Write a method body to invoke a partial page render event on your page.

What You May Need to Know About Dialogs in an Application that Uses Task Flows

If your Fusion web application renders modeless dialogs (non modal dialogs without task flows) and also uses ADF Controller features, such as task flows, you specify the `dialog:syntax` in the control flow rules of the application's `adfc-config.xml` file rather than the `faces-config.xml` file. Specify `dialog:syntax` in navigation rules within the `faces-config.xml` file if your Fusion web application does not use ADF Controller features, as described in Using the ADF Faces Dialog Framework Instead of Bounded Task Flows.

The following example shows what you can specify in the `adfc-config.xml` file. Note that although this example references a JSP page, you can define a view activity that invokes either a JSF page or a page fragment file.

```
<?xml version="1.0" encoding="windows-1252" ?>
  <adfc-config xmlns="http://xmlns.oracle.com/adf/controller" version="1.2"
id="__1">
    <view id="view1"
        <page>/view1.jsf</page>
    </view>
    <view id="dialog">
       <page>/dialog/untitled1.jsf</page>
   </view>
   <control-flow-rule>
        <from-activity-id>test</from-activity-id>
        <control-flow-case>
          <from-outcome>dialog:test</from-outcome
          <to-activity-id>dialog</to-activity-id>
         </control-flow-case>
</adfc-config>
```

Using the ADF Faces Dialog Framework Instead of Bounded Task Flows

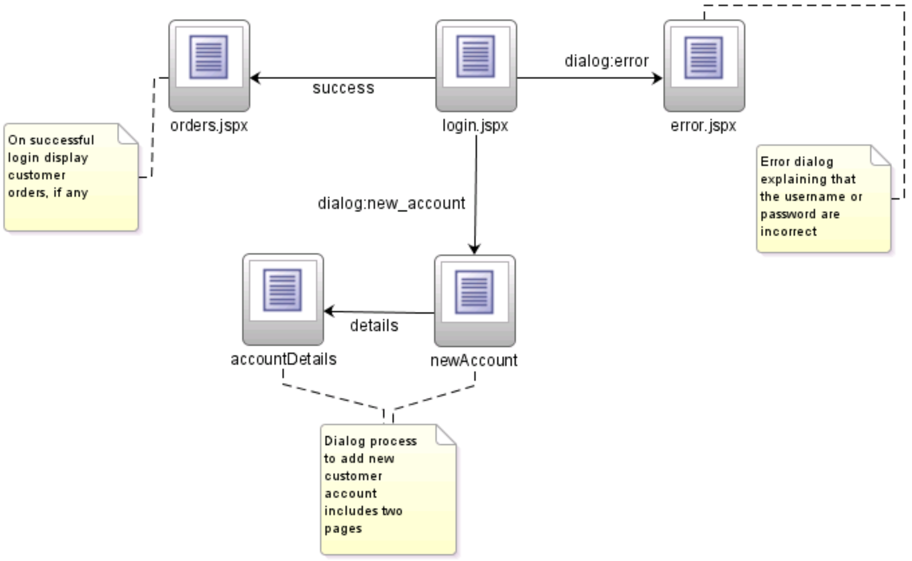
The ADF Faces dialog framework displays a page or a series of pages in a new browser window. To create modal and modeless dialogs in an application using ADF Faces dialog framework, you must define a JSF navigation rule to open a dialog, create the JSF page from which the dialog is opened, and create dialog page and return dialog values.

You can use the ADF Faces dialog framework to create modal and modeless dialogs in an application that does not use ADF Controller and task flows. The dialog framework enables you to display a page or series of pages in a new browser window instead of displaying it in the same window (using the same view ID) as the current page. There may also be cases where you want to use a series of inline dialogs, that is, dialogs that are part of the parent page, but that have a flow of their own, but that do not use a separate view ID. This is important for applications that do not support popups such as, for example, applications that run on client devices or that use the Active Data Service described in Using the Active Data Service . Ordinarily, you would need to use JavaScript to open the dialog and manage the process. With the dialog framework, ADF Faces makes it easy to open a new browser window as well as manage dialogs and processes without using JavaScript.

> **Note:**
>
> If your application uses the Fusion technology stack with ADF Controller, then you should use task flows to create dialogs launched in a separate window, or multiple dialog processes. See Running a Bounded Task Flow in a Modal Dialog.

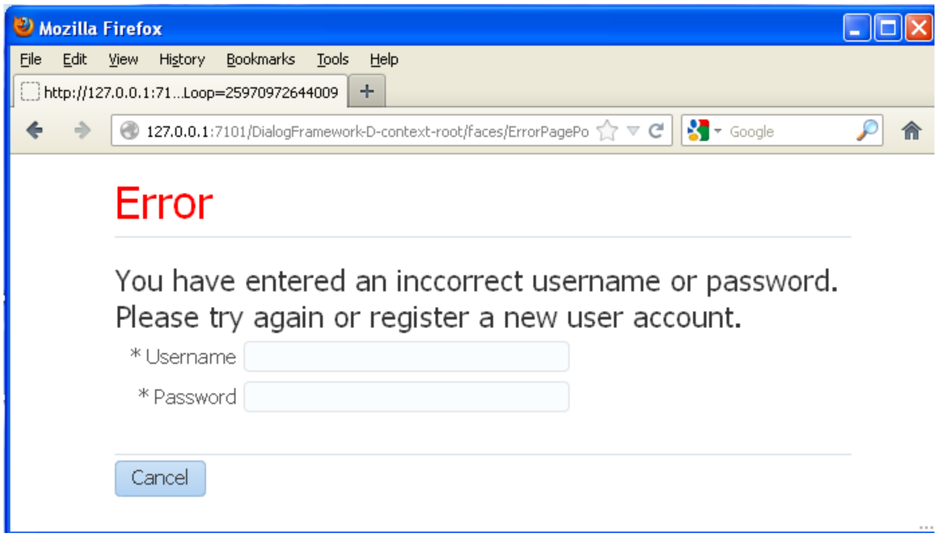
Consider a simple application that requires users to log in to see their orders. Figure 28-2 shows the page flow for the application, which consists of five pages: `login.jspx`, `orders.jspx`, `new_account.jspx`, `account_details.jspx`, and `error.jspx`.

Figure 28-2 Page Flow of an External Dialog Sample Application


success
dialog:error
orders.jspx
login.jspx
error.jspx
On successful login display customer orders, if any
Error dialog explaining that the username or password are incorrect
dialog:new_account
details
accountDetails
newAccount
Dialog process to add new customer account includes two pages

When an existing user logs in successfully, the application displays the Orders page, which shows the user's orders, if any. When a user does not log in successfully, the Error page displays in a separate popup dialog window, as shown in Figure 28-3.

Figure 28-3 Error Page Popup


Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://127.0.0.1:71...Loop=25970972644009
127.0.0.1:7101/DialogFramework-D-context-root/faces/ErrorPagePo
Google

Error

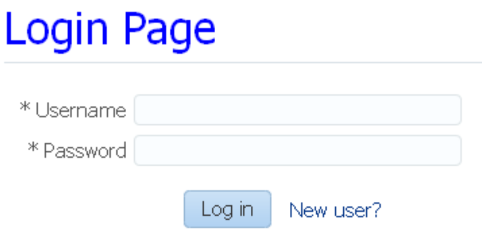
You have entered an inccorrect username or password. Please try again or register a new user account.

* Username

* Password

On the Error page there is a **Cancel** button. When the user clicks **Cancel**, the popup dialog closes and the application returns to the Login page and the original flow, as shown in Figure 28-4.

Figure 28-4 Login Page



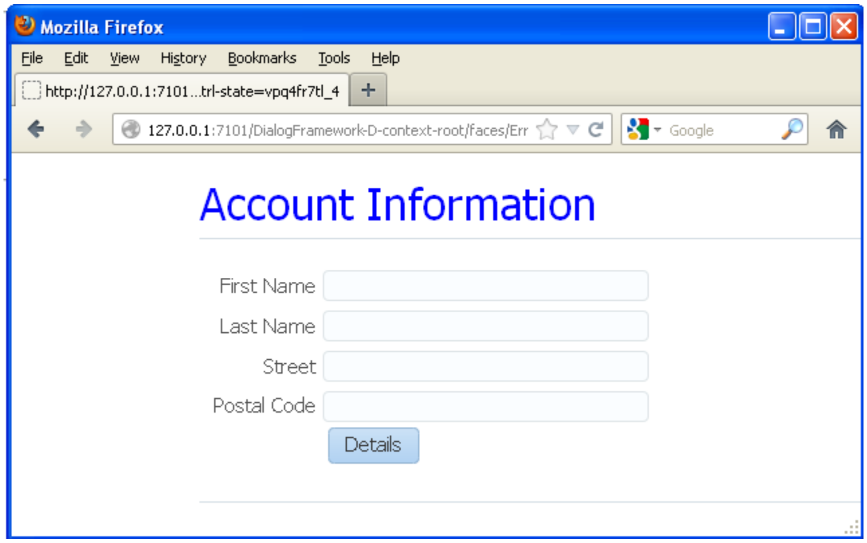
Login Page

* Username

* Password

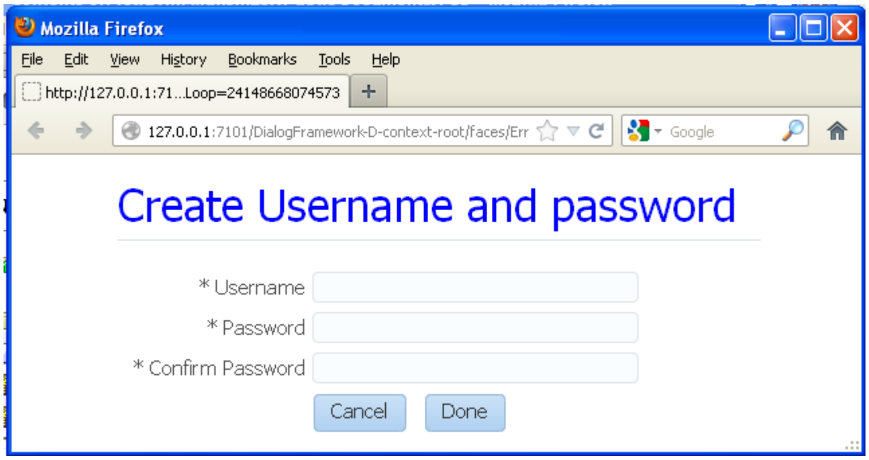
When a new user clicks the **New User** link on the Login page, the New Account page displays in a popup dialog in a new window, as shown in Figure 28-5.

Figure 28-5 New Account Page in a Separate Window


Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://127.0.0.1:7101...trl-state=vpq4fr7tl_4
127.0.0.1:7101/DialogFramework-D-context-root/faces/Err
Google
Account Information
First Name
Last Name
Street
Postal Code
Details

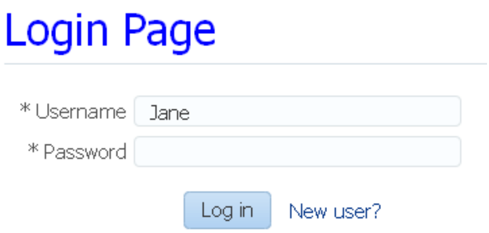
After entering information such as first name and last name, the user then clicks the **Details** button to display the Account Details page in the same popup dialog, as shown in Figure 28-6. In the Account Details page, the user enters other information and confirms a password for the new login account. There are two buttons on the Account Details page: **Cancel** and **Done**.

Figure 28-6 Account Details Page in a Popup Dialog


Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://127.0.0.1:71...Loop=24148668074573
127.0.0.1:7101/DialogFramework-D-context-root/faces/Err
Google
Create Username and password
* Username
* Password
* Confirm Password
Cancel
Done

If the new user decides not to proceed with creating a new login account and clicks **Cancel**, the popup dialog closes and the application returns to the Login page. If the new user clicks **Done**, the popup dialog closes and the application returns to the Login page where the **Username** field is now populated with the user's first name, as shown in Figure 28-7. The new user can then proceed to enter the new password and log in successfully.

Figure 28-7 LogIn Page with Username Field Populated



Login Page

* Username Jane

* Password

Note:

The dialog framework should not be used to have more than one dialog open at a time, or to launch dialogs that have a life span outside of the life span of the base page.

To make it easy to support dialog page flows in your applications, ADF Faces has built in the dialog functionality to action components (such as `commandMenuItem` and `button`). For ADF Faces to know whether or not to open a page in a new flow from an action component, the following conditions must exist:

- There must be a JSF navigation rule with an outcome that begins with `dialog:`.
- The command component's action outcome must begin with `dialog:`.
- The `useWindow` attribute on the command component must be `true`.

Note:

If the `useWindow` attribute is `false`, or if you configure the popup to be a separate window (and not inline) and the client device does not support popups, ADF Faces automatically shows the page in the current window instead of using a popup window; code changes are not required to facilitate this action.

The page that displays in a dialog is an ordinary JSF page, but for purposes of explaining how to implement external dialogs in this chapter, a page that displays in a popup dialog is called the *dialog* page, and a page from which the dialog is opened is called the *originating* page. A **dialog process** starts when the originating page opens a dialog (which can contain one dialog page or a series of dialog pages), and ends when the user dismisses the dialog and returns to the originating page.

To create a dialog page flow in an application:

1. Define a JSF navigation rule for opening a dialog.
2. Create the JSF page from which a dialog is opened.
3. Create the dialog page and return a dialog value.

4. Optional: Pass a value into a dialog.
5. Handle the return value.

The tasks can be performed in any order.

How to Define a JSF Navigation Rule for Opening a Dialog

You manage the navigation into a dialog flow by defining a standard JSF navigation rule with a special `dialog:` outcome.

To define a navigation rule to open a dialog:

1. In the Applications window, expand the WEB-INF node and double-click **adfc-config.xml**.
2. In the diagram editor, create a page flow for your originating page and dialog pages.

   For more information, see How to Add a Control Flow Rule to a Task Flow.
3. When creating navigation rules to the dialog pages, the outcome must begin with `dialog:`. For example, in the login sample application shown in Figure 28-2, the outcome from the Login page to the New Account dialog page is `dialog:newAccount`.

At runtime, the dialog navigation rules on their own simply show the specified pages in the originating page. But when used with command components with `dialog:` action outcomes and with `useWindow` attributes set to `true`, ADF Faces opens the pages in dialogs.

How to Create the JSF Page That Opens a Dialog

In the originating page, you need to use a command component to launch the dialog. The command component's action value needs to be the outcome to the dialog that is to be launched.

Before you begin:

It may be helpful to have an understanding of how the attributes you configure affect the functionality of the ADF Faces dialog framework. For more information, see About Using Dialogs in Your Application.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Using Dialogs in Your Application.

You will need to complete this task:

> You also need to create the JSF page that opens a dialog. For more information, see the "Creating a View Page" section in *Developing Web User Interfaces with Oracle ADF Faces*.

To create the JSF Page that opens a dialog:

1. In the Applications window, double-click the JSF page.
2. In the ADF Faces page of the Components window, from the General Controls panel, drag and drop a command component (for example, a **Button**) onto the JSF page.

For more information about adding a command component to a page, see the "Using Buttons and Links for Navigation" section in *Developing Web User Interfaces with Oracle ADF Faces.*

Note the following when setting the attributes on the command component:

- **Action**: Set the action attribute to the outcome that navigates to the dialog, as created in How to Define a JSF Navigation Rule for Opening a Dialog.

  > **Tip:**
  >
  > The action value can be either a static string or the return of a method on a managed bean.

  For example, the action attribute on the command component of the Login page is bound to a method that determines whether to navigate to the Orders page or to the Error dialog page, based on the returned outcome. If the method returns `dialog:error`, the error dialog opens. If the method returns success, the user navigates to the Orders page.

- **ActionListener**: As an alternative to setting the action attribute, configure the `actionListener` attribute to invoke the `launchDialog` method from an instance of the following class:

  ```
  oracle.adf.view.rich.context.AdfFacesContext
  ```

  For more information about the `launchDialog` method and the `AdfFacesContext` class, see the *Java API Reference for Oracle ADF Faces*.

- **UseWindow**: Set to `true` to have the dialog open.

  > **Tip:**
  >
  > When set to `false`, ADF Faces shows the dialog page in the current window after preserving all of the state of the current page. You do not have to write any code to facilitate this.

- **WindowHeight** and **WindowWidth**: Set the desired size of the dialog window. These values will set the `contentWidth` and `contentHeight` attributes on the popup component for the dialog.

  > **Tip:**
  >
  > While the user can change the values of these attributes at runtime, the values will not be retained once the user leaves the page unless you configure your application to use change persistence. For information about enabling and using change persistence, see the "Allowing User Customization on JSF Pages" chapter in *Developing Web User Interfaces with Oracle ADF Faces*.

- **PartialSubmit**: Set to true. This prevents the originating page from reloading (and hence being visible only momentarily) when the popup dialog is displayed.

- **WindowEmbedStyle**: Set to `inlineDocument` if you want the dialog to open in a popup that belongs to the originating page. Set to `window` if you want the dialog to open in a separate browser.
- **WindowModalityType**: Set to `applicationModal` if you want the dialog to be modal. Modal dialogs do not allow the user to return to the originating page until the dialog has been dismissed. Set to `modeless` if you want the user to be able to go back and forth between the originating page and the dialog.

When a command component is about to open a dialog, it delivers a `LaunchEvent` event. The `LaunchEvent` event stores information about the component that is responsible for opening a popup dialog, and the root of the component tree to display when the dialog process starts. A `LaunchEvent` can also pass a map of parameters into the dialog. For more information, see How to Pass a Value into a Dialog.

How to Create the Dialog Page and Return a Dialog Value

A dialog page is just like any other JSF page, with one exception. In a dialog page, you must provide a way to tell ADF Faces when the dialog process finishes, that is, when the user dismisses the dialog or series of dialogs.

For example, the New Account page and Account Details page belong in the same dialog process. A dialog process can have as many pages as you desire, you only need to notify the framework that the dialog process has ended once.

You do this declaratively using the `returnActionListener` tag as a child to the command component used to close the dialog. However, if you need to provide a return value or other action event processing, you can bind the `actionListener` attribute on the command component to a method that calls the `AdfFacesContext.returnFromDialog()` method. This method lets you send back a return value in the form of a `java.lang.Object` or a `java.util.Map` of parameters. You do not have to know where you are returning the value to - ADF Faces automatically takes care of it.

At runtime, the `AdfFacesContext.returnFromDialog()` method tells ADF Faces when the user dismisses the dialog. This method can be called whether the dialog page is shown in a popup dialog or in the main window. If a popup dialog is used, ADF Faces automatically closes it.

Before you begin:

It may be helpful to have an understanding of how the attributes you configure affect the functionality of the ADF Faces dialog framework. For more information, see About Using Dialogs in Your Application.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Using Dialogs in Your Application.

To close a dialog window and optionally return a value:

1. In the Applications window, double-click the dialog page.
2. In the ADF Faces page of the Components window, from the General Controls panel, drag and drop a command component (for example, a **Button**) onto the dialog page.

For more information about adding a command component to a page, see the "Using Buttons and Links for Navigation" section in *Developing Web User Interfaces with Oracle ADF Faces.*

3. If the command component will be used to close the window, in the Properties window, expand the **Behavior** section and select **true** from the **Immediate** dropdown list.
4. If the command component will be used to navigate to another page in the dialog process, configure the command component as though it were standard navigation. To do this, in the Properties window, expand the **Common** section and select **false** from the **UseWindow** dropdown list. This causes the next page to display in the same dialog window, preserving the state of the previous page.
5. If you need to end the dialog process and close the dialog, but do not need to return a value, in the Components window, from the Operations panel, in the Listeners group, drag a **Return Action Listener** and drop it as a child to the command component.

   The `returnActionListener` tag calls the `returnFromDialog` method on the `AdfFacesContext` object - no backing bean code is needed.

   No attributes are used with the `af:returnActionListener` tag. The `immediate` attribute on the `af:button` component is set to `true` if the user clicks **Cancel** without entering values in the required **Password** and **Confirm Password** fields, the default JSF `ActionListener` can execute during the Apply Request Values phase instead of the Invoke Application phase, thus bypassing input validation. For more information, see the "Using the JSF Lifecycle with ADF Faces" chapter in *Developing Web User Interfaces with Oracle ADF Faces*.
6. If you need to end the dialog process and do need to return a value, create a method on a managed bean that handles the `action` event and returns the needed values using the `returnFromDialog` method on the current instance of `AdfFacesContext`.

   > **Note:**
   >
   > The `AdfFacesContext.returnFromDialog()` method returns `null`. This is all that is needed in the backing bean to handle the Cancel action event.

   For example, when the user clicks **Done** on the Account Details page, the process ends and returns the user input values. The following example shows the code for the event handler method to which the **Done** button binds. The method gets the customer information, then either creates a Faces message for an incorrect password, or sets the values on the new customer object and return that object.

```
public void done(ActionEvent e)
{
  AdfFacesContext afContext = AdfFacesContext.getCurrentInstance();
  String firstname = afContext.getPageFlowScope().get("firstname").toString();
  String lastname = afContext.getPageFlowScope().get("lastname").toString();
  String street = afContext.getPageFlowScope().get("street").toString();
  String zipCode = afContext.getPageFlowScope().get("zipCode").toString();
  String country = afContext.getPageFlowScope().get("country").toString();
  String password = afContext.getPageFlowScope().get("password").toString();
  String confirmPassword =
   afContext.getPageFlowScope().get("confirmPassword").toString();
  if (!password.equals(confirmPassword))
  {
    FacesMessage fm = new FacesMessage();
    fm.setSummary("Confirm Password");
    fm.setDetail("You've entered an incorrect password. Please verify that you've
     entered a correct password!");
    FacesContext.getCurrentInstance().addMessage(null, fm);
  }
  else
  {
    //Get the return value
    Customer cst = new Customer();
    cst.setFirstName(firstname);
    cst.setLastName(lastname);
    cst.setStreet(street);
    cst.setPostalCode(zipCode);
    cst.setCountry(country);
    cst.setPassword(password);
    // And return it
    afContext.getCurrentInstance().returnFromDialog(cst, null);
  }
}
```

What Happens at Runtime: How to Raise the Return Event from the Dialog

When the dialog is dismissed, ADF Faces generates a return event (ReturnEvent). The AdfFacesContext.returnFromDialog() method sends a return value as a property of the return event. The return event is delivered to the return listener (ReturnListener) that is registered on the command component that opened the dialog (for example, the **New User** link on the Login page). How you handle the return value is described in How to Handle the Return Value.

How to Pass a Value into a Dialog

To pass a value into a dialog, you use a LaunchListener listener bound to a handler method for the LaunchEvent. You can use the getDialogParameters() method to add a parameter to a Map using a key-value pair.

Before you begin:

It may be helpful to have an understanding of how the attributes you configure affect the functionality of the ADF Faces dialog framework. For more information, see About Using Dialogs in Your Application.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Using Dialogs in Your Application.

To pass a value into a dialog:

1. In the Applications window, double-click the JSF page that contains the command component used to navigate to the dialog page.
2. In the design editor, select the command component used to navigate to the dialog page.
3. In the Properties window, expand the **Behavior** section.

4. From the icon that appears when you hover over the **LaunchListener** property field, select **Edit**.
5. In the Edit Property: LaunchListener dialog, click **New** beside the **Managed Bean** field.
6. In the Create Managed Bean dialog, name the bean and the class, set the bean's scope, and click **OK**.
7. In the Edit Property: LaunchListener dialog, click **New** beside the **Method** field.
8. In the Create Method dialog, name the method and click **OK** and click **OK** to close the Edit Property: LaunchListener dialog.
9. In the Applications window, expand **Application Sources** and then expand the package that contains the bean class and method that you created in Steps 3 to 8.
10. In the source editor, write a handler method for the `LaunchEvent` that uses the `getDialogParameters` method to get the parameters from a dialog.

    For example, in the sample application, a new user can enter a name in the **Username** field on the Login page, and then click the **New User?** link. When the New Account dialog page displays in a popup dialog, the **First Name** input field is automatically populated with the name that was entered in the Login page. To accomplish this, you create a handler that uses the `getDialogParameters` method to put the value of the username field into the dialog, as shown in the following example.

    ```
    public void handleLaunch(LaunchEvent event)
    {
      //Pass the current value of the field into the dialog
      Object usr = username;
      event.getDialogParameters().put("firstname", getUsername());
    }
    // Use by inputText value binding
    private String username;
    public String getUsername()
    {
      return username;
    }
    public void setUsername(String username)
    {
      this.username = username;
    }
    ```

11. In the Applications window, double-click the dialog page.
12. On the dialog page, use the `pageFlowScope` object to retrieve the key and value via a special EL expression in the format `#{pageFlowScope.someKey}`, as shown in the following example.

    ```
    <af:inputText label="First name" value="#{pageFlowScope.firstname}"/>
    ```

What Happens at Runtime: How the LaunchEvent is Handled

In ADF Faces, a process always gets a copy of all the values that are in the `pageFlowScope` of the page from which a dialog is launched. When the `getDialogParameters()` method has added parameters to a `Map`, those parameters also become available in `pageFlowScope`, and any page in the dialog process can get the values out of `pageFlowScope` by referring to the `pageFlowScope` objects via EL expressions.

> **Note:**
>
> Unlike `sessionScope`, `pageFlowScope` values are visible only in the current page flow or process. If the user opens a new window and starts navigating, that series of windows has its own process; values stored in each window remain independent. Clicking on the browser's Back button automatically resets `pageFlowScope` to its original state. When you return from a process the `pageFlowScope` is back to the way it was before the process started. To pass values out of a process you use `AdfFacesContext.returnFromDialog()`, `sessionScope` or `applicationScope`.

How to Handle the Return Value

To handle a return value once the dialog is dismissed, you define a return listener on the command component that launched the dialog. For example, in the sample application, once a new user enters information, that information needs to be handled once the dialog process is complete.

Before you begin:

It may be helpful to have an understanding of how the attributes you configure affect the functionality of the ADF Faces dialog framework. For more information, see About Using Dialogs in Your Application.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Using Dialogs in Your Application.

To handle the return value:

1. In the Applications window, double-click the JSF page that contains the command component used to navigate to the dialog page.
2. In the design editor, select the command component used to navigate to the dialog page.
3. In the Properties window, expand the **Behavior** section.
4. From the icon that appears when you hover over the **ReturnListener** property field, select **Edit**.
5. In the Edit Property: ReturnListener dialog, click **New** beside the **Managed Bean** field.
6. In the Create Managed Bean dialog, name the bean and the class, set the bean's scope, and click **OK**.
7. In the Edit Property: ReturnListener dialog, click **New** beside the **Method** field.
8. In the Create Method dialog, name the method and click **OK** and click **OK** to close the Edit Property: LaunchListener dialog.
9. In the Applications window, expand **Application Sources** and then expand the package that contains the bean class and method that you created in Steps 3 to 8.

10. In the source editor, write a handler method for the `returnEvent`. You use the `getReturnValue()` method to retrieve the return value, because the return value is automatically added as a property of the `ReturnEvent`.

    The following example shows the code for the return listener method that handles the return value.

    ```
    public void handleReturn(ReturnEvent event)
    {
      if (event.getReturnValue() != null)
      {
        Customer cst;
        String name;
        String psw;
        cst = (Customer)event.getReturnValue();
        name = cst.getFirstName();
        psw = cst.getPassword();
        CustomerList.getCustomers().add(cst);
        inputText1.setSubmittedValue(null);
        inputText1.setValue(name);
        inputText2.setSubmittedValue(null);
        inputText2.setValue(psw);
      }
    }
    ```

What Happens at Runtime: How the Launching Component Handles the ReturnEvent

At runtime in the sample application, when ADF Faces delivers a `ReturnEvent` to the `ReturnListener` registered on the `link` component, the `handleReturn()` method is called and the return value is processed accordingly. The new user is added to a customer list, and as a convenience to the user any previously submitted values in the Login page are cleared and the input fields are populated with the new information.

Part V

Creating a Databound Web User Interface

This part describes how to design the user interface of the Fusion web application using ADF Faces components.

Part V contains the following chapters:

29

Getting Started with Your Web Interface

This chapter describes how to get started with ADF data controls to create databound UI components, using ADF Faces components in the Fusion web application. It describes how to use page templates and page fragments to build a page. It also describes how to use managed beans to store logic for the page.
This chapter includes the following sections:

About Developing a Web Application with ADF Faces

ADF Faces is a set of Ajax-enabled JavaServer Faces (JSF) components that can be used to develop web applications with a minimal amount of hand-coded Javascript. You can use ADF Model data binding during the development of the web application that provides additional functionality.

Most of what you need to know to get started with your web interface is covered in Introduction to ADF Faces in *Developing Web User Interfaces with Oracle ADF Faces*. However, using the ADF Model layer for data binding instead of JSF managed beans provides additional functionality, such as the ability to declaratively bind components to your business services. For information on what ADF Model can provide, see Using ADF Model in a Fusion Web Application.

This chapter provides a high-level overview of the web interface development process as detailed in the Faces guide, and also provides information about the additional functionality available when you use ADF Model data binding.

Following the development process outlined in Introduction to Building Fusion Web Applications with Oracle ADF, developing a web application with ADF Faces and using ADF Model for data binding involves the following steps:

- Creating ADF Faces templates for your pages (optional)
- Creating the individual pages and page fragments for regions to be used within a page
- Creating any needed managed beans

Additionally, the lifecycle of a Fusion web application is different from that of a standard JSF or ADF Faces application. For information about how the lifecycle works, see Understanding the Fusion Page Lifecycle .

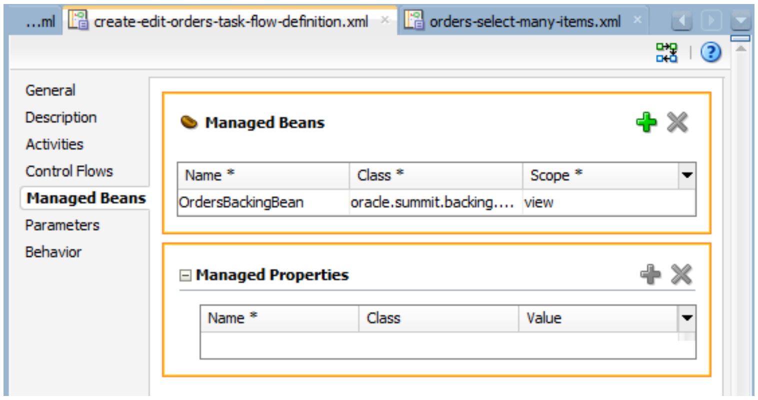
Managed beans are Java classes that give you the flexibility to add UI level code to your pages and task flows. You can use them for functions such as front end data manipulation and event handling.

ADF Faces page templates provides structure, consistency, and reusability for building web pages. You can create a page template and apply it to several pages for a consistent look and feel. It saves you time and effort because you do not need to layout the same elements each time you create a page.

Page Template and Managed Beans Use Cases and Examples

In the `create-edit-orders-task-flow-definition`, the `ShuttleBean` is used for code to support a shuttle component to shuttle the available products from the leading list to the selected products in the trailing list. The managed bean contains the code necessary to populate the available products list and the selected products list. Figure 29-1 shows the managed beans used in the `create-edit-orders-task-flow-definition`.

Figure 29-1   Managed Beans in the create-edit-orders-task-flow-definition



Managed Beans

Name *	Class *	Scope *
OrdersBackingBean	oracle.summit.backing....	view

Managed Properties

Name *	Class	Value

Additional Functionality for Page Templates and Managed Beans

You may find it helpful to understand other **Oracle ADF** features before you configure or use the ADF Model layer. Additionally, you may want to read about what you can do with your model layer configurations. Following are links to other functionality that may be of interest.

- For more information about using page templates, see Using ADF Model in a Fusion Web Application, and the How to Create a Page Template section of *Developing Web User Interfaces with Oracle ADF Faces*.
- For more information about managed beans in a standard JSF application, see the Java EE tutorial on the Oracle Technology Network website (`http://www.oracle.com/technetwork/java/javaee/overview/index.html`). For more information about managed beans and scope values, see About Object Scope Lifecycles.

Using Page Templates

ADF Page templates let you define entire page layouts, including values for certain attributes of the page. You can use any of the quick layout designs or you can create your own layout for a template.

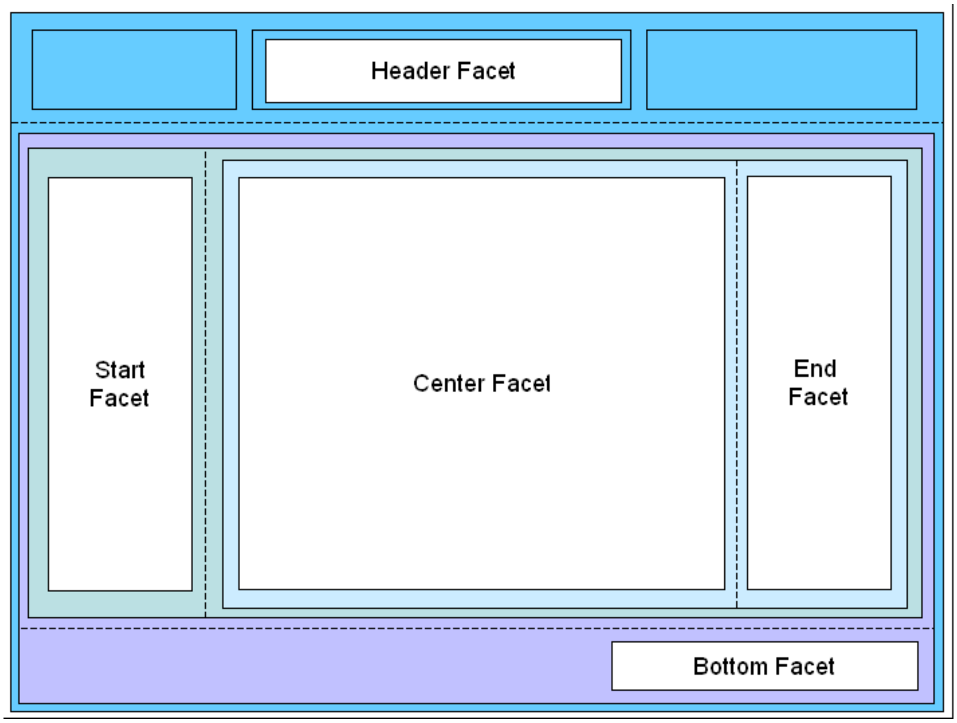
As you design the flow of your application, you can begin to think about the design of your pages. To ensure consistency throughout your application, you use ADF page templates. These page templates provide structure and consistency for other developers building web pages. Page templates typically contain static areas which cannot be changed when they are used, and dynamic areas, where developers can place content specific to the page they are building.

For example, a page template can provide a top area for branding and navigation, a bottom area for copyright information, and a center area for the main content of the page. Page developers do not need to do anything to the branding and copyright information when they use the template. They need only to develop the main content area.

In addition to using ADF Faces components to build a page template, you can add attributes to the template definition. These attributes provide placeholders for specific information that the page developer needs to provide when using the page template. For instance, if you have added an attribute for a welcome message and when page developers use this page template, they can add a different welcome message for each page.

You can also add facet references to the page template. These references act as placeholders for content on the page. Figure 29-2 shows a rendition of how facets can be used in a page template.

Figure 29-2 Facets in the Page Template



In this page template, facet references are used inside four different `panelSplitter` components. When the home page was created using this page template, the navigational links were placed in the Header facet and the accordion panels that hold the navigation trees and search panels were placed in the Start facet. The cart summary was placed in the End facet, and the main portion of the page was placed in the Center facet. The copyright information was placed in the Bottom facet.

When you choose to add databound components to a page template, an associated page definition file and the other metadata files that support ADF Model layer data binding are created. Each component is bound in the same fashion as for standard JSF pages, as described in Using ADF Model in a Fusion Web Application. You can also create model parameters for the page template. The values for these parameters can then be set at runtime by the calling page.

For example, if you wanted a list of products to appear on each page that uses the page template, you could drag and drop the `Name` attribute of the `ProductVO` collection as a list. Or, if you wanted the pages to display the currently selected product Id, you could create a model parameter for the page template that would evaluate to that product's Id.

> **Note:**
>
> Page templates are primarily a project artifact. While they can be reused between projects and applications, they are not fully self-contained and will always have some dependencies to external resources, for example, any ADF data binding, `Strings` from a message bundle, images, and managed beans.

If a page template does not contain databound components, it can be referenced dynamically by the calling page using an EL expression. That is, the page template to be used can be determined at runtime. For instance, a page may use templateA or templateB based on user selection. When you add a page template to a page, an `af:pageTemplate` tag is added to the page. The `af:pageTemplate` tag includes a `viewId` attribute that specifies the page template the page will use. You can set `viewId` with an EL expression to a managed bean method that returns the page template `Id`, as shown in the following example.

```
<af:pageTemplate
    id="pt1"
    viewId="#{myBean.templateViewId}"
```

If the page template has databound components, setting the `viewId` with an EL expression is not enough. Because databound components require access to the binding container, you must specify the page template as well as its associated binding container.

For databound page templates, you use the `pageTemplateModel` to manage both the page template `Id` and the associated binding container. In the JSF page, instead of using the `viewId` attribute, you set the `value` attribute to the `pageTemplateModel`. You must also modify the page executable section of the calling page's page definition file and create a managed bean with methods to process the page template `Ids`. For detailed instructions, see How to Add a Databound Page Template to a Page Dynamically.

How to Use ADF Data Binding in ADF Page Templates

Creating a page template for use in an application that uses **ADF Business Components** and ADF Model layer data binding is the same as creating a standard ADF Faces page template, as documented in the Using Page Templates section of *Developing Web User Interfaces with Oracle ADF Faces*. Once you create the template, you can drag and drop items from the Data Controls panel. JDeveloper automatically adds the page definition file when you drag and drop items from this panel.

The Create a Page Template wizard also allows you to create model parameters for use by the template.

Before you begin:

It may help to understand the options that are available to you when you create a basic page template. For more information, see Using Page Templates.

You may also find it useful to understand additional functionality that can be used with page templates. For more information, see Additional Functionality for Page Templates and Managed Beans.

You will need to complete this task:

> Create a page template that uses ADF Business Components and ADF Model layer data binding by following the instructions in the How to Create a Page Template section of *Developing Web User Interfaces with Oracle ADF Faces*. However, do not complete the dialog

To add model parameters to a template:

1. In the Data Binding page of the Create a Page Template wizard, select **Create Page Definition File**.

   > **Note:**
   >
   > You only need to select this checkbox if you wish to add parameters. JDeveloper automatically adds the page definition file when you drag and drop items from the Data Controls panel.

2. Click the **Add** icon.
3. Enter the following for the parameter:
   - **Id**: Enter a name for the parameter.
   - **Value**: Enter a value or an EL expression that can resolve to a value. If needed, click the Invoke Expression Builder (...) button to open the Expression Builder. You can use this to build the expression. For more information about EL expressions and the EL expression builder, see Creating ADF Data Binding EL Expressions.
   - **Option**: Select the option that determines how the parameter value will be specified.
     - **optional**: The binding definition's value is used only if the parameter is not specifically set by the caller. This is the default.

   - **final**: The binding definition has the expression to access the value that should be used for this parameter.
   - **mandatory**: The parameter value has to be set by the caller.
- **Read Only**: Select if the parameter's value is to be read-only and should not be overwritten

4. Create more parameters as needed. Use the order buttons to arrange the parameters into the order in which you want them to be evaluated.

You can now use the Data Controls panel to add databound UI components to the page, as you would for a standard JSF page, as described in the remaining chapters in this part of the book.

> **Note:**
>
> If your template contains any method actions bound to a method iterator, you cannot change the value of the `refresh` attribute on the iterator to anything other than `Default`. If set to anything other than `Default`, the method will not execute.

What Happens When You Use ADF Model Layer Bindings on a Page Template

When you add ADF databound components to a template or create model parameters for a template, a page definition file is created for the template, and any model parameters are added to that file.

> **Note:**
>
> This section describes what happens for a statically assigned page template. For information about dynamic templates, see How to Add a Databound Page Template to a Page Dynamically.

The following example shows what the page definition file for a template for which you created a `productId` model parameter might look like.

```
<parameters>
  <parameter id="productID" readonly="true"
             value="#{bindings.productId.inputValue}"/>
</parameters>
<executables/>
<bindings/>
```

Parameter binding objects declare the parameters that the page evaluates at the beginning of a request. For more information about binding objects and the ADF lifecycle, see Using ADF Model in a Fusion Web Application. However, since a template itself is never executed, the page that uses the template (the *calling* page) must access the binding objects created for the template (including parameters or any other type of binding objects created by dragging and dropping objects from the Data Controls panel onto the template).

In order to access the template's binding objects, the page definition file for the calling page must instantiate the binding container represented by the template's page definition file. As a result, a reference to the template's page definition is inserted as an executable into the calling page's page definition file, as shown in the following example.

```
<executables>
  <page path="oracle.summt.view.pageDefs.MyTemplatePageDef"
        id="pageTemplateBinding"/>
</executables>
```

In this example, the calling page was built using the `MyTemplate` template. Because the page definition file for the `MyTemplate` template appears as an executable for the calling page, when the calling page's binding container is instantiated, it will in turn instantiate the `MyTemplatePageDef`'s binding container, thus allowing access to the parameter values or any other databound values.

Because there is an ID for this reference (in this case, `pageTemplateBinding`), the calling page can have components that are able to access values from the template. When you create a JSF page from a template, instead of you having to repeat the code contained within the template, you can use the `af:pageTemplate` tag on the calling page. This tag contains the path to the template JSF page.

Additionally, when the template contains any ADF data binding, the value of that tag is the ID set for the calling page's reference to the template's page definition, as shown in the following example.

```
<af:pageTemplate viewId="/MyTemplate.jspx"
                 value="#{bindings.pageTemplateBinding}".../>
```

How to Add a Databound Page Template to a Page Dynamically

You can dynamically add a page template without databound components by using an EL expression to select the page template. For more information on how to do this, see Using Page Templates.

You can also statically add a page template with databound components. For more information on how to do this, see How to Use ADF Data Binding in ADF Page Templates.

This section describes how to dynamically add a page template with databound components to a page. For more information, see Using Page Templates.

You use the `pageTemplateModel` to dynamically manage a page template and its binding container. You use an EL expression in the page definition file to set the page template `Id`. You create managed bean methods to return the page template `Id`.

Before you begin:

It may help to understand the options that are available to you when you create a basic page template. For more information, see Using Page Templates.

You may also find it useful to understand additional functionality that can be used with page templates. For more information, see Additional Functionality for Page Templates and Managed Beans.

You will need to complete this task:

> Create a page template that uses ADF Business Components and ADF Model layer data binding as described in How to Use ADF Data Binding in ADF Page Templates.

To add a databound page template to a page dynamically:

1. In the JSF page source editor, remove the `viewId` attribute and change the `value` attribute to the `pageTemplateModel`. You do not need to create a `pageTemplateModel` explicitly. You can use the `pageTemplateModel` from the corresponding page definition file's page executable binding. For example, for a page executable binding called `pageTemplate1`, you would add the following line under the `af:pageTemplate` tag:

   ```
   value="#{bindings.pageTemplate1.templateModel}"/>
   ```

2. In the page definition file `<executable>` section, make the following changes to the `<page>` section:

   - Remove the `path` attribute. It is no longer needed. The `pageTemplateModel` manages databound components' access to the binding container.
   - Change the `id` attribute to the page executable binding. In this example, it is `pageTemplate1`.
   - Add a `Refresh` attribute and set it to `ifNeeded`.
   - Add a `viewId` attribute and set it to an EL expression with a managed bean method that returns the current page template `Id`.

   For example, for a page executable binding of `pageTemplate1`, the `id` attribute would also be `pageTemplate1`:

   ```
   <executables>
     <page id="pageTemplate1"
           viewId="#{myBean.templateViewId}"
           Refresh="ifNeeded"/>
     ...
   </executables>
   ```

3. Create a `pageFlowScope` managed bean with a method that returns the current page template `Id`.

   The managed bean code should be similar to that in the following example. In this example, `gettemplateViewId()` obtains the user's page template selection and returns the page template `Id`. `setMDTemplateViewId()` sets the page template to be `MDPageTemplate` and `setPopupTemplateViewId()` sets the page template to be `PopupPageTemplate`.

   ```
   public class myClass {
       final private String MDPageTemplate = "/MDPageTemplate.jspx";
       final private String PopupPageTemplate = "/PopupPageTemplate.jspx";
       private String templateViewId;

       public myClass() {
           super();
           templateViewId = MDPageTemplate;
       }
       public String gettemplateViewId() {
           return templateViewId;
       }
       public void setMDTemplateViewId(ActionEvent ae) {
   ```

```
            templateViewId = MDPageTemplate;
        }
        public void setPopupTemplateViewId(ActionEvent ae) {
            templateViewId = PopupPageTemplate;
        }
    }
```

What Happens at Runtime: How Pages Use Templates

When a page is created using a template that contains ADF data binding, like a page without a template, the page definition file is used to create the page. During the lifecycle, the binding container for page template is also created and the UI components and biding for both the page and template are processed.

Specifically, the following happens:

1. The calling page follows the standard JSF/ADF lifecycle, as documented in Understanding the Fusion Page Lifecycle. As the page enters the Restore View phase, the URL for the calling page is sent to the binding context, which finds the corresponding page definition file.
2. During the Initialize Context phase, the binding container for the calling page is created based on the page definition file.
3. During the Prepare Model phase, the page template executable is refreshed. At this point, the binding container for the template is created based on the template's page definition file, and added to the binding context.
4. The lifecycle continues, with UI components and bindings from both the page and the template being processed.

Creating a Web Page

During the development of a Fusion web application, web pages are built using page templates and page fragments. You can create a web page using the Create JSF Page dialog and you can also secure web pages within the ADF Security framework.

You can create pages either by double-clicking a view activity in a task flow or by using the New Gallery. When creating the page (or dropping a view activity onto a task flow), you can choose to create the page as a JSF JSP or as a JSF JSP fragment. JSF fragments provide a simple way to create reusable page content in a project, and are what you use when you wish to use task flows as regions on a page. When you modify a JSF page fragment, the JSF pages that consume the page fragment are automatically updated.

> **Note:**
>
> Although JDeveloper supports XHTML files to be used in applications that use Facelets, the `faces-config.xml` and `adfc-config.xml` diagrammers do not support XHTML. In order to add navigation to these files, you have to manually edit the code by clicking the **Source** tab.

When you begin adding content to your page, you typically use the Components window and Data Controls panel of JDeveloper. The Components window contains all the ADF Faces components needed to declaratively design your page. Once you have the basic layout components placed on the page, you can then drag and drop items from the Data Controls panel to create ADF Model databound UI components. The remaining chapters in this part of the book explain in detail the different types of databound components and pages you can create using ADF Model data binding.

How to Create a Web Page

You create web pages using the Create JSF Page dialog.

Before you begin:

It may be helpful to have an understanding of the different options when creating a page. For more information, see Creating a Web Page.

To create a web page:

1. In the Applications window, right-click the node (directory) where you would like the page to be saved, and choose **New > Page**.
2. Complete the Create JSF Page dialog. For help, click **Help** in the dialog.

For more information on the templates available and what happens when you create the page, see Creating a View Page section in *Developing Web User Interfaces with Oracle ADF Faces*.

What You May Need to Know About Creating Blank Web Pages

When you create a new blank web page, the generated code might differ depending on the type of project you are working in and the tag libraries that have been imported in the project. For example, if you are working in a project created from the Custom Project template and you select the **Create Blank Page** option in the Create JSF page dialog, the generated code might not include ADF Faces tags as it would under similar circumstances in the UI project of an ADF Fusion Web Application workspace.

If you want to create a page without a pre-defined layout that includes ADF Faces tags, it might be easiest to select the **Copy Quick Start Layout** option in the Create JSF page dialog and select a simple layout. Selecting this option will trigger the importing of the ADF Faces tag library into the project. In addition, any subsequent pages that you create in the project, even those created with the **Create Blank Page** option selected, will use ADF Faces tags.

What You May Need to Know About Securing a Web Page

When creating web pages, you should have a plan for how you will apply security to them.

To secure a web page within the ADF Security framework, you actually secure the page's page definition file. If the page is within a bounded task flow, you secure the bounded task flow (and the security policies applied to that bounded task flow apply to all of the pages within the task flow). If a page is within a bounded task flow, you should *not* apply security policies to the page definition file, as that could enable a user to access the page directly.

For more information on securing web pages, see Defining ADF Security Policies.

Using a Managed Bean in a Fusion Web Application

A standard JSF application includes one or more managed beans, each of which can be associated with the components used in a particular page. Managed Beans must be registered in the configuration files of JSF application or Fusion web application.

Managed beans are Java classes that you register with the application using various configuration files. When the JSF application starts up, it parses these configuration files, and the beans listed within them are made available. The managed beans can be referenced in an EL expression, allowing access to the beans' properties and methods. Whenever a managed bean is referenced for the first time and it does not already exist, the Managed Bean Creation Facility instantiates the bean by calling the default constructor method on it. If any properties are also declared, they are populated with the declared default values.

Often, managed beans handle events or some manipulation of data that is best handled at the front end. For a more complete description of how managed beans are used in a standard JSF application, see the Java EE tutorial on the Oracle Technology Network website (`http://www.oracle.com/technetwork/java/javaee/overview/index.html`).

> **Best Practice:**
>
> Use managed beans to store logic that is related to the UI rendering only. All application data and processing should be handled by logic in the business layer of the application. Similar to how you store data-related logic in the database using PL/SQL rather than a Java class, the rule of thumb in a Fusion web application is to store business-related logic in the middle tier. This way, you can expose this logic as business service methods, which can then become accessible to the ADF Model layer and be available for data binding.

In an application that uses ADF data binding and ADF task flows, managed beans are registered in different configuration files from those used for a standard JSF application. In a standard JSF application, managed beans are registered in the `faces-config.xml` configuration file. In a Fusion web application, managed beans can be registered in the `faces-config.xml` file, the `adfc-config.xml` file, or a task flow definition file. Which configuration file you use to register a managed bean depends on what will need to access that bean, whether or not it needs to be customized at runtime, what the bean's scope is, and in what order all the beans in the application need to be instantiated. Table 29-1 describes how registering a bean in each type of configuration file affects the bean.

> **Note:**
>
> Fusion web applications take advantage of the functionality provided by ADF task flows. Managed beans accessed within the task flow definition must be registered in that task flow's definition file and not in the `faces-config.xml` file.

Table 29-1 Effects of Managed Bean Configuration Placement

Managed Bean Placement	Effect
`adfc-config.xml`	• Managed beans can be of any scope. However, any backing beans for page fragments or declarative components should use `BackingBean` scope. For information regarding scope, see About Object Scope Lifecycles. • When executing within an unbounded task flow, `faces-config.xml` will be checked for managed bean definitions before the `adfc-config.xml` file. • Lookup precedence is enforced per scope. Request-scoped managed beans take precedence over session-scoped managed beans. Therefore, a request-scoped managed bean named `foo` in the `adfc-config.xml` file will take precedence over a session-scoped managed bean named `foo` in the current task flow definition file. • Already instantiated beans take precedence over new instances being instantiated. Therefore, an existing session-scoped managed bean named `foo` will always take precedence over a request-scoped bean named `foo` defined in the current task flow definition file.
Task flow definition file	• Managed bean can be of any scope. However, managed beans of `pageFlow` scope or `view` scope that are to be accessed within the task flow definition must be defined within the task flow definition file. Any backing beans for page fragments in a task flow should use `BackingBean` scope. • Managed bean definitions within task flow definition files will be visible only to activities executing within the same task flow. • When executing within a bounded task flow, `faces-config.xml` will be checked for managed bean definitions before the currently executing task flow definition. If no match is found in either location, `adfc-config.xml` and other bootstrap configuration files will be consulted. However, this lookup in other `adfc-config.xml` and bootstrap configuration files will only occur for session- or application-scoped managed beans. • Lookup precedence is enforced per scope. Request-scoped managed beans take precedence over session-scoped managed beans. Therefore, a request-scoped managed bean named `foo` in the `adfc-config.xml` file will take precedence over a session-scoped managed bean named `foo` in the current task flow definition file. • Already instantiated beans take precedence over new instances being instantiated. Therefore, an existing session-scoped managed bean named `foo` will always take precedence over a request-scoped bean named `foo` registered in the current task flow definition file. • Customizations are allowed.

Table 29-1 (Cont.) Effects of Managed Bean Configuration Placement

Managed Bean Placement	Effect
faces-config.xml	• Managed beans can be of any scope other than pageFlow scope or view scope. • When searching for any managed bean, the faces-config.xml file is always consulted first. Other configuration files will be searched only if a match is not found. Therefore, beans registered in the faces-config.xml file will always win any naming conflict resolution. • No customizations can be made.

As a general rule for Fusion web applications, a bean that may be used in more than one page or task flow, or one that is used by pages within the main unbounded task flow (adfc-config), should be registered in the adfc-config.xml configuration file. A managed bean that will be used only by a specific task flow should be registered in that task flow's definition file. There should be no beans registered in the faces-config.xml file.

Note:

If you create managed beans from dialogs within JDeveloper, the bean is registered in the adfc-config.xml file, if it exists.

For example, in the Summit sample application for Oracle ADF, the ShuttleBean is a managed bean used by the showshuttle page to handle the selections in the shuttle component on that page. Because it is used solely within the create-edit-orders-task-flow-definition task flow, it is registered in the create-edit-orders-task-flow-definition definition file.

You can create a managed bean for use within a task flow (either the default adfc-config flow or any bounded task flow). For more information regarding managed beans and how they are used as backing beans for JSF pages, see Creating and Using Managed Beans in *Developing Web User Interfaces with Oracle ADF Faces*.

How to Use a Managed Bean to Store Information

Within the editors for a task flow definition, you can create a managed bean and register it with the JSF application at the same time.

Before you begin:

It may help to understand the options that are available to you when you create a managed bean. For more information, see Using a Managed Bean in a Fusion Web Application.

You may also find it useful to understand additional functionality that can be used with managed beans. For more information, see Additional Functionality for Page Templates and Managed Beans.

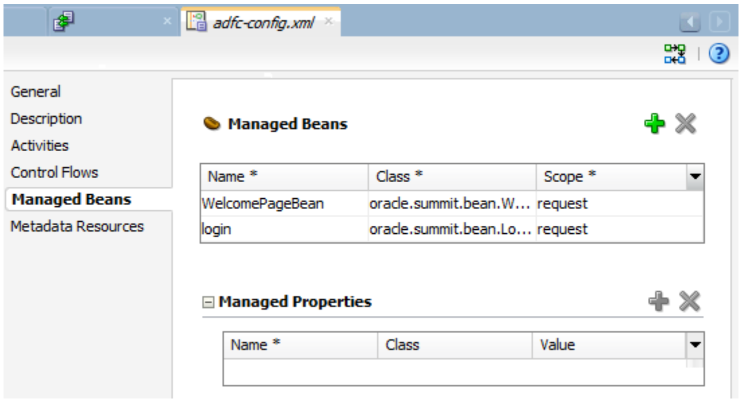
You will need to complete this task:

Create the configuration file (if it doesn't already exist) that you want the managed bean to be associated with. This can be `faces-config.xml`, `adfc-config.xml`, or a bounded task flow definition file.

To create a managed bean for adfc-config.xml or a task flow:

1. In the Applications window, double-click either the **adfc-config.xml** file or any other task flow definition file.
2. In the overview editor, click the **Managed Beans** navigation tab. Figure 29-3 shows the editor for the `adfc-config.xml` file.

Figure 29-3 Managed Beans in the adfc-config.xml File



Name *	Class *	Scope *
WelcomePageBean	oracle.summit.bean.W...	request
login	oracle.summit.bean.Lo...	request

Name *	Class	Value

3. In the Managed Bean page, click the **Add** icon to add a row to the **Managed Beans** table.
4. In the fields, enter the following:
   - **Name**: A name for the bean.
   - **Class**: If the corresponding class has already been created for the bean, use the browse (...) button for the **managed-bean-class** field to search for and select the class. If a class does not exist, enter the name you'd like to use. Be sure to include any package names as well. You can then use the drop-down menu to choose **Generate Class**, and the Java file will be created for you.
   - **Scope**: The bean's scope. For more information about the different object scopes, see About Object Scope Lifecycles.

> **Note:**
>
> When determining what scope to register a managed bean with or to store a value in, keep the following in mind:
>
> – Always try to use the narrowest scope possible.
>
> – If your managed bean takes part in component binding by accepting and returning component instances (that is, if UI components on the page use the binding attribute to bind to component properties on the bean), then the managed bean must be stored in `BackingBean` scope. If it can't be stored in one of those scopes (for example, if it needs to be stored in `sessionScope` for high availability reasons), then instead of using component binding, you need to use the ComponentReference API. For more information, see the What You May Need to Know About Component Bindings and Managed Beans section of *Developing Web User Interfaces with Oracle ADF Faces*
>
> – Use the `sessionScope` scope only for information that is relevant to the whole session, such as user or context information. Avoid using the `sessionScope` scope to pass values from one page to another.
>
> – You can also set the scope to `none`. While not technically a scope, `none` means that the bean will not live within any particular scope, but will instead be instantiated each time it is referenced. You should set a bean's scope to `none` when it is referenced by another bean.

5. You can optionally add needed properties for the bean. With the bean selected in the **Managed Beans** table, click the **Add** icon for the **Managed Properties** table. Enter a property name (other fields are optional).

> **Note:**
>
> While you can declare managed properties using this editor, the corresponding code is not generated on the Java class. You will need to add that code by creating private member fields of the appropriate type and then using the **Generate Accessors** menu item on the context menu of the source editor to generate the corresponding getter and setter methods for these bean properties.

What Happens When You Create a Managed Bean

When you use the configuration editor to create a managed bean and elect to generate the Java file, JDeveloper creates a stub class with the given name and a default constructor. The following example shows the code added to the `MyBean` class stored in the view package.

```
package view;
```

```
public class MyBean {
    public MyBean() {
    }
}
```

You now need to add the logic required by your task flow or page. You can then refer to that logic using an EL expression that refers to the `managed-bean-name` value given to the managed bean. For example, to access the `myInfo` property on the bean, the EL expression would be:

```
#{my_bean.myInfo}
```

JDeveloper also adds a `managed-bean` element to the appropriate task definition file. The following shows the `managed-bean` element created for the `MyBean` class.

```
<managed-bean>
  <managed-bean-name>my_bean</managed-bean-name>
  <managed-bean-class>view.MyBean</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

How to Set Managed Bean Memory Scopes in a Server-Cluster Environment

Typically, in an application that runs in a clustered environment, a portion of the application's state is serialized and copied to another server or a data store at the end of each request so that the state is available to other servers in the cluster.

> **Note:**
>
> If the managed bean will be calling `set` and `get` methods on ADF Faces components, you cannot serialize the managed beans because ADF Faces components are not serializable. You will need to access the ADF Faces components in another way.

When you are designing an application to run in a clustered environment, you must:

- Ensure that all managed beans with a lifespan longer than one request are serializable (that is, they implement the `java.io.Serializable` interface). Specifically, beans stored in session scope, page flow scope, and view scope need to be serializable.

  > **Tip:**
  >
  > To identify failures with objects stored in page flow scope and view scope, use `writeObject()`. This method provides additional information in an exception about the object and scope that failed to serialize. Additional information might be a region's page flow scope and the key of the object.

- Make sure that the framework is aware of changes to managed beans stored in ADF scopes (view scope and page flow scope).

When a value within a managed bean in either view scope or page flow scope is modified, the application needs to notify the framework so that it can ensure that the bean's new value is replicated.

In this example, an attribute of an object in view scope is modified.

```
Map<String, Object> viewScope =
    AdfFacesContext.getCurrentInstance().getViewScope();
MyObject obj = (MyObject)viewScope.get("myObjectName");
Obj.setFoo("newValue");
```

Without additional code, the framework will be unaware of this change and it will not know that a new value needs to be replicated within the cluster. To inform the framework that an object in an ADF scope has been modified and that replication is needed, use the `markScopeDirty()` method, as shown in the following example. The `markScopeDirty()` method accepts only `viewScope` and `pageFlowScope` as parameters.

```
    ControllerContext ctx = ControllerContext.getInstance();
    ctx.markScopeDirty(viewScope);
```

This code is needed for any request that modifies an existing object in one of the ADF scopes. If the scope itself is modified by the scope's `put()`, `remove()`, or `clear()` methods, it is not necessary to notify the framework.

If an application is not deployed to a clustered environment, the tracking of changes to ADF memory scopes is not needed, and by default, this functionality is disabled. To enable ADF Controller to track changes to ADF memory scopes and replicate the page flow scope and view scope within the server cluster, set the `<adf-scope-ha-support>` parameter in the `adf-config.xml` file to `true`. Because scope replication has a small performance overhead, it should be enabled only for applications running in a server-cluster environment.

The folllowing shows `adf-scope-ha-support` set to `true` in the `adf-config.xml` file.

```
<?xml version="1.0" encoding="US-ASCII" ?>
<adf-config xmlns="http://xmlns.oracle.com/adf/config"
      xmlns:adfc="http://xmlns.oracle.com/adf/controller/config">
   <adfc:adf-controller-config>
   ...
    <adfc:adf-scope-ha-support>true</adfc:adf-scope-ha-support>
   ...
  </adfc:adf-controller-config>
   ...
</adf-config>
```

30
Understanding the Fusion Page Lifecycle

This chapter describes the ADF page lifecycle, its phases, and how to best use the lifecycle within a Fusion web application.
This chapter includes the following sections:

About the Fusion Page Lifecycle

The Fusion page lifecycle provides support for a web page to interact with ADF Model data bindings. You can learn about the sequence of events that occur while processing a web page request using JSF and Oracle ADF.

When a JSF page is submitted and a new page is requested, the JSF page request lifecycle is invoked. This lifecycle handles the submission of values on the page, validation for components on the current page, navigation to and display of the components on the resulting page, as well as saving and restoring state. The JSF lifecycle phases use a UI component tree to manage the display of the components. This tree is a runtime representation of a JSF page: each UI component tag in a page corresponds to a UI component instance in the tree.

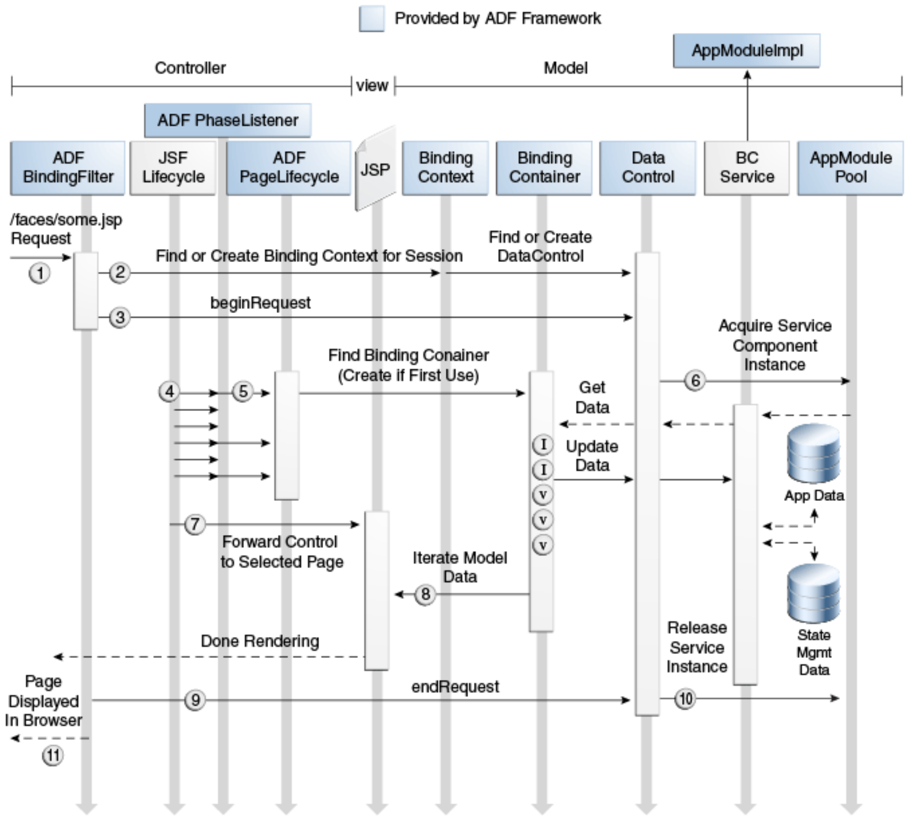
The `FacesServlet` object manages the page request lifecycle in JSF applications. The `FacesServlet` object creates an object called `FacesContext`, which contains the information necessary for request processing, and invokes an object that executes the lifecycle.

In addition to the JSF lifecycle, the ADF Faces and ADF page lifecycle are also run. The ADF Faces lifecycle extends the JSF lifecycle, providing additional functionality, such as a client-side value lifecycle, a subform component that allows you to create independent submittable sections on a page without the drawbacks (for example, lost user edits) of using multiple forms on a single page, and additional scopes. For information about both the JSF lifecycle and the ADF Faces lifecycle, including what you can do with the lifecycle for your application, see Using the JSF Lifecycle with ADF Faces.

The ADF page lifecycle handles preparing and updating the data model, validating the data at the model layer, and executing methods on the business layer. The ADF page lifecycle uses the binding container to make data available for easy referencing by the page during the current page request.

The combined JSF and ADF page lifecycle is only one sequence within a larger sequence of events that begins when an HTTP request arrives at the application server and continues until the page is returned to the client. This overall sequence of events can be called the **web page lifecycle**. It follows processing through the model, view, and controller layers as defined by the MVC architecture. The page lifecycle is not a rigidly defined set of events, but is rather a set of events for a typical use case. Figure 30-1 shows a sequence diagram of the lifecycle of a web page request using JSF and **Oracle ADF** in tandem.

Figure 30-1 Lifecycle of a Web Page Request Using JSF and Oracle ADF



The basic flow of processing a web page request using JSF and Oracle ADF happens as follows:

1. A web request for `http://yourserver/yourapp/faces/some.jsp` arrives from the client to the application server.
2. The `ADFBindingFilter` object looks for the ADF binding context in the HTTP session, and if it is not yet present, initializes it for the first time. Some of the functions of the `ADFBindingFilter` include finding the name of the binding context metadata file, and finding and constructing an instance of each data control.
3. The `ADFBindingFilter` object invokes the `beginRequest()` method on each data control participating in the request. This method gives the data control a notification at the start of every request so that it can perform any necessary setup.
4. The JSF `Lifecycle` object, which is responsible for orchestrating the standard processing phases of each request, notifies the `ADFPhaseListener` class during each phase of the lifecycle, so that it can perform custom processing to coordinate the JSF lifecycle with the ADF Model data binding layer. For information about the details of the JSF and ADF page lifecycle phases, see About the JSF and ADF Page Lifecycles.

> **Note:**
>
> The FacesServlet class (in javax.faces.webapp), configured in the web.xml file of a JSF application, is responsible for initially creating the JSF Lifecycle class (in javax.faces.lifecycle) to handle each request. However, since it is the Lifecycle class that does all the interesting work, the FacesServlet class is not shown in the diagram.

5. The ADFPhaseListener object creates an ADF PageLifecycle object to handle each request and delegates the appropriate before and after phase methods to corresponding methods in the ADF PageLifecycle class. If the binding container for the page has never been used before during the user's session, it is created.
6. The first time an **application module data control** is referenced during the request, it acquires an instance of the application module from the **application module pool**.
7. The JSF Lifecycle object forwards control to the page to be rendered.
8. The UI components on the page access value bindings and iterator bindings in the page's binding container and render the formatted output to appear in the browser.
9. The ADFBindingFilter object invokes the endRequest() method on each data control participating in the request. This method gives a data control notification at the end of every request, so that they can perform any necessary resource cleanup.
10. An application module data control uses the endRequest notification to release the instance of the application module back to the application module pool.
11. The user sees the resulting page in the browser.

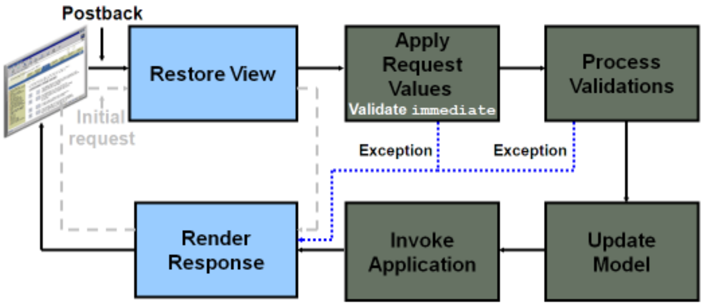
The ADF page lifecycle also contains phases that are defined simply to notify ADF page lifecycle listeners before and after the corresponding JSF phase is executed (that is, there is no implementation for these phases). These phases allow you to create custom listeners and register them with any phase of both the JSF and ADF page lifecycles, so that you can customize the ADF page lifecycle if needed, both globally or at the page level.

About the JSF and ADF Page Lifecycles

The JSF and ADF page lifecycles perform a sequence of events that begins when an HTTP request arrives at the application server and continues until the page is returned to the client. You can learn about the different phases of a page lifecycle in a JSF application that uses an ADF Model.

Figure 30-2 shows a high-level view of the combined JSF and ADF page lifecycles.

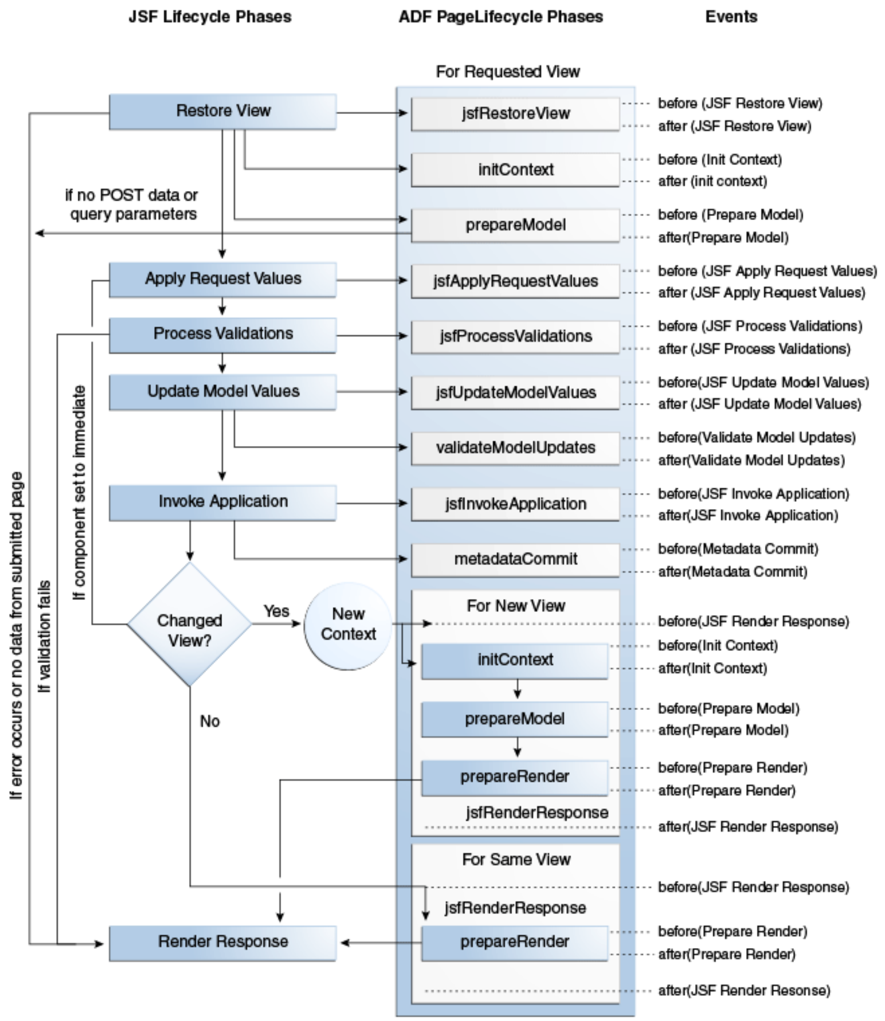
Figure 30-2 JSF and ADF Page Lifecycles



Say for example, you have a page with some text displayed in an input text component and a button. When that page is first rendered, the component tree is built during the Restore View phase and then the lifecycle goes straight to the Render Response phase, as the components are rendered. When the user clicks the button, the full lifecycle is invoked. The component tree is rebuilt, the input text component extracts any new value during the Apply Request Values phase and Process Validations phase, and if there is an error, for example due to validation, then the lifecycle jumps to the Render Response phase. Otherwise, the model is then updated with the new value during the Update Model phase, and then any application processing associated with the button (such as navigation), is executed during the Invoke Application phase.

Figure 30-3 shows the details of how the JSF and ADF Faces and ADF Model phases integrate in the lifecycle of a page request.

Figure 30-3 Lifecycle of a Page Request in a Fusion Web Application



In a JSF application that uses ADF Model, the phases in the page lifecycle are as follows:

- Restore View: The URL for the requested page is passed to the `bindingContext` object, which finds the page definition file that matches the URL. The component tree of the requested page is either newly built or restored. All the component tags, event handlers, converters, and validators on the submitted page have access to the `FacesContext` instance. If the component tree is empty, (that is, there is no data from the submitted page), the page lifecycle proceeds directly to the Render Response phase.

  If any discrepancies between the request state and the server-side state are detected, an error will is thrown and the page lifecycle jumps to the Render Response phase.

- JSF Restore View: Provides before and after phase events for the Restore View phase. You can create a listener and register it with the before or after event of this phase, and the application will behave as if the listener were registered with the Restore View phase. The Initialize Context phase of the ADF Model page lifecycle listens for the `after(JSF Restore View)` event and then executes. ADF Controller uses listeners for the before and after events of this phase to synchronize the server-side state with the request. For example, it is in this phase that browser back button detection and bookmark reference are handled. After the before and after listeners are executed, the page flow scope is available.

- Initialize Context: The page definition file is used to create the `bindingContainer` object, which is the runtime representation of the page definition file for the requested page. The `LifecycleContext` class used to persist information throughout the ADF page lifecycle phases is instantiated and initialized with values for the associated request, binding container, and lifecycle.
- Prepare Model: The ADF page lifecycle enters the Prepare Model phase by calling the `BindingContainer.refresh(PREPARE_MODEL)` method. During the Prepare Model phase, `BindingContainer` page parameters are prepared and then evaluated. If parameters for a task flow exist, they are passed into the flow.

  Next, any executables that have their refresh property set to `prepareModel` are refreshed based on the order of entry in the page definition file's `<executables>` section and on the evaluation of their `RefreshCondition` properties (if present). When an executable leads to an iterator binding refresh, the corresponding data control will be executed, and that leads to execution of one or more collections in the service objects. If an iterator binding fails to refresh, a JBO exception will be thrown and the data will not be available to display. For more information about the `RefreshCondition` property, see pageNamePageDef.xml.

  If the incoming request contains no `POST` data or query parameters, then the lifecycle forwards to the Render Response phase.

  If the page was created using a template, and that template contains bindings using ADF Model, the template's page definition file is used to create the binding container for the template. The container is then added to the binding context.

  If any `taskFlow` executable bindings exist (for example, if the page contains a region), the `taskFlow` binding creates an ADF Controller `ViewPortContext` object for the task flow, and any nested binding containers for pages in the flow are then executed.
- Apply Request Values: Each component in the tree extracts new values from the request parameters (using its decode method) and stores those values locally. Most associated events are queued for later processing. If you have set a component's `immediate` attribute to `true`, then the validation, conversion, and events associated with the component are processed during this phase and the lifecycle skips the Process Validations, Update Model Values, and Invoke Application phases. Additionally, any associated iterators are invoked. For more information about ADF Faces validation and conversion, see the Validating and Converting Input chapter in *Developing Web User Interfaces with Oracle ADF Faces*.
- JSF Apply Request Values: Provides before and after phase events for the Apply Request Values phase. You can create a listener and register it with the before or after event of this phase, and the application will behave as if the listener were registered with the Apply Request Values phase.
- Process Validations: Local values of components are converted and validated on the client. If there are errors, the lifecycle jumps to the Render Response phase. At the end of this phase, new component values are set, any validation or conversion error messages and events are queued on `FacesContext`, and any value change events are delivered. Exceptions are also caught by the binding container and cached.

- JSF Process Validations: Provides before and after phase events for the Process Validations phase. You can create a listener and register it with the before or after event of this phase, and the application will behave as if the listener were registered with the Process Validations phase.
- Update Model Values: The component's validated local values are moved to the model and the local copies are discarded. For any updateable components (such as an `inputText` component), corresponding iterators are refreshed, if the refresh condition is set to the default (`deferred`) and the refresh condition (if any) evaluates to `true`.

  If you are using a backing bean for a JSF page to manage your UI components, any UI attributes bound to a backing bean property will also be refreshed in this phase.
- JSF Update Model Values: Provides before and after phase events for the Update Model Values phase. You can create a listener and register it with the before or after event of this phase, and the application will behave as if the listener were registered with the Update Model Values phase.
- Validate Model Updates: The updated model is now validated against any validation routines set on the model. Exceptions are caught by the binding container and cached.
- Invoke Application: Any action bindings for command components or events are invoked.
- JSF Invoke Application: Provides before and after phase events for the Invoke Application phase. You can create a listener and register it with the before or after event of this phase, and the application will behave as if the listener were registered with the Invoke Application phase.
- Metadata Commit: Changes to runtime metadata are committed. This phase stores any runtime changes made to the application using the Metadata Service (MDS). For more information about using MDS to persist runtime changes, see Customizing Applications with MDS.
- Initialize Context (only if navigation occurred in the Invoke Application lifecycle): The Initialize Context phase listens for the `beforeJSFRenderResponse` event to execute. The page definition file for the next page is initialized.
- Prepare Model (only if navigation occurred in the Invoke Application lifecycle): Any page parameters contained in the next page's definition are set.
- Prepare Render: The binding container is refreshed to allow for any changes that may have occurred in the Apply Request Values or Validation phases. Any iterators that correspond to read-only components (such as an `outputText` component) are refreshed. Any dynamic regions are switched, if needed. The `prepareRender` event is sent to all registered listeners, as is the `afterJSFRenderResponse` event.

> **Note:**
>
> Instead of displaying `prepareRender` as a valid phase for a selection, JDeveloper displays `renderModel`, which represents the `refresh(RENDER_MODEL)` method called on the binding container.

- Render Response: The components in the tree are rendered as the Java EE web container traverses the tags in the page. State information is saved for subsequent requests and the Restore View phase.

  In order to lessen the wait time required to display both a page and any associated data, certain ADF Faces components such as the `table` component, use data streaming for their initial request. When a page contains one or more of these components, the page goes through the normal lifecycle. However, instead of fetching the data during that request, a special separate request is run. Because the page has just rendered, only the Render Response phase executes for the components that use data streaming, and the corresponding data is fetched and displayed. If the user's action (for example scrolling in a table), causes a subsequent data fetch another request is executed. Collection-based components, such as tables, trees, tree tables, and data visualization components all use data streaming.
- JSF Render Response: Provides before and after phase events for the Render Response phase. You can create a listener and register it with the before or after event of this phase, and the application will behave as if the listener were registered with the Render Response phase.

What You May Need to Know About Partial Page Rendering and Iterator Bindings

ADF Faces provides an optimized lifecycle that you can use when you want the page request lifecycle (including conversion and validation) to be run only for certain components on a page, usually for components whose values have changed.

One way to use the optimized lifecycle is to manually set up dependencies so that the events from one component act as triggers for another component, known as the *target*. When any event occurs on the trigger component, the lifecycle is run on any target components, as well as on any child components of both the trigger and the target, causing only those components to be rerendered. This is considered a partial page rendering (PPR)

The following example shows radio buttons as triggers and a `panelGroupLayout` component that contains the output text to be the target (the `panelGroupLayout` component must be the target because the `outputText` component may not always be rendered).

```
<af:form>
  <af:inputText label="Required Field" required="true"/>
  <af:selectBooleanRadio id="show" autoSubmit="true"
text="Show"
                        value="#{validate.show}"/>
  <af:selectBooleanRadio id="hide" autoSubmit="true" text="Hide"
                         value="#{validate.hide}"/>
  <af:panelGroupLayout partialTriggers="show hide" id="panel">
    <af:outputText value="You can see me!" rendered="#{validate.show}"/>
  </af:panelGroupLayout>
</af:form>
```

Because the `autoSubmit` attribute is set to `true` on the radio buttons, when they are selected, a `SelectionEvent` is fired. Because the `panelGroupLayout` component is set to be a target for both radio components, when that event is fired, only the `selectOneRadio` (the trigger), the `panelGroupLayout` component (the trigger's target), and its child component (the `outputText` component) are processed through the lifecycle. Because the `outputText` component is configured to render only when the **Show** radio button is selected, the user is able to select that radio button and see the output text, without having to enter text into the required input field above the radio buttons.

Instead of having to set partial triggers and targets within code manually, the ADF Faces framework provides automatic PPR. Automatic PPR happens when an event has a corresponding event root component, or when the component itself is an event root. An event root component determines the boundaries of PPR. For example, the `attributeChangeEvent` considers the `inputText` component that invokes the event, its event root. Therefore, in the previous example, when the value changes for the `inputText` component, because it is the initial boundary for the PPR, the lifecycle would be run only on that `inputText` component and any child components.

Along with event roots, by default for Fusion web applications, automatic PPR also occurs for all UI components associated with the same iterator binding. All associated components refresh whenever any one of them has a value change event. This functionality is controlled by setting the `changeEventPolicy` attribute for iterators to `ppr`. By default, this is set globally.

For example, say you create a number of `inputText` components using the Data Controls panel (for example, when you create a form). When the `attributeChangeEvent` occurs on one of those `inputText` components, because the other `inputText` components are associated with the same iterator, all those `inputText` components will also be refreshed.

Another example might be a page that contains a `panelSplitter` component, where the left part of the splitter contains a form to create a new customer (created by dropping a form using the `Customer` collection), and the right part of the splitter contains instructions for using the form. Because the bindings for the `inputText` components that make up the form are all associated with the `Customer` iterator, anytime the form is submitted and a value for one of the `inputText` components has changed, only the `inputText` components associated with the `Customer` will be processed through the lifecycle. The rest of the page, including the instructions in the right side of the splitter, will not be refreshed.

Note:

In order for automatic PPR to happen, the event and event root component must be listed in Table 5-1 of the Events and Partial Page Rendering section of *Developing Web User Interfaces with Oracle ADF Faces*. If it is not listed, you will need to set up PPR manually, as described in the Using Partial Triggers section of the same guide.

For example, say you create a form and a table from the same data control collection. The form shows the details for a row of data, while the table shows all rows in a collection, with the row currently displayed in the form shown as selected. When you click the **Next** button on the form, you want the table to refresh to show the new row as selected. Because the selection event is not supported by automatic PPR, the table will not refresh, even though the table and form both use the same iterator. You will need to set the **Next** and **Previous** buttons to be triggers of PPR for the target table.

> **Note:**
>
> Only UI components that have a control binding defined in the page definition file associated with the configured iterator will be refreshed.
>
> If you directly access the iterator from a managed bean and expose iterator values to UI components through that bean, then automatic PPR will not happen for those UI components. In these cases, you need to configure PPR manually. For more information, see the Using Partial Triggers section of *Developing Web User Interfaces with Oracle ADF Faces*.

By default, all new applications use global PPR. You do not have to set it. You can however, override the setting in the page definition file for a page.

Before you begin:

It may be helpful to have an understanding of partial page rendering. For more information, see What You May Need to Know About Partial Page Rendering and Iterator Bindings.

To set iterator bindings to use PPR:

1. To set a specific iterator binding to use or not use PPR, open the associated page definition file, select the iterator, and in the Properties window, set **ChangeEventPolicy**.
2. To set all iterator bindings to use PPR:
   a. In the Applications Resource panel, expand the **Descriptors** and **ADF META-INF** nodes, and then double click **adf-config.xml**.
   b. In the overview editor, click the **Model** navigation tab, and then select **Default Change Event Policy is Partial Page Rendering (PPR)**.

For more information about how the ADF Faces framework uses PPR, see the Rerendering Partial Page Content chapter in *Developing Web User Interfaces with Oracle ADF Faces*.

What You May Need to Know About Task Flows and the Lifecycle

Task flows are initially refreshed when the parent binding container (the one associated with the page) is refreshed. This happens in the Prepare Model phase. On a subsequent request, the task flow will be refreshed during the Prepare Render phase, depending on its `refresh` and `refreshCondition` attributes and its parameter value.

> **Note:**
>
> Any child page fragment's page definition still handles the refresh of the bindings of the child page fragments.

Tip:

If you have a region on a page that is not initially disclosed (for example, a popup dialog), the parameters still need to be available when the parent page is rendered, even though the region might not be displayed. If a region requires parameters, but those parameter values will not be available when the parent page is rendered, then you should use dynamic regions. If the parameters are null, an empty task flow can be used until the parameters for the region are ready and that region can display. To swap in an empty task flow, you set the dynamic region's `taskFlowId` attribute to an empty string.

If you set an EL expression as the value of the `refreshCondition` attribute, it will be evaluated during the Prepare Render phase of the lifecycle. When the expression evaluates to `true`, the task flow will be refreshed again. When `refreshCondition` evaluates to `false`, the behavior is the same as if the `refreshCondition` had not been specified.

Note:

If the variable `bindings` is used within the EL expression, the context refers to the binding container of the parent page, not the page fragment displayed within the region.

The valid values for the `refresh` property of a task flow executable are as follows:

- `default`: The region will be refreshed only once, when the parent page is first displayed.
- `ifNeeded`: Refreshes the region only if there has been a change to `taskFlow` binding parameter values. If the `taskFlow` binding does not have parameters, then `ifNeeded` is equivalent to the default. When `ifNeeded` is used, the `refreshCondition` attribute is not considered.

> **Note:**
>
> Setting the `refresh` attribute to `ifNeeded` takes precedence over any value for the `refreshCondition` attribute. Also note that `ifNeeded` is not supported when you pass parameters to the `taskFlow` binding using a dynamic parameter `Map`. Instead, use `refreshCondition="#{EL.Expression}"`.

Because the only job of the `taskFlow` binding is to refresh its parameters, setting `Refresh` to `always` does not make sense. If the `taskFlow` binding's parameters don't change, there is no reason to refresh the ADF region.

Note that the child page fragment's page definition still handles the refresh of the bindings of the child page fragments.

About Object Scope Lifecycles

When an object is created in a JSF application, it defines or defaults to a given scope; this object scope describes how widely it is available and who has access to it. You can use the standard JSF scopes or the ADF Faces scopes that are available for an object in a JSF application.

At runtime, ADF objects such as the binding container and managed beans are instantiated. Each of these objects has a defined lifespan set by its scope attribute. You can access a scope as a `java.util.Map` from the `RequestContext` API. For example, to access an object named `foo` in the request scope, you would use the expression `#{requestScope.foo}`.

Object scopes are analogous to global and local variable scopes in programming languages. The wider the scope, the higher and longer the availability of an object. During their life, these objects may expose certain interfaces, hold information, or pass variables and parameters to other objects. For example, a managed bean defined in session scope will be available for use during multiple page requests. However, a managed bean defined in request scope will only be available for the duration of one page request.

There are six types of scopes in a Fusion web application:

- Application scope: The object is available for the duration of the application.
- Session scope: The object is available for the duration of the session.

> **Note:**
>
> There is no window uniqueness for session scope, all windows in the session share the same session scope instance. If you are concerned about multiple windows being able to access the same object (for example to ensure that managed beans do not conflict across windows), you should use a scope that is window-specific, such as page flow or view scope.

- Page flow scope: The object is available for the duration of a task flow. As its name suggests, page flow scope is specific to an instance of a task flow (bounded or unbounded). So if you have a page with two or more regions (created by a task flow), separate instances of the page flow scope will exist for each region. Additionally, if the user opens the page with the regions in separate browser tabs, a separate instance of the page flow scope bean will be instantiated for each browser tab.

> **Note:**
>
> Because this is not a standard JSF scope, EL expressions must explicitly include the scope to reference bean. For example, to reference the `MyBean` managed bean from the `pageFlowScope` scope, your expression would be `#{pageFlowScope.MyBean}`.

- Request scope: The object is available from the time an HTTP request is made until a response is sent back to the client.
- Backing bean scope: Used for managed beans for page fragments and declarative components only, the object is available from the time an HTTP request is made until a response is sent back to the client. This scope is needed for fragments and declarative components because there may be more than one page fragment or declarative component on a page, and to prevent collisions, any values must be kept in separate scope instances. Therefore, any managed bean for a page fragment or declarative component must use backing bean scope.

> **Note:**
>
> Because this is not a standard JSF scope, EL expressions must explicitly include the scope to reference bean. For example, to reference the `MyBean` managed bean from the backing bean scope, your expression would be `#{backingBeanScope.MyBean}`.

- View scope: The object is available until the view ID for the current view activity changes. This scope can be used to hold values for a given page. However, unlike request scope, which can be used to store a value needed from one page to the next, anything stored in view scope will be lost once the view ID changes.

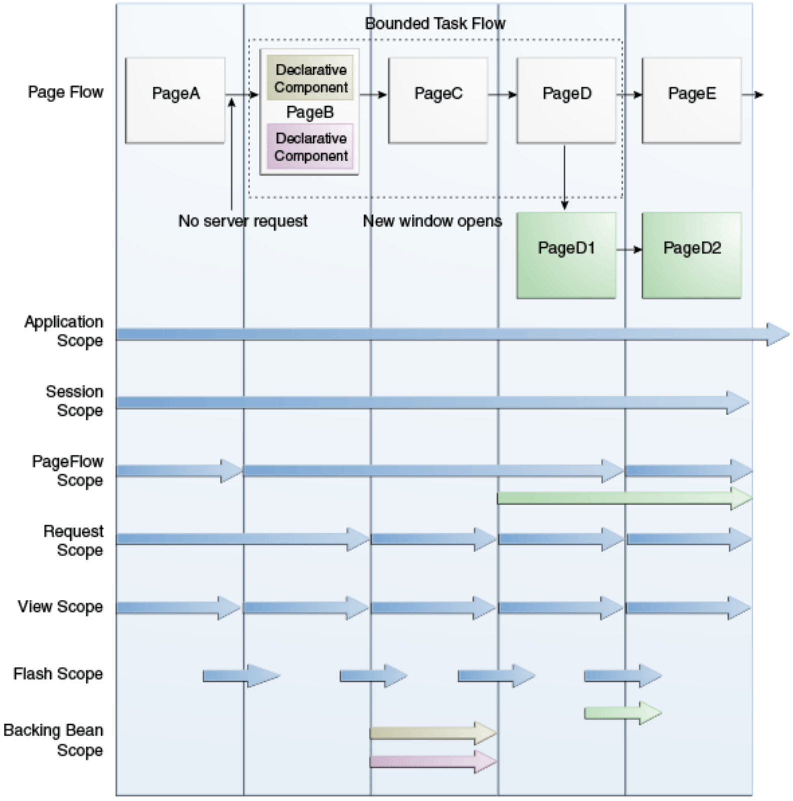
> **Note:**
>
> Both JSF and ADF Faces have an implementation of view scope. Only the ADF Faces view scope survives page redirects and refreshes. In a Fusion web application, when you use `viewScope` in an expression, it resolves to the ADF Faces view scope.
>
> Because this view scope is not a standard JSF scope, EL expressions must explicitly include the scope to reference bean. For example, to reference the `MyBean` managed bean from the view scope, your expression would be `#{viewScope.MyBean}`.

> **Note:**
>
> When you create objects (such as a managed bean) that require you to define a scope, you can set the scope to `none`, meaning that it will not live within any particular scope, but will instead be instantiated each time it is referenced.

By default, the binding container and the binding objects it contains are defined in session scope. However, the *values* referenced by value bindings and iterator bindings are undefined between requests and for scalability reasons do not remain in session scope. Therefore, the values that binding objects refer to are valid only during a request in which that binding container has been prepared by the ADF lifecycle. What stays in session scope are only the binding container and binding objects themselves.

Figure 30-4 shows the time period during which each type of scope is valid.

Figure 30-4 Relationship Between Scopes and Page Flow



When determining what scope to register a managed bean with, always try to use the narrowest scope possible. Only use the session scope for information that is relevant to the whole session, such as user or context information. Avoid using session scope to pass values from one task flow to another. When creating a managed bean for a page fragment or a declarative component, you must use backing bean scope.

Managed beans can be registered in either the `adfc-config.xml` or the configuration file for a specific task flow. For more information about using managed beans in a Fusion application, see Using a Managed Bean in a Fusion Web Application.

Note:

Do not register managed beans in the `faces-config.xml` in a Fusion web application.

What You May Need to Know About Object Scopes and Task Flows

When determining what scope to use for variables within a task flow, you should use any of the scope options other than application or session scope. These two scopes will persist objects in memory beyond the life of the task flow and therefore compromise the encapsulation and reusable aspects of a task flow. In addition, application and session scopes may keep objects in memory longer than needed, causing unneeded overhead.

When you need to pass data values between activities within a task flow, you should use page flow scope. View scope should be used for variables that are needed only within the current view activity, not across view activities. Request scope should be used when the scope does not need to persist longer than the current request. It is the only scope that should be used to store UI component information. Lastly, backing bean scope must be used for backing beans in your task flow if there is a possibility that your task flow will appear in two region components or declarative components on the same page and you would like to achieve region instance isolations.

Customizing the ADF Page Lifecycle

The ADF page lifecycle performs a sequence of events that occur while processing a web page request using JSF and Oracle ADF. You can create and register a custom phase listeners to customize the ADF page lifecycle.

The ADF lifecycle contains clearly defined phases that notify ADF lifecycle listeners before and after the corresponding JSF phase is executed. You can customize this lifecycle by creating a custom phase listener that invokes your needed code, and then registering it with the lifecycle to execute during one of these phases.

> **Note:**
>
> An application cannot have multiple phase listener instances. An initial `ADFPhaseListener` instance is, by default, registered in the `META-INF/faces-config.xml` configuration file. Registering, for example, a customized subclass of the `ADFPhaseListener` creates a second instance. In this scenario, only the instance that was most recently registered is used.
>
> The following warning message indicates when an instance has been replaced by a newer one: "ADFc: Replacing the ADF Page Lifecycle implementation with *class name of the new listener*."

How to Create a Custom Phase Listener

To create a custom phase listener, you must create a listener class that implements the `PagePhaseListener` interface. You then add methods that execute code either before or after the phase that the code needs to execute.

The following example contains a template that you can modify to create a custom phase listener. See How to Generate Custom Classes, for more information about creating a class in JDeveloper.

```
public class MyPagePhaseListener implements PagePhaseListener
{
   public void afterPhase(PagePhaseEvent event)
   {
      System.out.println("In afterPhase " + event.getPhaseId());
   }


   public void beforePhase(PagePhaseEvent event)
```

```
    {
       System.out.println("In beforePhase " + event.getPhaseId());
    }
}
```

Once you create the custom listener class, you need to register it with the phase in which the class needs to be invoked. You can either register it globally (so that the whole application can use it), or you can register it only for a single page.

How to Register a Listener Globally

To customize the ADF lifecycle globally, register your custom phase listener by editing the `adf-settings.xml` configuration file. The `adf-settings.xml` file is shared by several ADF components, including ADF Controller, to store configuration information.

To register the listener in adf-settings.xml:

1. In the Applications window, beneath the **META-INF** node, double-click the **adf-settings.xml** file.
2. In the editor window, click the **Source** tab
3. In the source editor, scroll down to `<adfc-controller-config xmlns= "http://xmlns.oracle.com/adf/controller/config">`

   If this entry does not exist, add it to the file.
4. Enter the remaining elements shown in italics:

```
 <?xml version="1.0" encoding="US-ASCII" ?> <adf-config xmlns="http://
xmlns.oracle.com/adf/config">
  .
  .
  .
    <adfc-controller-config xmlns="http://xmlns.oracle.com/adf/controller/
config">
       <lifecycle>
         <phase-listener>
            <listener-id>MyPagePhaseListener</listener-id>
            <class>mypackage.MyPagePhaseListener</class>
         </phase-listener>
      </lifecycle>
   </adfc-controller-config>
  .
  .
  .
</adf-config>
```

5. Add values for the following elements:
   - `<listener-id>` A unique identifier for the listener (you can use the fully qualified class name)
   - `<class>` The class name of the listener

What You May Need to Know About Listener Order

You can specify multiple phase listeners in the `adf-settings.xml` and, optionally, the relative order in which they are called. When registering a new listener in the file, you determine the position in the list of listeners using two parameters:

- `beforeIdSet`: The listener is called before any of the listeners specified in `beforeIdSet`
- `afterIdSet`: The listener is called after any of the listeners specified in `afterIdSet`

The following example contains an example configuration file in which multiple listeners have been registered for an application.

```
<lifecycle>
   <phase-listener>
      <listener-id>MyPhaseListener</listener-id>
      <class>view.myPhaseListener</class>
      <after-id-set>
         <listener-id>ListenerA</listener-id>
         <listener-id>ListenerC</listener-id>
      </after-id-set>
      <before-id-set>
         <listener-id>ListenerB</listener-id>
         <listener-id>ListenerM</listener-id>
         <listener-id>ListenerY</listener-id>
      </before-id-set>
   </phase-listener>
</lifecycle>
```

In the example, `MyPhaseListener` is a registered listener that executes after listeners A and C but before listeners B, M, and Y. To execute `MyPhaseListener` after listener B, move the `<listener-id>` element for listener B under the `<after-id-set>` element.

How to Register a Lifecycle Listener for a Single Page

To customize the lifecycle of a single page, you set the `ControllerClass` attribute on the page definition file. This listener will be valid only for the lifecycle of the particular page described by the page definition. For more information about the page definition file and its role in a Fusion web application, see Working with Page Definition Files.

You specify a different controller class depending on whether it is for a standard JSF page or a page fragment.

To customize the ADF Lifecycle for a single page or page fragment:

1. In the Applications window, right-click the page or page fragment and choose **Go To Page Definition**.
2. In the Structure window, select the page definition node.
3. In the Properties window, click the icon that appears when you hover over the **ControllerClass** field, and choose **Edit**.
4. Click the **Hierarchy** tab and navigate to the appropriate controller class for the page or page fragment. Following are the controller classes to use for different types of pages:
   - Standard JSF page - specify `oracle.adf.controller.v2.lifecycle.PageController`

     If you need to receive afterPhase/beforePhase events, specify `oracle.adf.controller.v2.lifecycle.PagePhaseListener`
   - Page fragment - specify `oracle.adf.model.RegionController`

Tip:

You can specify the value of the page definition's `ControllerClass` attribute as a fully qualified class name or you can enter an EL expression that resolves to a class directly in the **ControllerClass** field.

When using an EL expression for the value of the `ControllerClass` attribute, the Structure window may show a warning indicating that e "`#{YourExpression}`" is not a valid class. You can safely ignore this warning.

31

Creating a Basic Databound Page

This chapter describes how to create databound forms in the Fusion web application with data modeled from ADF Business Components, using ADF data controls and ADF Faces components. It describes how to create text fields from individual attributes, generate entire forms from collections for editing and creating new records, and create dynamic forms.
This chapter includes the following sections:

About Creating a Basic Databound Page

A simple data bound web UI helps you to get a feel of the visual and declarative development experience offered by the JDeveloper IDE along with Oracle ADF framework. You can drag and drop UI components as per your requirement.

You can create UI pages that allow you to display and collect information using data controls created for your business services. For example, using the Data Controls panel, you can drag an attribute for an item, and then choose to display the value either as output text or as an input text field with a label. JDeveloper creates all the necessary JSF tags and binding code needed to display and update the associated data. For more information about the Data Controls panel and the declarative binding experience, see Using ADF Model in a Fusion Web Application.

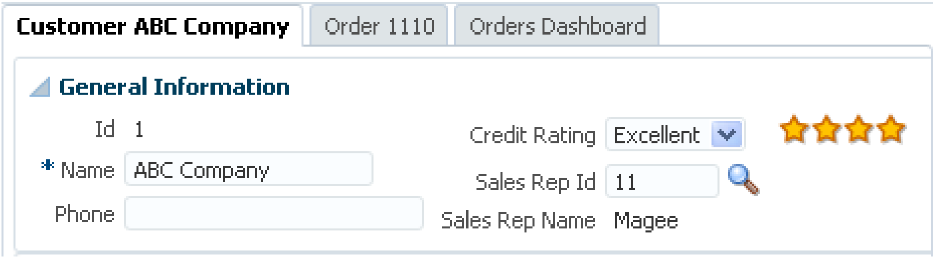
In addition to being able to drop individual attributes, you can drop all attributes for an object at once as a form, table, or single-column list view. This chapter includes information on creating forms that display values, forms that allow users to edit values, and forms that collect values (input forms). For information on creating tables and list views, see Creating ADF Databound Tables .

Once you drop the UI components, you can then drop built-in operations as command UI components that allow you to navigate through the records in a collection or that allow users to operate on the data, such as committing, deleting, or creating a record. For example, you can create a button that allows users to delete data objects displayed in the form. You can also modify the default components to suit your needs.

ADF Databound Form Use Cases and Examples

You use forms when you need to collect or display a row of data. For example, the Summit sample application for **Oracle ADF** contains a panel that allows users to update information about their customers as shown in Figure 31-1. This form was created by dragging and dropping the `Customers` collection from the Data Controls panel and then selecting the fields to display.

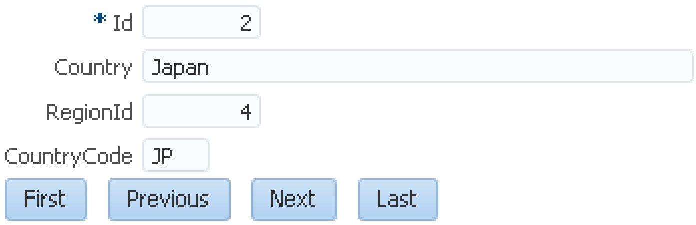
Figure 31-1 Customer General Information Form in the Summit ADF Sample Application



Id 1
* Name ABC Company
Phone
Credit Rating Excellent ★★★★
Sales Rep Id 11
Sales Rep Name Magee

When you create a form, you can also choose to add navigation, so that the user can navigate quickly between records, as shown in Figure 31-2.

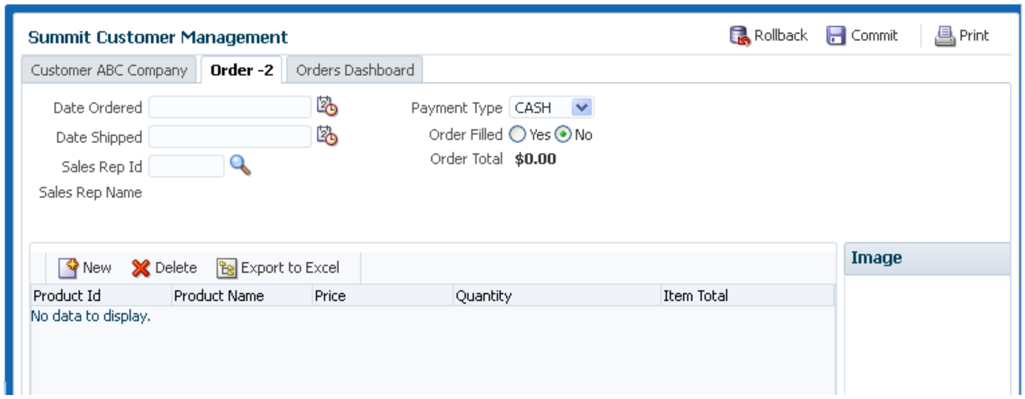
Figure 31-2 Navigate Between Records in a Form



* Id 2
Country Japan
RegionId 4
CountryCode JP

You can also add command buttons that invoke processing on the row displayed in the form. For example, you can change values for a record and save those changes in an edit form, or you can create an input form that allows users to create a new record. Figure 31-3 shows an input form where a user can create a new order.

Figure 31-3 Input Form



Date Ordered

Date Shipped

Sales Rep Id

Sales Rep Name

Payment Type CASH

Order Filled ○ Yes ● No

Order Total $0.00

Product Id	Product Name	Price	Quantity	Item Total

No data to display.

> **Note:**
>
> When the attributes to display in a form are only available at runtime, you can create a dynamic form. For more information, see Creating a Form with Dynamic Components.

Additional Functionality for Databound Pages

You may find it helpful to understand other Oracle ADF features before you implement your databound pages. Following are links to other functionality that you may find useful.

- **ADF view objects**: Much of how the components display and function in the form is controlled by the corresponding **view objects**. See Defining SQL Queries Using View Objects.
- **ADF application modules**: The Data Controls panel, from which you drag databound components to your pages, is populated with representations of view objects that you have added to application modules. See Implementing Business Services with Application Modules.
- **Adapter-based data controls**: If you are using other types of business services, such as EJB components or web services, you can create data controls for those business services as described in Creating and Configuring EJB Data Controls in *Developing Applications with Oracle ADF Data Controls*.
- **ADF Model and data binding**: When you create forms in an ADF web application, you use ADF Model and data binding. See Using ADF Model in a Fusion Web Application.
- **ADF Faces**: You also use ADF Faces UI components. For detailed information about developing with ADF Faces, see Introduction to ADF Faces in *Developing Web User Interfaces with Oracle ADF Faces*.
- **Advanced UI controls**: Basic forms use input and output components. You can also use more advanced components, such as lists, tables, trees, search forms, LOV components, graphs, gauges, treemaps, and more. See the remaining chapters in Creating a Databound Web User Interface .

- **Task flows**: If your form takes part in a transaction, for example an input form, then you may need to use an ADF task flow to invoke certain operations before or after the form is rendered. See Creating ADF Task Flows .
- **Page lifecycle:** For information about how forms work with the page lifecycle, see Understanding the Fusion Page Lifecycle .
- **Validation**: You may want certain fields to be validated before they are submitted to the data store. See Defining Validation and Business Rules Declaratively and Using Validation in the ADF Model Layer .
- **Active data**: If your application uses active data, then you can have the data in your UI components update automatically, whenever the data in the data source changes. See Using the Active Data Service .

Creating Text Fields Using Data Control Attributes

The Data Controls panel helps you to create text fields that can display or update an attribute. A page definition file is created for the page (if one does not already exist) after you drag an attribute onto a JSF page and drop it as a UI component.

JDeveloper allows you to create text fields declaratively in a WYSIWYG development environment for your JSF pages, meaning you can design most aspects of your pages without needing to look at the code. When you drag and drop items from the Data Controls panel, JDeveloper declaratively binds ADF Faces text UI components to attributes on a data control using an attribute binding.

How to Create a Text Field

To create an individual text field that can display or update an attribute, you drag and drop an attribute of a collection from the Data Controls panel. For **ADF Business Components**, the Data Controls panel is populated with elements of your data model that you have defined in at least one **application module**.

Before you begin:

It may be helpful to have a general understanding of using data control attributes to create text fields. For more information, see Creating Text Fields Using Data Control Attributes.

You will need to complete these tasks:

- Create an application module that contains instances of the view objects that you want in your data model, as described in Creating and Modifying an Application Module. Alternatively, if you are using another type of business service, create a data control for that business service as described in Exposing Business Services with Data Controls in *Developing Applications with Oracle ADF Data Controls*.
- Create a JSF page as described in Creating a Web Page.

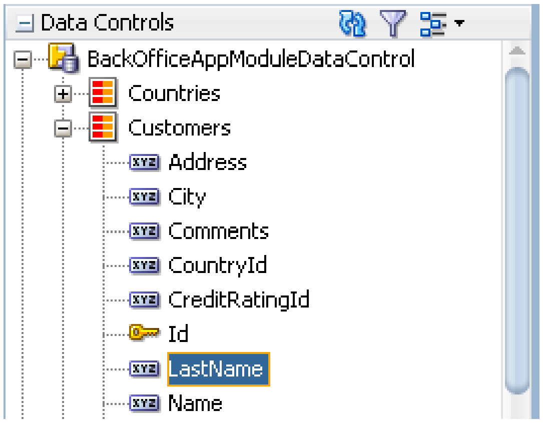
To create a databound text field:

1. From the Data Controls panel, select an attribute from a collection. For a description of the icons that represent attributes and other objects in the Data Controls panel, see Using the Data Controls Panel.

   For example, Figure 31-4 shows the `LastName` attribute under the `Customers` collection of the `BackOfficeAppModuleDataControl` data control in the Summit

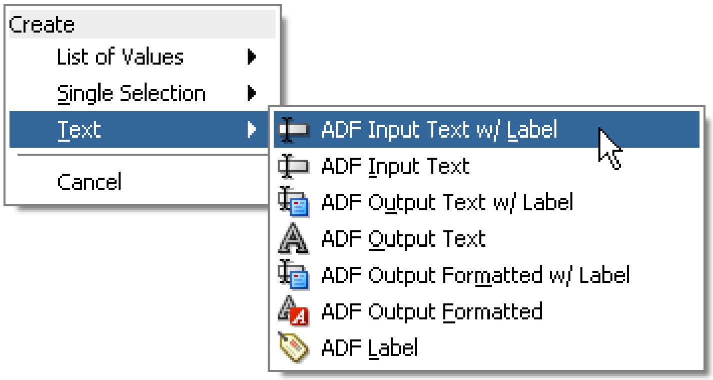
ADF sample application. This is the attribute to drop to display, update, or insert the customer's last name.

Figure 31-4 Attributes Associated with a Collection in the Data Controls Panel



2. Drag the attribute onto the page, and from the context menu choose the type of UI component to display or collect the attribute value, as shown in Figure 31-5.

Figure 31-5 Context Menu When Dropping Attributes on a Page



For an attribute, you are given the following choices:

- **Text**:
  - **ADF Input Text w/ Label**: Creates an ADF Faces `inputText` component in which the `label` attribute is populated. In addition, a nested `validator` component is created, which you can use to add client-side validation.

    > **Tip:**
    >
    > For more information about the various attributes of the `inputText` component, see the Using Input Components and Defining Forms chapter of *Developing Web User Interfaces with Oracle ADF Faces*.

  - **ADF Input Text**: Creates an ADF Faces `inputText` component without a label. In addition, a nested `validator` component is generated, which you can use for client-side validation.
  - **ADF Output Text w/ Label**: Creates a `panelLabelAndMessage` component that holds an ADF Faces `outputText` component. The `label` attribute on the `panelLabelAndMessage` component is populated.
  - **ADF Output Text:** Creates an ADF Faces `outputText` component.
  - **ADF Output Formatted w/Label**: Same as ADF Output Text w/Label, but uses an `outputFormatted` component instead of an `outputText` component. The `outputFormatted` component allows you to add a limited amount of HTML formatting. For more information, see the Displaying Output Text and Formatted Output Text section of *Developing Web User Interfaces with Oracle ADF Faces*.
  - **ADF Output Formatted**: Same as ADF Output Formatted w/Label, but without the label.
  - **ADF Label**: An ADF Faces `outputLabel` component.
- **List of Values:** Creates ADF LOV lists. For more information about how these lists work for Business Components, see Working with List of Values (LOV) in View Object Attributes. For more information on how these lists work for adapter-based data controls, see Creating List of Values Objects in *Developing Applications with Oracle ADF Data Controls*. For more information about using the lists on a JSF page, see Creating a Selection List.
- **Single selections**: Creates single selection lists. For more information about creating lists on a JSF page, see Creating a Selection List.

> **Note:**
>
> These selections are your choices by default. However, the list of components available to use for an attribute can be configured as a control hint on the associated entity or view object. For more information, see Defining Attribute Control Hints for Entity Objects and Defining UI Hints for View Objects. For information on adding control hints for adapter-based data controls, see How to Set UI Hints on Attributes in *Developing Applications with Oracle ADF Data Controls*.

For the purposes of this chapter, only the text components (and not the lists) will be discussed.

What Happens When You Create a Text Field

Among other things, when you drag an attribute onto a JSF page and drop it as a UI component, a page definition file is created for the page (if one does not already exist). For a complete account of what happens when you drag an attribute onto a page, see What Happens When You Use the Data Controls Panel. Bindings for the iterator and attributes are created and added to the page definition file. Additionally, the necessary JSPX or JSF page code for the UI component is added to the JSF page.

Iterator Bindings Created in the Page Definition File

Whenever you create UI components on a page by dropping an item that is part of a collection from the Data Controls panel (or you drop the whole collection as a form or table), JDeveloper creates an iterator binding if it does not already exist. An iterator binding references an iterator for the data collection, which facilitates iterating over its data objects. It also manages currency and state for the data objects in the collection. An iterator binding does not actually access the data. Instead, it simply exposes the object that can access the data and it specifies the current data object in the collection. Other bindings then refer to the iterator binding in order to return data for the current object or to perform an action on the object's data. Note that the iterator binding is not an iterator. It is a binding to an iterator. In the case of ADF Business Components, the actual iterator is the default **row set iterator** for the default row set of the view object instance in the application module's data model.

For example, if you drop the `LastName` attribute of the `Customers` collection, JDeveloper creates an iterator binding for the `Customers` collection.

> **Tip:**
>
> There is one iterator binding created for each collection. This means that when you drop two attributes from the same collection (or drop the collection twice), they use the same binding. One advantage of this behavior is that the currency of components stays in sync. For example, if the page has a table and a form dropped from the same collection, the form will show the record of the selected row in the table. If you need the binding to behave differently for the different components, you need to manually create separate iterator bindings.

The iterator binding's `rangeSize` attribute determines how many rows of data are fetched from a data control each time the iterator binding is accessed. This attribute gives you a relative set of 1-*n* rows positioned at some absolute starting location in the overall row set. When you create the iterator binding by dragging an attribute or collection on to a page, the attribute is initially set to `25`. For more information about using this attribute, see Iterator RangeSize Attribute Defined. The following example shows the iterator binding created when you drop an attribute from the `Customers` collection.

```
<executables>
  <iterator Binds="Customers" RangeSize="25"
            DataControl="BackOfficeAppModuleDataControl"
            id="Customers"/>
</executables>
```

For information regarding the iterator binding element attributes, see Oracle ADF Binding Properties.

This metadata allows the ADF binding container to access the attribute values. Because the iterator binding is an executable, by default, it is invoked when the page is loaded, thereby allowing the iterator to access and iterate over the `Customers` collection. This means that the iterator will manage all the `Customers` objects in the collection, including determining the current `Customers` object or range of `Customers` objects.

Value Bindings Created in the Page Definition File

When you drop an attribute from the Data Controls panel, JDeveloper creates an attribute binding that is used to bind the UI component to the attribute's value. This type of binding presents the value of an attribute for a single object in the current row in the collection. Value bindings can be used both to display and to collect attribute values.

For example, if you drop the LastName attribute under the Customers collection as an ADF Output Text w/Label widget onto a page, JDeveloper creates an attribute binding for the LastName attribute. This allows the binding to access the attribute value of the current record. The following example shows the attribute binding for LastName created when you drop the attribute from the Customers collection. Note that the attribute value references the iterator named CustomersIterator.

```
<bindings>
    ...
  <attributeValues IterBinding="CustomersIterator"
                   id=""LastName
    <AttrNames>
      <Item Value="LastName">
    </AttrNames>
  </attributeValues>
</bindings>
```

For information regarding the attribute binding element properties, see Oracle ADF Binding Properties.

Component Tags Created in JSF Page

When you create a text field by dropping an attribute from the Data Controls panel, JDeveloper creates the UI component associated with the widget dropped by writing the corresponding tag to the JSF page.

For example, when you drop the LastName attribute as an Output Text w/Label widget, JDeveloper inserts the tags for a panelLabelAndMessage component and an outputText component. It creates an EL expression that binds the label attribute of the panelLabelAndMessage component to the label property of hints created for the LastName's binding. This expression evaluates to the label hint set on the view object (for more information about hints, see Defining UI Hints for View Objects). It creates another expression that binds the outputText component's value attribute to the inputValue property of the LastName binding, which evaluates to the value of the LastName attribute for the current row. An ID is also automatically generated for both components.

> **Tip:**
>
> JDeveloper automatically generates IDs for all ADF Faces components. You can override these values as needed.

The following example shows the code generated on the JSF page when you drop the LastName attribute as an Output Text w/Label widget.

```
<af:panelLabelAndMessage label="#{bindings.LastName.hints.label}" id="plam1">
  <af:outputText value="#{bindings.LastName.inputValue}" id="ot1"/>
</af:panelLabelAndMessage>
```

If instead you drop the `LastName` attribute as an Input Text w/Label widget, JDeveloper creates an `inputText` component. As the following example shows, the value is bound to the `inputValue` property of the `LastName` binding. Additionally, the following properties are also set:

- `label`: Bound to the object's `label` UI hint.
- `required`: Bound to the object's `mandatory` property.
- `columns`: Bound to the object's `displayWidth` UI hint, which determines how wide the text box will be.
- `maximumLength`: Bound to the object's `precision` property, which determines the maximum number of characters per line that can be entered into the field.
- `shortDesc`: Bound to the `tooltip` UI hint.

In addition, JDeveloper nests a `validator` tag within the `inputText` component. You can use this tag to create client-side validation rules to supplement any validation rules that exist in the model or business service layers.

```
<af:inputText value="#{bindings.LastName.inputValue}"
              label="#{bindings.LastName.hints.label}"
              required="#{bindings.LastName.hints.mandatory}"
              columns="#{bindings.LastName.hints.displayWidth}"
              maximumLength="#{bindings.LastName.hints.precision}">
              shortDesc="#{bindings.LastName.hints.tooltip}" id="it1">
  <f:validator binding="#{bindings.LastName.validator}"/>
</af:inputText>
```

For further information regarding the `validator` and `converter` tags, see the Validating and Converting Input chapter of *Developing Web User Interfaces with Oracle ADF Faces*.

You can change any of these values to suit your needs. For example, the `mandatory` control hint on the view object is set to `false` by default, which means that the `required` attribute on the component will evaluate to `false` as well. You can override this value by setting the `required` attribute on the component to `true`. If you decide that all instances of the attribute should be mandatory, then you can change the control hint on the view object, and all instances will then be required. For more information about these properties, see Oracle ADF Binding Properties. For more information on changing these properties at the entity object level, see Setting Attribute Properties. For more information on changing these properties at the view object level, see How to Edit a View Object.

Creating Basic Forms Using Data Control Collections

A basic form helps you to created a page and display data from a collection. The Data Controls panel helps you to drag and drop a collection to create a form. You can also modify the data present in the form.

Instead of dropping each of the individual attributes of a collection to create a form, you can drop the collection node to create a form from multiple attributes in the collection.

For example, you could create a page that displays basic information about customers in the Summit ADF sample application by dragging and dropping the `Customers` collection.

You can also create forms that provide more functionality than simply displaying data from a collection. For information about creating a form that allows a user to update data, see Creating a Form to Edit an Existing Record. For information about creating forms that allow users to create a new object for the collection, see Creating an Input Form. You can also create search forms. For more information, see Creating ADF Databound Search Forms .

How to Create a Form Using a Data Control Collection

When you create a form using a data control, you bind the UI components to the attributes on the corresponding object in the data control. JDeveloper allows you to do this declaratively by dragging and dropping a collection from the Data Controls panel.

Before you begin:

It may be helpful to have an understanding of basic databound forms. For more information, see Creating Basic Forms Using Data Control Collections.

You may also find it helpful to understand other ADF functionality and features. For more information, see Additional Functionality for Databound Pages.

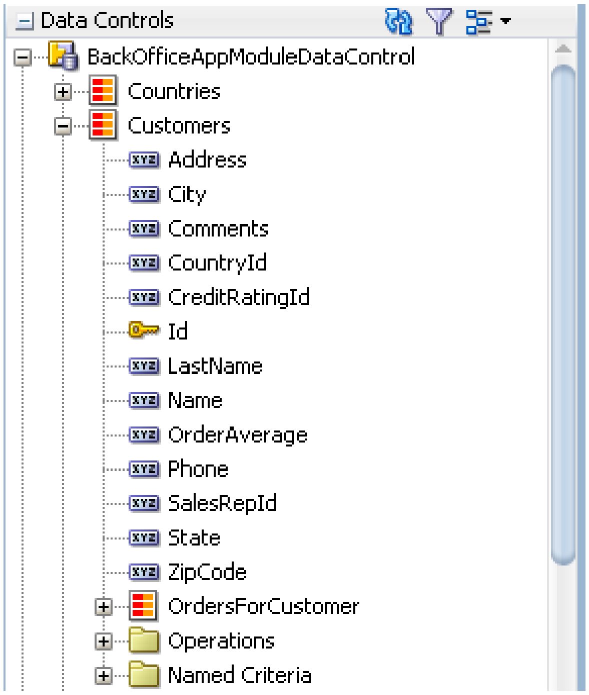
You will need to complete these tasks:

- Create an application module that contains instances of the view objects that you want in your data model, as described in Creating and Modifying an Application Module. Alternatively, if you are using another type of business service, create a data control for that business service as described in Exposing Business Services with Data Controls in *Developing Applications with Oracle ADF Data Controls*.
- Create a JSF page as described in Creating a Web Page.

To create a basic form:

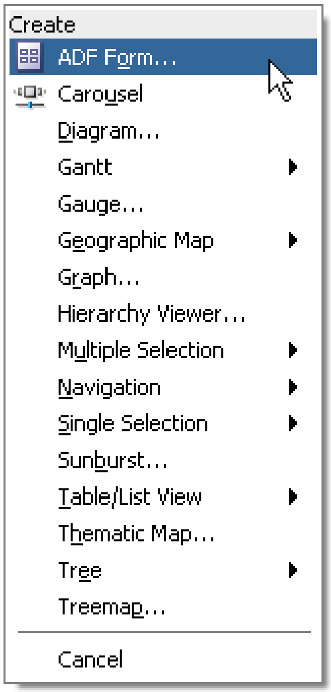
1. From the Data Controls panel, select the collection that represents the data you wish to display. Figure 31-6 shows the `Customers` collection for the `BackOfficeAppModuleDataControl` data control.

Figure 31-6 CustomerVO Object in the Data Controls Panel



2. Drag the collection onto the page, and from the context menu choose **ADF Form**.

Figure 31-7 Context Menu When Dropping a Collection to a Page



3. In the Create Form dialog, configure your form.

   - You can exclude attributes from appearing in the form.

- If you do not select the **Read-Only Form** checkbox, ADF `inputText` components are used for most attributes. Each `inputText` component has the `label` attribute populated. By default, the label obtains its value from a binding to the `label` property on the associated binding object.

  Attributes that are dates use the `InputDate` component.

  If a control type control hint has been created for an attribute, or if the attribute has been configured to be a list, then the component set by the hint is used instead.

  `InputText` components contain a `validator` tag that allows you to set up client-side validation for the attribute. If the attribute is a number or a date, a `converter` is also included.

  > **Tip:**
  >
  > For more information about the various attributes of the `inputText` component, see the Using Input Components and Defining Forms chapter of *Developing Web User Interfaces with Oracle ADF Faces.*

- If you select the **Read-Only Form** checkbox, read-only `outputText` components are used. Since the form is meant to display data, no `validator` tags are added (though converters are included). Attributes of type `Date` use the `outputText` component when in a read-only form. All components are placed inside `panelLabelAndMessage` components, which have the `label` attribute populated, and are placed inside a `panelFormLayout` component. This ensures that the labels are aligned appropriately with the `outputText` components at runtime.
- You can elect to include navigational controls that allow users to navigate through all the data objects in the collection. For more information, see Incorporating Range Navigation into Forms.
- You can also include a **Submit** button to submit the HTML form and apply the data in the form to the bindings as part of the JSF/ADF page lifecycle. For additional help in using the dialog, click **Help**.

4. If you are building a form that allows users to update data, you now need to drag and drop an operation that will commit the changes to update the record. For more information, see Creating a Form to Edit an Existing Record.

> **Note:**
>
> If you select the **Fields Generated at Runtime** in the Create Form dialog, the attributes to display are determined at runtime based on configuration of the business service. For more information, see Creating a Form with Dynamic Components.

What Happens When You Create a Form Using a Data Control Collection

Dropping an object as a form from the Data Controls panel has a similar effect as dropping a single attribute, except that multiple attribute bindings and associated UI components are created and all UI components are placed inside a `panelFormLayout` component. The attributes on the UI components (such as `value`) are bound to properties on that attribute's binding object (such as `inputValue`) or to the values of control hints set on the corresponding business object. The following example shows some of the code generated on the JSF page when you drop the `Customers` collection as a default ADF Form.

> **Note:**
>
> If an attribute is marked as hidden on the associated view or entity object, then no corresponding UI is created for it.

```
<af:panelFormLayout id="pfl1">
    <af:inputText value="#{bindings.Id.inputValue}"
                  label="#{bindings.Id.hints.label}"
                  required="#{bindings.Id.hints.mandatory}"
                  columns="#{bindings.Id.hints.displayWidth}"
                  maximumLength="#{bindings.Id.hints.precision}"
                  shortDesc="#{bindings.Id.hints.tooltip}" id="it1">
        <f:validator binding="#{bindings.Id.validator}"/>
        <af:convertNumber groupingUsed="false" pattern="#{bindings.Id.format}"/>
    </af:inputText>
    <af:inputText value="#{bindings.Name.inputValue}"
                  label="#{bindings.Name.hints.label}"
                  required="#{bindings.Name.hints.mandatory}"
                  columns="#{bindings.Name.hints.displayWidth}"
                  maximumLength="#{bindings.Name.hints.precision}"
                  shortDesc="#{bindings.Name.hints.tooltip}" id="it2">
        <f:validator binding="#{bindings.Name.validator}"/>
    </af:inputText>
    <af:inputText value="#{bindings.Phone.inputValue}"
                  label="#{bindings.Phone.hints.label}"
                  required="#{bindings.Phone.hints.mandatory}"
                  columns="#{bindings.Phone.hints.displayWidth}"
                  maximumLength="#{bindings.Phone.hints.precision}"
                  shortDesc="#{bindings.Phone.hints.tooltip}" id="it3">
        <f:validator binding="#{bindings.Phone.validator}"/>
    </af:inputText>
. . .
</af:panelFormLayout>
```

For more information about the code that is generated when you drop an item from the Data Controls panel, see What Happens When You Create a Text Field .

When you choose to create an input form using an object that contains a defined **list of values (LOV)**, then a `selectOneChoice` component is created instead of an `inputText` component. For example, the `CustomerVO` view object contains defined LOVs for the `CountryId` and `CreditRatingId` attributes. When you drop the `Customers` data control object as an ADF Form, a dropdown list showing all values is created for each LOV (instead of an empty input text field). For more information about how these lists work, see Working with List of Values (LOV) in View Object Attributes. For more information on how these lists work for adapter-based data controls, see Creating List of Values Objects in *Developing Applications with Oracle ADF Data Controls*. For more information about using the lists on a JSF page, see Creating a Selection List.

> **Note:**
>
> If the object contains a structured attribute (an attribute that is neither a Java primitive type nor a collection), that attribute will not appear in the dialog, and it will not have a corresponding component in the form. You will need to create those fields manually.

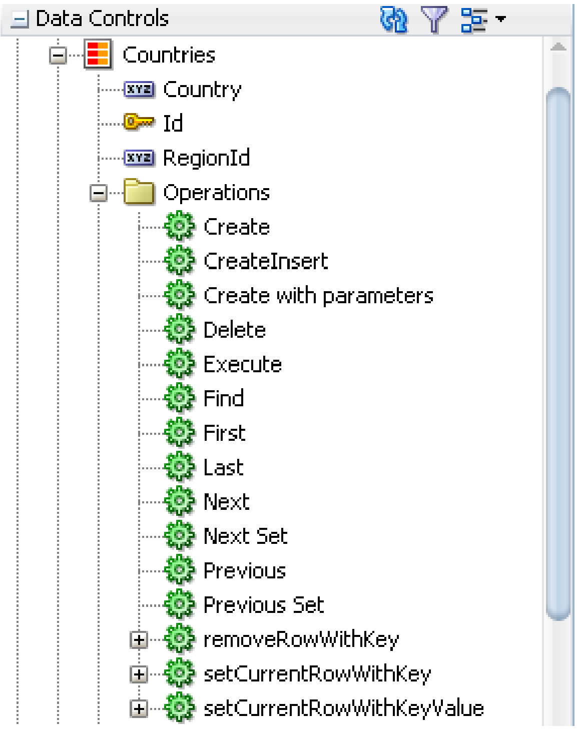
Creating Command Components Using Data Control Operations

Command components are represented by the UICommand component, which performs an action when it is activated. The Data Controls panel helps you to drag and drop a command component based on a data control operation to a page.

In addition to providing a representation of your data objects, the Data Controls panel also has nodes that represent standard operations for use in a form. These built-in operations enable you to declaratively handle common form functions such as navigating between records and committing changes to a database. Most operations are available for individual data collections in a data control. The Commit and Rollback operations are available on the whole data control. When you drag an operation from the Data Controls panel to a page, you are prompted to choose what kind of command component to create, such as a button or a link.

Figure 31-8 shows the operations available for the `Countries` collection in the Summit ADF sample application.

Figure 31-8 Data Control Collection Operations



Data Controls
- Countries
  - Country
  - Id
  - RegionId
  - Operations
    - Create
    - CreateInsert
    - Create with parameters
    - Delete
    - Execute
    - Find
    - First
    - Last
    - Next
    - Next Set
    - Previous
    - Previous Set
    - removeRowWithKey
    - setCurrentRowWithKey
    - setCurrentRowWithKeyValue

This section shows how to create command components from built-in data control operations. For information on other ways you can use command components in a databound page, see Using Command Components to Invoke Functionality in the View Layer.

How to Create Command Components From Operations

To create a command component based on a data control operation, you drag and drop the operation from the Data Controls panel to a page.

Before You Begin:

It may be helpful to have a general understanding of using data control operations to create command components. For more information, see Creating Command Components Using Data Control Operations.

You will need to complete these tasks:

- Create an application module that contains instances of the view objects that you want in your data model, as described in Creating and Modifying an Application Module. Alternatively, if you are using another type of business service, create a data control for that business service as described in Exposing Business Services with Data Controls in *Developing Applications with Oracle ADF Data Controls*.
- Create a JSF page as described in Creating a Web Page.

To create a command component from an operation:

1. From the Data Controls panel, drag the operation onto the page.
2. From the ensuing context menu, choose either **ADF Button** or **ADF Link**.

What Happens When You Create Command Components Using Operations

When you drop an operation to create a command component, JDeveloper:

- Defines an action binding in the page definition file for the associated operation
- Configures the iterator binding to use partial page rendering for the collection
- Inserts code in the JSF page for the command component

Action Bindings Created for Operations

Action bindings execute business logic. For example, they can invoke operations on the action binding object. These operations operate on the iterator or on the data control itself.

Like value bindings, action bindings for operations contain a reference to the iterator binding when the action binding is bound to one of the iterator-level actions, such as `Next` or `Previous`. These types of actions are performed by the iterator, which determines the current object and can therefore determine the correct object to display when a navigation button is clicked. Action bindings to actions that are not at the iterator level, such as custom methods on an application module or the commit or rollback operations, will not contain this reference.

Action bindings use the `RequiresUpdateModel` property, which determines whether or not the model needs to be updated before the action is executed. In the case of navigation operations, by default this property is set to `true`, which means that any changes made at the view layer must be moved to the model before navigation can occur.

The following example shows the action bindings for the navigation operations.

```
<action IterBinding="CustomersIterator" id="First"
         RequiresUpdateModel="true" Action="first"/>
<action IterBinding="CustomersIterator" id="Previous"
        RequiresUpdateModel="true" Action="previous"/>
<action IterBinding="CustomersIterator" id="Next"
        RequiresUpdateModel="true" Action="next"/>
<action IterBinding="CustomersIterator" id="Last"
        RequiresUpdateModel="true" Action="last"/>
```

Iterator RangeSize Attribute Defined

Iterator bindings have a `rangeSize` attribute that the binding uses to determine the number of data objects to make available for the page for each iteration. This attribute helps in situations when the number of objects in the data source is quite large. Instead of returning all objects, the iterator binding returns only a set number, which then become accessible to the other bindings. Once the iterator reaches the end of the range, it accesses the next set. The following example shows the default range size for the `Customers` iterator.

```
<iterator Binds="Customers" RangeSize="25"
          DataControl="BackOfficeAppModuleDataControl"
          id="CustomersIterator"
          ChangeEventPolicy="ppr"/>
```

> **Note:**
>
> This `rangeSize` attribute is not the same as the `rows` attribute on a table component. For more information, see Table 32-1.

When the iterator binding is initially generated, the `RangeSize` attribute is set to `25`. This means that a user can view 25 objects, navigating back and forth between them, without needing to access the data source. The iterator keeps track of the current object. Once a user clicks a button that requires a new range (for example, clicking the **Next** button on object number 25), the binding object executes its associated method against the iterator, and the iterator retrieves another set of 25 records. The bindings then work with that set. You can change this setting as needed. You can set it to `-1` to have the full record set returned. If the `RangeSize` attribute is not specified, the full record set is returned.

EL Expressions Created to Bind to Operations

When you create command components using data control operations, JDeveloper creates an EL expression that binds a command button's `actionListener` attribute to the `execute` property of the action binding for the given operation.

At runtime an action binding will be an instance of the `FacesCtrlActionBinding` class, which extends the core `JUCtrlActionBinding` implementation class. The `FacesCtrlActionBinding` class adds the following methods:

- `public void execute(ActionEvent event)`: This is the method that is referenced in the `actionListener` property, for example `#{bindings.First.execute}`.

  This expression causes the binding's operation to be invoked on the iterator when a user clicks the button. For example, the **First** command button's `actionListener` attribute is bound to the `execute` method on the `First` action binding.

- `public String outcome()`: This can be referenced in an `Action` property, for example `#{bindings.Next.outcome}`.

  This can be used for the result of a method action binding (once converted to a `String`) as a JSF navigation outcome to determine the next page to navigate to.

  > **Note:**
  >
  > Using the outcome method on the action binding breaks the separation between the view-controller layer and the model layer, so it should be rarely used.

Every action binding for an operation has an `enabled` boolean property that Oracle ADF sets to `false` when the operation should not be invoked. By default, JDeveloper binds the UI component's `disabled` attribute to this value to determine whether or not the component should be enabled. For example, the UI component for the **First** button has the following as the value for its `disabled` attribute:

```
#{!bindings.First.enabled}
```

This expression evaluates to `true` whenever the binding is not enabled, that is, when the operation should not be invoked, thereby disabling the button. In this example, because the framework will set the `enabled` property on the binding to `false` whenever the first record is being shown, the **First** button will automatically be disabled because its `disabled` attribute is set to be `true` whenever `enabled` is `False`. For more information about the `enabled` property, see Oracle ADF Binding Properties.

The following example shows the code generated after dropping the `Create` and `Delete` operations as buttons on a page.

```
<af:button actionListener="#{bindings.Create.execute}" text="Create"
           disabled="#{!bindings.Create.enabled}" id="b5"/>
<af:button actionListener="#{bindings.Delete.execute}" text="Delete"
           disabled="#{!bindings.Delete.enabled}" id="b6"/>
```

What Happens at Runtime: How Action Events and Action Listeners Work

When the user clicks a command component, the form is submitted and then an action event is fired. Action events might affect only the user interface (for example, a link to change the locale, causing different field prompts to display), or they might involve some logic processing in the back end (for example, a button to navigate to the next record). That event object then takes information about the current data object from the iterator and then passes it to the action binding's method, which is bound to the command component through its `actionListener` attribute.

> **Note:**
>
> An action listener is a class that registers to be notified when a command component fires an action event. An action listener contains an action listener method that processes the action event object passed to it by the command component.

For example, when the user clicks a **Delete** button that was created from the ADF `Delete` operation, an action event is fired. This event object stores currency information about the current data object, taken from the iterator. Because the component's `actionListener` attribute is bound to the `execute` method of the `Delete` action binding, the `Delete` operation is invoked when the event fires. This method takes the currency information passed in the event object to determine which data object to delete.

Typically, the `actionListener` property's value is in the form of an EL expression. For example, if the value of the `actionListener` attribute is `{bindings.Delete.execute}`, the `execute()` method of the `Delete` action binding is called.

What You May Need to Know About Overriding Declarative Methods

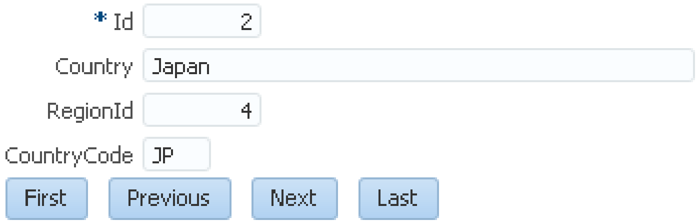
When you drop an operation or method as a command button, JDeveloper binds the button to the execute method for the operation or method. However, there may be occasions when you need to add logic before or after the existing logic. JDeveloper allows you to add logic to a declarative operation by creating a new method and property on a managed bean that provides access to the binding container. By default, this generated code executes the operation or method. You can then add logic before or after this code. JDeveloper automatically binds the command component to this new method, instead of to the execute property on the original operation or method. Now when the user clicks the button, the new method is executed. For more information, see Overriding Declarative Methods.

Incorporating Range Navigation into Forms

You may choose to include navigational controls when you create an ADF Form. By default, JDeveloper creates First, Last, Previous, and Next buttons to navigate within the collection. You can also include navigational buttons manually, as required.

When you create an ADF Form, if you choose to include navigational controls, JDeveloper includes ADF Faces command components bound to existing navigational logic on the data control. This built-in logic allows the user to navigate through all the data objects in the collection. For example,Figure 31-9 shows a form that would be created if you drag the `Countries` collection and drop it as an ADF Form that uses navigation.

Figure 31-9 Navigation in a Form



* Id	2
Country	Japan
RegionId	4
CountryCode	JP

Table 31-1 shows the built-in navigation operations provided on data controls and the result of invoking the operation or executing an event bound to the operation. For more information about action events, see What Happens at Runtime: How Action Events and Action Listeners Work .

Table 31-1 Built-in Navigation Operations

Operation	When invoked, the associated iterator binding will...
First	Move its current pointer to the beginning of the result set.
Last	Move its current pointer to the end of the result set.
Previous	Move its current pointer to the preceding object in the result set. If this object is outside the current range, the range is scrolled backward a number of objects equal to the range size.
Next	Move its current pointer to the next object in the result set. If this object is outside the current range, the range is scrolled forward a number of objects equal to the range size.
Previous Set	Move the range backward a number of objects equal to the range size attribute.
Next Set	Move the range forward a number of objects equal to the range size attribute.

How to Manually Insert Navigation Controls into a Form

By default, when you choose to include navigation when creating a form using the Data Controls panel, JDeveloper creates **First**, **Last**, **Previous**, and **Next** buttons that allow the user to navigate within the collection.

You can also add navigation buttons to an existing form manually.

Before you begin:

It may be helpful to have an understanding of navigation controls. For more information, see Incorporating Range Navigation into Forms.

You may also find it helpful to understand other ADF functionality and features. For more information, see Additional Functionality for Databound Pages.

You will need to complete these tasks:

- Create an application module that contains instances of the view objects that you want in your data model, as described in Creating and Modifying an Application Module. Alternatively, if you are using another type of business service, create a data control for that business service as described in Exposing Business Services with Data Controls in *Developing Applications with Oracle ADF Data Controls*.
- Create a JSF page as described in Creating a Web Page.

To manually add navigation buttons:

1. From the Data Controls panel, select the operation associated with the collection of objects on which you wish the operation to execute, and drag it onto the JSF page.

   For example, if you want to navigate forward through a collection of countries, you would drag the **Next** operation associated with the `Countries` collection. Figure 31-8 shows the operations associated with the `Countries` collection.
2. From the ensuing context menu, choose either **ADF Button** or **ADF Link**.

> **Tip:**
>
> You can also drop the **First**, **Previous**, **Next**, and **Last** buttons at once. To do so, drag the corresponding collection, and from the context menu, choose **Navigation > ADF Navigation Buttons**.

What Happens When You Manually Add Navigation Buttons

When you drop a navigation operation from the Data Controls panel, JDeveloper:

- Defines an action binding in the page definition file for the associated operations
- Configures the iterator binding to use partial page rendering for the collection
- Inserts code in the JSF page for the command components
- Inserts the `partialSubmit` attribute for the component in the JSF page and sets it to `true` in order to fire a partial page request when the button is clicked. For

more information, see the Using Partial Triggers section of *Developing Web User Interfaces with Oracle ADF Faces*.

For more information on code generated for command components, see What Happens When You Create Command Components Using Operations.

The following example shows the code generated on the JSF page for navigation operation buttons.

```
<f:facet name="footer">
   <af:panelGroupLayout>
     <af:button actionListener="#{bindings.First.execute}"
                       text="First"
                       disabled="#{!bindings.First.enabled}"
                       partialSubmit="true" id="cb1"/>
     <af:button actionListener="#{bindings.Previous.execute}"
                       text="Previous"
                       disabled="#{!bindings.Previous.enabled}"
                       partialSubmit="true" id="cb2"/>
     <af:button actionListener="#{bindings.Next.execute}"
                       text="Next"
                       disabled="#{!bindings.Next.enabled}"
                       partialSubmit="true" id="cb3"/>
     <af:button actionListener="#{bindings.Last.execute}"
                       text="Last"
                       disabled="#{!bindings.Last.enabled}"
                       partialSubmit="true" id="cb4"/>
   </af:panelGroupLayoutr>
 </f:facet>
```

What Happens at Runtime: Navigation Controls

When the user clicks a navigation control button, an action event is fired. That event object then takes information about the current data object from the iterator and then passes it to the operation's method, which is bound to the button through its `actionListener` attribute.

For more information, see What Happens at Runtime: How Action Events and Action Listeners Work .

In addition, when a user clicks a navigation button, only those components associated with the same iterator as the button's action binding are processed through the lifecycle. For more information, see What You May Need to Know About Partial Page Rendering and Iterator Bindings.

What You May Need to Know About the Browser Back Button and Navigating Through Records

You must use the navigation buttons to navigate through the records displayed in a form; you cannot use the browser's back or forward buttons. Because navigation forms automatically use PPR, only part of the page goes through the lifecycle, meaning that when you click a navigation button, the components displaying the data are refreshed and display new data, and you actually remain on the same page. Therefore, when you click the browser's back button, you will be returned to the page that was rendered before the page with the form, instead of to the previous record displayed in the form.

For example, say you are on a page that contains a link to view all current orders. When you click the link, you navigate to a page with a form and the first order, Order #101, is displayed. You then click **Next** and Order #102 is displayed. You click **Next** again, and Order #103 is displayed. If you click the browser's back button, you will not be shown Order #102. Instead, you will be returned to the page that contained the link to view all current orders.

What You May Need to Know About the CacheResults Property

When you create a navigable form using the Data Controls panel, the `CacheResults` property on the associated iterator is set to `true`. This ensures that the iterator's state, including currency information, is cached between requests, allowing it to determine the current object. If this property is set to `false`, navigation will not work.

Creating a Form to Edit an Existing Record

Edit forms allow you to make amendments to the existing data in the form. In order to create an edit form, you need to drop a collection on your page as a form and then drop the appropriate operations as command components.

You can create a form that allows a user to edit the current data, and then commit those changes to the data source. To do this, you use operations that can modify data records associated with the collection or the data control itself to create command buttons. For example, you can use the `Delete` operation to create a button that allows a user to delete a record from the current range.

For data controls based on ADF Business Components or on EJB session beans that have an explicit commit model, it is important to note that these operations are executed only against objects in the ADF cache. You need to use the `Commit` operation on the root data control to actually commit any changes to the data source. You use the data control's `Rollback` operation to roll back any changes made to the cached object. If the page is part of a transaction within a bounded task flow, you would most likely use these operations to resolve the transaction in a task flow return activity. See Managing Transactions in Task Flows.

How to Create an Edit Form

To create an edit form, you drop a collection on your page as a form and then drop the appropriate operations as command components.

Before you begin:

It may be helpful to have an understanding of creating edit forms. For more information, see Creating a Form to Edit an Existing Record.

You may also find it helpful to understand other ADF functionality and features. For more information, see Additional Functionality for Databound Pages.

You will need to complete these tasks:

- Create an application module that contains instances of the view objects that you want in your data model, as described in Creating and Modifying an Application Module. Alternatively, if you are using another type of business service, create a data control for that business service as described in Exposing Business Services with Data Controls in *Developing Applications with Oracle ADF Data Controls*.

- Create a JSF page as described in Creating a Web Page.

To create an edit form:

1. From the Data Controls panel, drag the collection for which you wish to create the form, and choose **ADF Form** from the context menu.

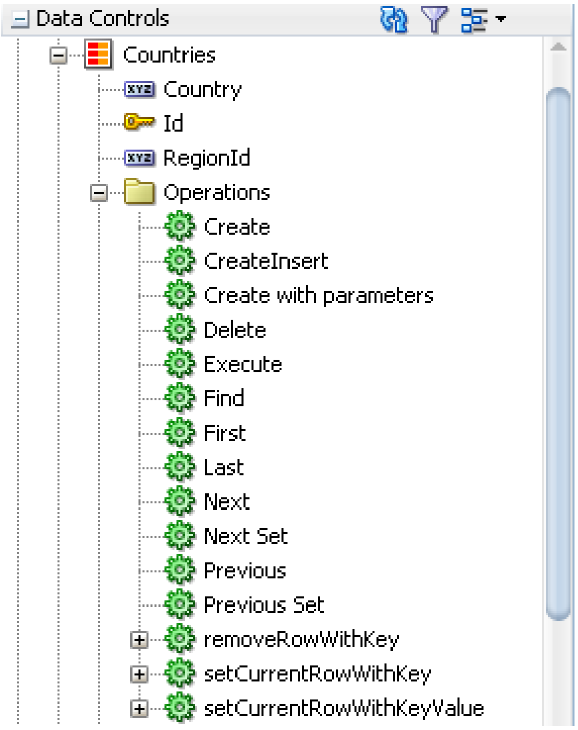
   This creates a form using `inputText` components, which will allow the user to edit the data in the fields.

2. In the Create Form dialog, set the fields and buttons that you would like added to the form and click **OK**.

3. Optionally, from the Data Controls panel, select any additional operation associated with the collection that you would like to add to the form, drag it to the page, and choose **ADF Button** or **ADF Link** from the context menu.

   For example, if you want to be able to delete a country, you would select the `Delete` operation associated with the `Countries` collection. Figure 31-10 shows the operations associated with the `Countries` collection.

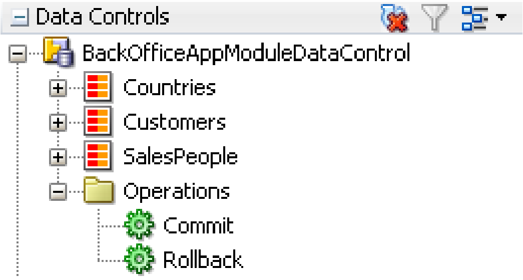
   Figure 31-10   Operations Associated with a Collection



4. Add operations to the page to save or cancel changes.

   If the page is not part of a transaction within a bounded task flow, then you need to create buttons that allow the user either to commit or roll back the changes. From the Data Controls panel, drag the **Commit** and **Rollback** operations associated with the root-level data control, and drop them as either a command button or a command link. Figure 31-11 shows the commit and rollback operations for the `BackOfficeAppModuleDataControl` data control.

Figure 31-11 Commit and Rollback Operations for a Data Control



If the page is part of a transaction within a bounded task flow, then you can simply enter `Commit` and `Rollback` as the values for the transaction resolution when creating the task flow return activity. For more information, see Managing Transactions in Task Flows.

If you are basing the form on a data control that is based on business services other than ADF Business Components, the `Commit` and `Rollback` operations might not be available. For more information, see What You May Need to Know About Working With Data Controls for Stateless Business Services.

5. Optionally, change the `autoSubmit` property of each editable field to `true`.

   Doing so enables any user changes to be submitted to the page when tabbing out of the field and thus enabling the `Commit` and `Rollback` operations, if they have not already been enabled by another sequence of events, such as a change to a field plus a button click.

   You can access the `autoSubmit` property for a field by selecting the field and then scrolling to the **Behavior** section in the Properties window.

> **Tip:**
>
> If the same page on which you create your edit form also has a table created from the same collection as the form, the table and the form use the same iterator binding, which enables the user to select the record to edit in the form by selecting a row in the table. For more information on creating databound tables, see Creating a Basic Table.

What Happens When You Create Edit Forms

For information on what happens when you drop a collection on to a page, see What Happens When You Create a Form Using a Data Control Collection.

Dropping any data control operation as a command button causes code generation similar to when you drop navigation operations. For more information, see What Happens When You Create Command Components Using Operations.

The only difference is that the action bindings for the `Commit` and `Rollback` operations do not require a reference to the iterator, because they execute a method on the application module (the data control itself), as opposed to the iterator. Note that the `Rollback` action has the `RequiresUpdateModel` property set to `false`. This is because the model should not be updated before the operation is executed, since all changes need to be discarded. The following example shows the action bindings generated in the page definition file for these operations.

```
<action id="Commit" RequiresUpdateModel="true" Action="commitTransaction"
        DataControl="BackOfficeAppModuleDataControl"/>
<action id="Rollback" RequiresUpdateModel="false"
        Action="rollbackTransaction"
        DataControl="BackOfficeAppModuleDataControl"/>
```

Table 31-2 shows the built-in non-navigation operations provided on data controls and data control objects, along with the result of invoking the operation or executing an event bound to the operation. For more information about action events, see What Happens at Runtime: How Action Events and Action Listeners Work .

Table 31-2 More Built-in Operations

Operation	When invoked, the associated iterator binding will...
`CreateInsert`	Creates a row directly before the current row, inserts the new record into the row set, then moves the current row pointer to the new row. Note that the range does not move, meaning that the last row in the range may now be excluded from the range. For more information about using the `CreateInsert` operation to create objects, see Creating an Input Form. **Note:** `CreateInsert` is only available for data controls based on ADF Business Components application modules.
`Create`	Creates a row directly before the current row, then moves the current row pointer to the new row. Note that the range does not move, meaning that the last row in the range may now be excluded from the range. For data controls based on application modules and most other business services, the record will not be inserted into the row set, preventing a blank row should the user navigate away without actually creating data. The new row will be created when the user submits the data. For more information, see What You May Need to Know About Create and CreateInsert. However, for JPA-based data controls, the `Create` operation does insert data into the row set and can be configured to persist the data. For more information, see Data Control Built-in Operations in *Developing Applications with Oracle ADF Data Controls*.
`CreateWith Parameters`	Same as the `CreateInsert` operation (the new record is inserted into the row set). However, it uses named parameters to create the object. **Note:** `Create With Parameters` is only available for data controls based on ADF Business Components application modules.
`Delete`	Deletes the current row from the cache and moves the current row pointer to the next row in the result set. Note that the range does not move, meaning that a row may be added to the end of the range. If the last row is deleted, the current row pointer moves to the preceding row. If there are no more rows in the collection, the enabled attribute is set to `disabled`.
`RemoveRowWithKey`	Uses the row key as a `String` converted from the value specified by the input field to remove the data object in the bound data collection.

Table 31-2 (Cont.) More Built-in Operations

Operation	When invoked, the associated iterator binding will...
SetCurrentRowWithKey	Sets the row key as a String converted from the value specified by the input field. The row key is used to set the currency of the data object in the bound data collection. For an example of when this is used, see What You May Need to Know About Setting the Current Row in a Table.
SetCurrentRowWithKeyValue	Sets the current object on the iterator, given a key's value. For more information, see What You May Need to Know About Setting the Current Row in a Table.
ExecuteWithParams	Refreshes the data collection by first assigning new values to the named bind variables passed as parameters, then (re)executing the view object's query. You would use this operation in the same manner as you would use the CreateInsert operation to create an input form. For more information, see Creating an Input Form. This operation appears only for view objects that have defined one or more named bind variables at design time. For more information, see Working with Bind Variables. For EJB and bean data controls, this operation is only available on data control collection objects that are based on parameterized queries.
Commit	Causes all items currently in the cache to be committed to the database.
Rollback	Clears the cache and returns the transaction and iterator to the initial state. Resets the ActionListener method.
Execute and Find	These operations are used only in search forms. See Creating ADF Databound Search Forms for more information.

What You May Need to Know About Working With Data Controls for Stateless Business Services

When you create user interfaces based on ADF Business Components services, any data changes that a user makes are not propagated to the underlying database until the user triggers the Commit operation through a command component.

For user interfaces that are based on data controls for stateless business services, such as stateless EJB session beans or RESTful web services, the Commit and Rollback operations are not available. In such cases, you can use custom methods to handle interactions with the underlying database. In the case of EJB data controls, any persistEntity and mergeEntity methods in the session bean are exposed through the Data Controls panel, and you can create command components from them. For more information, see About Commit Models for EJB Session Beans in *Developing Applications with Oracle ADF Data Controls*.

Using Parameters to Create a Form

You can create ADF parameter forms using different mechanisms. You need to gather information about the creation of the ADF Parameter Form using ExecuteWithParams operation.

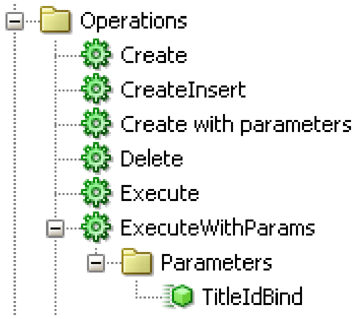
In Oracle ADF, there are several ways that you can create a form where the record that is displayed is determined by an input parameter. The following are some of those mechanisms:

- Create an ADF Parameter Form based on a data control collection's `ExecuteWithParams` operation or on a custom method that takes parameters.
- Pass a parameter to the form through a task flow.
- Nest a `setPropertyListener` tag within a command component and provide application logic within a managed bean to access the parameters and invoke any needed logic.

This section covers the creation of ADF parameter forms based on the `ExecuteWithParams` operation. For information on passing parameters to a form through a task flow, see Using Parameters in Task Flows and Specifying Parameters for an ADF Region. For information on using the `setPropertyListener` tag, see Setting Parameter Values Using a Command Component.

The `ExecuteWithParams` operation is available only for data control objects that contain parameters, such as those based on view objects containing bind variables and those based on JPA-based beans that contain named queries with parameters. For example, the Summit ADF sample application contains a view object instance called `SalesPeople`, which is based on the `EmpVO` view object, which contains the bind variable `TitleIdBind`. So, as shown in Figure 31-12, the `SalesPeople` object contains an `ExecuteWithParams` operation with a subnode for the `TitleIdBind` parameter.

Figure 31-12 ExecuteWithParams Operation in the Data Controls Panel



View object bind variables and JPA parameterized queries allow you to supply attribute values at runtime to the view object or **view criteria**. The `ExecuteWithParams` operation refreshes the data collection by first assigning new values to the named bind variables passed as parameters, then (re)executing the view object's query. For more information about bind variables in view objects, see Working with Bind Variables. For more information on named parameters in JPA queries, see EJB Data Control Prerequisites and Considerations in *Developing Applications with Oracle ADF Data Controls*.

How to Create a Form Using Parameters

To create a form using parameters, you drop the `ExecuteWithParams` operation of a collection on to a page as an ADF Parameter Form and then drop the collection itself as an ADF Form.

Before you begin:

It may be helpful to have an understanding of creating parameter forms. For more information, see Using Parameters to Create a Form.

You may also find it helpful to understand other ADF functionality and features. For more information, see Additional Functionality for Databound Pages.

You will need to complete these tasks:

- Create an application module that contains instances of the view objects that you want in your data model, as described in Creating and Modifying an Application Module.

  Alternatively, if you are using another type of business service, create a data control for that business service as described in Exposing Business Services with Data Controls in *Developing Applications with Oracle ADF Data Controls*.
- Add bind variables to the view criteria definition as described in How to Add WHERE Clause Bind Variables to a View Object Definition.

  For JPA-based data controls you can create named queries in the session bean or service facade that specify named parameters. For more information, see EJB Data Control Prerequisites and Considerations in *Developing Applications with Oracle ADF Data Controls*.
- Create a JSF page as described in Creating a Web Page.

To create a parameter form based on the ExecuteWithParams operation:

1. From the Data Controls panel, drag the `ExecuteWithParams` operation for the collection for which you wish to create the form, and choose **ADF Parameter Form** from the context menu.
2. In the Create Form dialog, select the parameters on which you want users to be able to query and click **OK**.
3. From the Data Controls panel, drag the corresponding collection, and drop it as the type of table or form that you want to display the results of the query based on the user input.

> **Note:**
>
> You can also declaratively incorporate selection lists into parameter forms. To create a selection list, drag a parameter node that is a child of the `ExecuteWithParams` operation on to the form. Then drag the `ExecuteWithParams` operation and drop it as a button.

What Happens When You Create a Parameter Form

When you drop the `ExecuteWithParams` operation as a parameter form, JDeveloper:

- Inserts code in the JSF page for the form using `inputText` components bound to the attribute bindings, and a `button` component bound to the `ExecuteWithParams` operation.
- Defines the following in the page definition file:

- A variable iterator for the attributes to use to access the variables (as opposed to a collection's iterator which is used for other forms).
- A variableUsage variable for each parameter to hold the data values. These variables inherit the default value and UI control hints from the view object named bind variables to which they are bound. The variables are local, meaning they live only during a single request. Though they are carried across subsequent post-backs to the same form, they are forgotten (and re-initialized) when a user navigates to some other page.
- An action binding for the operation that contains NamedData elements for each parameter. Each NamedData element is bound to a corresponding variable. These bindings allow the operation to access the value for the parameter upon execution.
- Attribute bindings for the associated attributes.

The following example shows the executables and bindings in the page definition file that are created by dropping the SalesPeople collection's ExecuteWithParams operation on a page.

```
<executables>
  <variableIterator id="variables">
    <variableUsage DataControl="BackOfficeAppModuleDataControl"
                   Binds="SalesPeople.variablesMap.TitleIdBind"
                   Name="ExecuteWithParams_TitleIdBind" IsQueriable="false"/>
  </variableIterator>
  <iterator Binds="SalesPeople" RangeSize="25"
            DataControl="BackOfficeAppModuleDataControl"
            id="SalesPeopleIterator"/>
</executables>
<bindings>
  <action IterBinding="SalesPeopleIterator" id="ExecuteWithParams"
          RequiresUpdateModel="true"
          Action="executeWithParams">
    <NamedData NDName="TitleIdBind" NDType="java.lang.Integer"
               NDValue="${bindings.ExecuteWithParams_TitleIdBind}"/>
  </action>
  <attributeValues IterBinding="variables" id="TitleIdBind">
    <AttrNames>
      <Item Value="ExecuteWithParams_TitleIdBind"/>
    </AttrNames>
  </attributeValues>
</bindings>
```

When you drop the collection that will be used to display the results of the parameter form, code is generated for a basic form, as described in What Happens When You Create a Form Using a Data Control Collection. In addition, if you use a table to display the results, the partialTriggers property on table is set to the command button in the parameter form.

What Happens at Runtime: How Parameters are Populated

When the page is rendered, the framework checks if the page is being rendered for the first time and if the exception list has any items. If both are false, then the framework executes the ExecuteWithParams action binding. This action binding executes the query, taking the values of the named data elements as values for the needed parameters.

Creating an Input Form

An input form is similar to an edit form that helps you to enter information for a new record. In order to create a basic input form, you need to first create an edit form and add a command component for creating new records.

You can create a form that allows a user to enter information for a new record and then commit that record to the data source. Creating a basic input form is similar to creating an edit form, except that the input form also contains the `CreateInsert` operation, which causes a blank row to be inserted into the row set which the user can then populate using a form.

You can create a simple input form by dragging and dropping a collection and the appropriate operations from the Data Controls panel on to a page. However, you might need to employ a selection of other techniques to control the user workflow and insert other form processing logic. Such techniques may include:

- Using parameters in ADF task flows to pass values with which to pre-populate given attributes of a new record. For more information, see Passing Parameters to a View Activity.
- Using task flow return activities to set the transaction boundary in the UI, as shown in Using Task Flow Return Activities .
- Embedding operations in task flow method calls, as shown in Passing Parameters to a Bounded Task Flow.
- Using managed beans to carry out some operations as shown in Overriding Declarative Methods.
- Incorporating the form into a region in a task flow as shown in Using Task Flows as Regions .
- Incorporating the form into a multi-step flow, as shown in Using Train Components in Bounded Task Flows.
- Placing the form inside a dialog as shown in Running a Bounded Task Flow in a Modal Dialog.
- Using save points as described inManaging Transactions in Task Flows.

The following section shows how to create a basic input form and provides information on a couple of variants.

How to Create an Input Form

Creating a basic input form involves creating an edit form and adding a command component for creating new records.

Before you begin:

It may be helpful to have an understanding of input forms. For more information, see Creating an Input Form.

You may also find it helpful to understand other ADF functionality and features. For more information, see Additional Functionality for Databound Pages.

You will need to complete this task:

Create an edit form as described in How to Create an Edit Form.

To create an input form:

1. In the Data Controls panel, expand the node for the collection object on which the form is based.
2. Expand the collection's **Operations** node and drag the `CreateInsert` operation to the edit form.

> **Note:**
>
> The `CreateInsert` operation is available only for data controls based on ADF Business Components application modules. If your data control is based on another business service, use the `Create` operation instead. For more information on the differences between the two, see Table 31-2.

Creating an Input Form Using a Bounded Task Flow

You can use a task flow to provide the workflow for input forms. Instead of using command components to create new objects, you use a method call activity in the task flow or a method in a managed bean. Instead of using a command component to commit changes, you use a task flow return activity.

For example, you can use a method call activity in a task flow to call a data control object's `CreateInsert` operation and then pass control to a view activity that displays a form where the user can enter the data.

To create an input form within a bounded task flow:

1. Create a bounded task flow as described in Creating a Task Flow.
2. In the Properties window, set the **Transaction** property to **Always Begin New Transaction**.

   For more information, see Managing Transactions in Task Flows.
3. In the bounded task flow, add a method call activity.

   For more information, see Using Method Call Activities.
4. From the Data Controls panel, drag the `CreateInsert` operation associated with the collection for which you are creating the form and drop it on the method activity.
5. With the method call activity still selected, in the Properties window, enter a string for the `fixed-outcome` property. After dropping the `CreateInsert` operation on to the method call activity, this value is set to `CreateInsert`, but you may want to change it to something more specific.
6. In the overview editor for the task flow, add a view activity that represents the page for the input form. For information on adding view activities, see Adding Activities to a Task Flow.
7. Add a control flow case from the method activity to the view activity.
8. With the control flow case still selected in the design editor, in the Properties window, make sure that the `from-outcome` property of the control flow case is the

same as the value of the fixed-outcome property of the method activity set in Step 5.

9. Open the page for the view activity in the design editor, and from the Data Controls panel, drag the collection for which the form will be used to create a new record, and choose **ADF Form** from the context menu.
10. In the task flow, add a return activity. This return activity must execute the commit operation on the data control. For these procedures, see Using Task Flow Return Activities .

Using a Commit Button Instead of a Return Activity Within a Task Flow

Generally you should use a return activity to call the Commit operation. However, there might be cases where you need to use a command component for the Commit operation, such as when the task flow contains data managed by more than one data control or you need to commit the data before the end of the flow.

To create an input form that has a Commit button within the form:

1. Create an input form as described in Creating an Input Form Using a Bounded Task Flow, but without setting the task flow return activity.
2. In the Data Controls panel, drag the Commit operation for the data control that contains the collection associated with the input form, and drop it as a command button.
3. In the Structure window, select the command button for the commit operation.
4. In the Properties window, set the action to the outcome string that will navigate back to the method activity. You then need to add a control flow case from the page back to the activity, using the same outcome value.
5. Set the command button's disabled property to false.

   By default, JDeveloper binds the disabled attribute of the button to the enabled property of the binding, causing the button to be disabled when the enabled property is set to false. For this binding, the enabled property is false until an update has been posted. For the purposes of an input form, the button should always be enabled, since there will be no changes posted before the user needs to create the new object.

Creating an Input Form That Allows Multiple Entries in a Single Transaction

If you want the user to be able to create multiple entries before committing to the database, do the following:

To enable an input form to allow multiple entries in a single transaction:

1. Create an input form as described in Creating an Input Form Using a Bounded Task Flow
2. In the task flow, add another control flow case from the view activity back to the method activity, and enter a value for the from-outcome method. For example, you might enter createAnother.
3. Drag and drop a command component from the Components window onto the page, and set the action attribute to the from-outcome just created. This will cause the task flow to return to the method activity and reinvoke the CreateInsert operation.

What Happens When You Create an Input Form

When you use an ADF Form to create an input form, JDeveloper:

- Creates an iterator binding for the collection and an action binding for the `CreateInsert` operation in the page definition for the method activity. The `CreateInsert` operation is responsible for creating a row in the row set and populating the data source with the entered data. In the page definition for the page, JDeveloper creates an iterator binding for the collection and attribute bindings for each of the attributes of the object in the collection, as for any other form. If you created command buttons or links using the `Commit` and `Rollback` operations, JDeveloper also creates action bindings for those operations.
- Inserts code in the JSF page for the form using ADF Faces `inputText` components, and in the case of the operations, `button` components.

What Happens at Runtime: CreateInsert Action from the Method Activity

When the `CreateInsert` button is clicked (or a method corresponding to that operation is called), the `CreateInsert` action binding is invoked, which executes the `CreateInsertRow` action, and a new blank instance for the collection is created.

> **Note:**
>
> When you use a task flow method activity to the call the `CreateInsert` operation, the method activity's binding container skips validation for required attributes during routing from the method activity to the view activity, allowing the blank instance to be displayed in the form on the page.

What You May Need to Know About Displaying Sequence Numbers

Because the `Create` action is executed before the page is displayed, if you are populating the primary key using sequences, the next number in the sequence will appear in the input text field, unlike the rest of the fields, which are blank. The sequence number is displayed because the associated entity class contains a method that uses an eager fetch to generate a sequence of numbers for the primary key attribute. The eager fetch populates the value as the row is created. Therefore, using sequences works as expected with input forms.

However, if instead you've configured the attribute's type to `DBSequence` (which uses a database trigger to generate the sequence), the number would not be populated until the object is committed to the database. In this case, the user would see a negative number as a placeholder. To avoid this, you can use the following EL expression for the `Rendered` attribute of the input text field:

```
#{bindings.EmployeeId.inputValue.value > 0}
```

This expression will display the component only when the value is greater than zero, which will not be the case before it is committed. Similarly, you can simply set the

`Rendered` attribute to `false`. However, then the page will never display the input text field component for the primary key.

What You May Need to Know About Input Forms Backed By Stateless Services

If the data control on which your input form is based is a data control for a stateless business service, there is no built-in `Commit` operation for you to use to save changes back to the data source.

Instead, you need to use a custom method to save to the database. For more information, see What You May Need to Know About Working With Data Controls for Stateless Business Services .

Creating a Form with Dynamic Components

A form can be created dynamically. ADF Faces dynamic components are used to create a form that can display different data each time you run it.

Instead of creating static databound forms where you provide tags for each component directly in the page, you can use a dynamic component to create forms where the binding metadata and the components used to display the bound content are determined at runtime.

This section provides information on creating databound dynamic forms. You can also use the dynamic component to create databound tables. See Creating a Table with Dynamic Components.

For information on the dynamic form component, including how to create a custom data model, see Determining Components at Runtime in *Developing Web User Interfaces with Oracle ADF Faces*.

About Dynamic Components

ADF Faces provides the `af:dynamicComponent` tag that you can use to create model-driven forms and tables where the binding metadata and the tags used to display the bound content are determined at runtime.

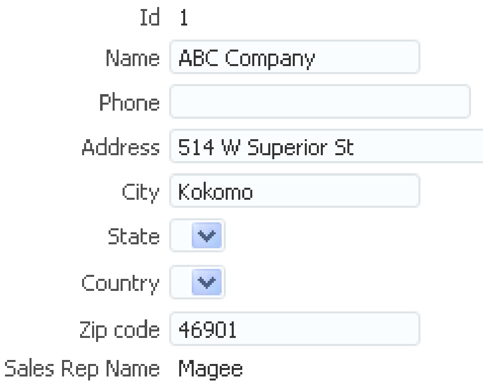
The dynamic building of the bindings provides the following possibilities:

- Enables you to create pages where the fields displayed and the ADF Faces components used to display them are determined by the data model. Any changes to the data model, such as additional columns or changed UI hints are reflected in the page without having to redesign the page. For more information about UI hints on view objects, see Defining UI Hints for View Objects.
- Enables you to create pages based on polymorphic view objects, in which the fields available for a given record can differ depending on the record's base entity object. Since the fields to display are determined at runtime, you do not have to include fields in the page that may not apply to certain records or do coding in the view layer to adjust the fields that are displayed for a given record. For more information about building business components that work with multiple row types, see Defining Polymorphic View Objects.

You set display information using UI hints on a view object instead of configuring the information in the Create Form or Create Table dialog when you drop the control onto the page. Then if you want to change how the data displays, you need only change it on the view object, and all dynamic components bound to that view object will change their display accordingly.

Figure 31-13 shows a dynamic form at runtime that was designed by setting UI hints for attributes on the `CustomerVO` view object and then dropping the `Customers` data control collection as a dynamic form. Among the UI hints set were `LABEL` and `DISPLAYHINT` (the latter of which can be set to `Hide` in order to not include the given attribute in the dynamic form).

Figure 31-13 Dynamic Form Displays Based on Hints Set on the View Object



Id	1
Name	ABC Company
Phone	
Address	514 W Superior St
City	Kokomo
State	
Country	
Zip code	46901
Sales Rep Name	Magee

The following ADF Faces components can be rendered at runtime when you use `af:dynamicComponent` at design time:

- `af:inputText`
- `af:inputDate`
- `af:inputListOfValues`
- `af:selectOneChoice`
- `af:selectManyChoice`
- `af:selectOneListbox`
- `af:selectManyListbox`
- `af:selectOneRadio`
- `af:selectBooleanRadio`
- `af:selectBooleanCheckbox`
- `af:selectManyCheckbox`

How to Create a Dynamic Form

To create a dynamic form, you drop a collection from the data controls panel as an ADF Form and specify that the fields be generated dynamically.

Before you begin:

It may be helpful to have an understanding of dynamic forms. See Creating a Form with Dynamic Components.

If you want to take advantage of dynamic forms to display records that contain varying attributes, you first need to have data objects that contain that capability. For information on creating view objects with this capability, see Defining Polymorphic View Objects.

You may also find it helpful to understand other ADF functionality and features. See Additional Functionality for Databound Pages.

You will need to complete these tasks:

- Create an application module that contains instances of the view objects that you want in your data model, as described in Creating and Modifying an Application Module. Alternatively, if you are using another type of business service, create a data control for that business service as described in "Exposing Business Services with Data Controls" in *Developing Applications with Oracle ADF Data Controls*.
- Create a JSF page as described in Creating a Web Page.

UI hints determine things such as the type of UI component to use to display the attribute, the label, the tooltip, whether the field should be automatically submitted, etc. You can also determine whether a given attribute is displayed or hidden. For the procedure to create UI hints, see Defining UI Hints for View Objects.

To create a dynamic form:

1. Set UI hints on the view object or data control structure definition file that corresponds to the collection for which you are creating the dynamic form.
2. Optionally, for the `Category` hint on each view object attribute, specify a category or click the **New Category** icon to create a new category.

   Any attributes that have the same category can be grouped together in dynamic forms that you create based on that view object.
3. Optionally, for any categories that you have created set the UI hints for the category's label and tooltip. These can be set in the **UI Categories** tab of the view object's overview editor.
4. From the Data Controls panel, select the collection that represents the view object.
5. Drag the collection onto the page, and from the context menu, choose **ADF Form**.
6. In the Create Form dialog, select the **Fields Generated at Runtime** checkbox.
7. If you have specified any categories for attributes, select the **Include Field Groups** checkbox in order for the form to group the attributes according to those categories.

What Happens When You Create Dynamic Forms

When you drop a collection as a dynamic form on a page, the following things happen:

- The page definition is populated with a `variableIterator` binding, an `iterator` binding to the iterator, and a `tree` value binding.
- The JSF page is populated with one or more `af:iterator` and `af:dynamicComponent` tags. In addition, if the **Include Field Groups** option is selected, `af:switcher` and `af:group` tags are added.

Bindings Created for the Dynamic Form

The page definition file for a dynamic form contains the following executables and bindings:

- **variableIterator**: an internal iterator that contains variables declared for the binding container.
- **iterator**: the iterator binding for the collection. For more information on iterator bindings, see Iterator Bindings Created in the Page Definition File.
- **tree**: the value binding for the attributes of the collection. Unlike a standard form, which has individual value bindings for each attribute set at design time, a dynamic form uses a single tree value binding to encompass the attributes that are exposed at runtime.

  Tree value bindings are also used for databound tables (both standard and dynamic). For more information, see Iterator and Value Bindings for Tables.

The following example shows the bindings for a page that contains a dynamic form component based on the `Customers` collection.

```
<executables>
  <variableIterator id="variables"/>
  <iterator
Binds="SummitAppModuleDataControl.dataProvider.BackOfficeAM.Customers"
            DataControl="SummitAppModuleDataControl" RangeSize="25"
id="CustomersIterator"/>
</executables>
<bindings>
  <tree IterBinding="CustomersIterator" id="Customers">
    <nodeDefinition DefName="oracle.summit.model.views.CustomerVO"
Name="Customers0"/>
  </tree>
</bindings>
```

Tags Created for a Dynamic Form without Grouping

In the JSF page, JDeveloper inserts an `af:iterator` tag, within which it nests an `af:dynamicComponent` tag. The `af:iterator` tag loops through all of the attributes that are exposed by the collection at runtime and uses the `af:dynamicComponent` tag to render the appropriate component for each attribute on the page.

The following example shows the code that is generated when you drop the `Customers` data control object as a dynamic form (but do not select the **Include Field Groups** option).

```
<af:panelFormLayout id="pfl1">
    <af:iterator id="i1" value="#{bindings.Customers.attributesModel.attributes}"
                 var="attr">
        <af:dynamicComponent id="d2" attributeModel="#{attr}">
                   value="#{bindings[attr.containerName]
[attr.name].inputValue}"/>
    </af:iterator>
</af:panelFormLayout>
```

The `value` attribute of the `af:iterator` tag uses an EL expression that evaluates to the `attributesModel.attributes` property of the collection's tree binding. The `attributesModel` property is used to retrieve the data object's attributes and their metadata, such as component type, label, tooltip, and other properties of the real component to be rendered. The `attributes` property of `attributesModel` signifies that a flat (unhierarchical) list of displayable attributes and their metadata is provided.

The `attributeModel` attribute of the `dynamicComponent` tag is bound to the EL expression `#{attr}`, which references the variable that is defined in the iterator's `var` attribute and that serves as a pointer to the current attribute of the data control collection and its corresponding metadata. The EL expression for the `dynamicComponent`'s `value` attribute also references the variable `attr`.

Tags Created for a Dynamic Form with Grouping

If you have selected the **Include Field Groups** checkbox in the Create Form dialog, the generated JSF page includes the `af:switcher` and `af:group` tags, in addition to the tags described in Tags Created for a Dynamic Form without Grouping.

The `af:switcher` tag is nested directly within a an `af:iterator` tag. Within the `af:switcher` tag are nested `f:facet` tags named `GROUP` and `ATTRIBUTE`. The `GROUP` facet contains an `af:group` tag, within which is an `afoutputText` tag to display the group name and an `iterator` tag, which contains an `af:dynamicComponent` tag. The `ATTRIBUTE` facet only contains an `af:dynamicComponent` tag.

For each attribute that the high-level `iterator` iterates over, the switcher dynamically determines whether to render a group or a component for the individual attribute. If it renders a group, the iterator within the group then is used to render the components for the attributes within that group.

The following example shows the code that is generated if you create a dynamic form based on the `Customers` collection and choose to include field groups.

```
<af:panelFormLayout id="pfl1">
    <af:iterator id="i1"

value="#{bindings.Customers.attributesModel.hierarchicalAttributes}"
              var="attr">
        <af:switcher id="sw1" facetName="#{attr.descriptorType}"
                     defaultFacet="ATTRIBUTE">
            <f:facet name="GROUP">
                <af:group title="#{attr.label}" id="g1">
                    <af:outputText value="#{attr.name}" id="ot1"/>
                    <af:iterator id="gi1" value="#{attr.descriptors}"
                                 var="nestedAttr">
                        <af:dynamicComponent id="gd1"
                                             attributeModel="#{nestedAttr}"/>
                    </af:iterator>
                </af:group>
            </f:facet>
            <f:facet name="ATTRIBUTE">
                <af:dynamicComponent id="ad1" attributeModel="#{attr}"/>
            </f:facet>
        </af:switcher>
    </af:iterator>
</af:panelFormLayout>
```

The `value` attribute of the `af:iterator` tag uses an EL expression that evaluates to the `attributesModel.hierarchicalAttributes` property of the collection's tree binding. The `attributesModel` property is used to retrieve the data object's attributes and their metadata, such as component type, label, tooltip, and other properties of the real component to be rendered. The `hierarchicalAttributes` property signifies that a hierarchical list of displayable attributes and their metadata is provided, including any categories that have been set for any attributes in the UI hints.

The `attributeModel` attribute of the `af:dynamicComponent` tag is set to the EL expression `#{attr}`, which references the variable that is defined in the iterator's `var` attribute and that serves as a pointer to the current attribute (or category) of the data control collection and its corresponding metadata.

What Happens at Runtime: How Attribute Values Are Dynamically Determined

When a page with dynamic components is rendered, the bindings are created just as they are when items are dropped from the Data Controls panel at design time, except that they are created at runtime.

At runtime, the `iterator` iterates over the set of attributes and instantiates a `dynamicComponent` for each attribute. The dynamic component uses its `attributeModel` property to obtain information from the iterator about the current attribute, such as label and control type. Each `dynamicComponent` then renders a component based on the metadata that is returned for the attribute.

If the `DISPLAYHINT` UI hint for an attribute is set to `Hide`, no component for the attribute is rendered.

If the dynamic form is created from a data control object that is based on a polymorphic view object, the rendered components depend on the row type for the currently selected record.

How to Apply Validators and Converters to a Dynamic Component

By default, standard ADF Faces converters and validators are applied to the components generated at runtime when using dynamic components. For example, `converterDateTime` converters are applied to components based on attributes of type `TimeStamp`.

If you need to add more specific validation or conversion to a component, you can do so by adding ADF Faces validator and converter tags to the source for the dynamic component and use the tags' `disabled` attribute to determine on which attributes the validators and converters are applied.

Before you begin:

It may be helpful to have an understanding of dynamic forms. For more information, see Creating a Form with Dynamic Components.

You may also find it helpful to understand other ADF functionality and features. For more information, see Additional Functionality for Databound Pages.

You will need to complete this task:

> Create a dynamic form as shown in How to Create a Dynamic Form or a dynamic table as shown in How to Create a Dynamic Table.

To manually insert an ADF Faces validator or converter into a dynamic form:

1. Open the page containing the dynamic component in the source editor.

2. Within the `af:dynamicComponent` tag, insert any validator and converter components that you want to use.
3. Within the each validator and converter tag, insert the `disabled` attribute and bind it to an EL expression that disables the tag for all attributes except for those to which you want to apply it.

   For example, the following expression would enable a validator only for the `Hiredate` attribute:

   ```
   #{attr.name == 'Hiredate' ? false : true}
   ```

The following example shows a dynamic component with two validators and two converters applied.

```
<af:panelFormLayout id="pfl1">
  <af:iterator id="i1" value="#{bindings.EmpVO1.attributesModel.attributes}"
                     var="attr">
    <af:dynamicComponent
                  value="#{bindings[attr.containerName][attr.name].inputValue}"
                  attributeModel="#{attr}" id="dc1"/>
      <af:convertDateTime disabled="#{attr.name == 'Hiredate' ? false : true}"
                          pattern="yyyy/MM/dd"/>
      <af:convertNumber disabled="#{attr.name == 'Sal' ? false : true}"
                        pattern="#,###,###" />
      <af:validateLength disabled="#{attr.name == 'Job' ? false : true}"
                         maximum="10" hintMaximum="maxmum length is 10"/>
      <af:validateLongRange disabled="#{attr.name == 'Sal' ? false : true}"
                            minimum="1000"/>
    </af:dynamicComponent>
  </af:iterator>
</af:panelFormLayout>
```

What You May Need to Know About Mixing Dynamic Components with Static Components

When you create a dynamic form, you essentially create a block of components that are rendered dynamically. It is possible place static components before and after that block of components, but you can not intersperse static components within that block.

The following example illustrates the granularity of mixing static and dynamic components that is possible. It consists of three `af:group` tags, the first and third of which contain static content and the second of which contains an `af:iterator` tag with a nested `af:dynamicComponent` tag.

```
<af:group title="static before dynamic" id="g1">
    <af:separator id="sp2"/>
    <af:inputText value="#{bindings.Empno.inputValue}" label="Static Empno"
                  required="#{bindings.Empno.hints.mandatory}"
                  columns="#{bindings.Empno.hints.displayWidth}"
                  maximumLength="#{bindings.Empno.hints.precision}"
                  shortDesc="#{bindings.Empno.hints.tooltip}" id="it1">
        <f:validator binding="#{bindings.Empno.validator}"/>
        <af:convertNumber groupingUsed="false"
                          pattern="#{bindings.Empno.format}"/>
    </af:inputText>
</af:group>
<af:group title="dynamic part" id="g2">
    <af:separator id="sp3"/>
    <af:panelFormLayout id="pfl1">
```

```
        <af:iterator id="i1"
value="#{bindings.EmpVO1.attributesModel.attributes}"
                     var="attr">
            <af:dynamicComponent
                     value="#{bindings[attr.containerName]
[attr.name].inputValue}"
                     id="dc1" attributeModel="#{attr}"/>
        </af:iterator>
    </af:panelFormLayout>
</af:group>
<af:group title="static after dynamic" id="g3">
    <af:separator id="sp4"/>
    <af:inputText value="#{bindings.Ename.inputValue}" label="Static Ename"
                  required="#{bindings.Ename.hints.mandatory}"
                  columns="#{bindings.Ename.hints.displayWidth}"
                  maximumLength="#{bindings.Ename.hints.precision}"
                  shortDesc="#{bindings.Ename.hints.tooltip}" id="it2">
        <f:validator binding="#{bindings.Ename.validator}"/>
    </af:inputText>
</af:group>
<af:button text="Back" id="cbb1" action="back"/>
```

How to Programmatically Set Dynamic Component Behavior

If you have any custom processing that you need to do to determine how the dynamic component is rendered, you can do so in a managed bean and then bind the dynamic component's `attributeModel` property to the appropriate bean method.

Before you begin:

It may be helpful to have an understanding of dynamic forms. For more information, see Creating a Form with Dynamic Components.

You may also find it helpful to understand other ADF functionality and features. For more information, see Additional Functionality for Databound Pages.

You will need to complete this task:

> Create a dynamic form as shown in How to Create a Dynamic Form or a dynamic table as shown in How to Create a Dynamic Table.

For information about creating managed beans see Using a Managed Bean in a Fusion Web Application

To programmatically set a dynamic component's behavior:

1. In the UI project that contains the dynamic component, create a Java class that extends the `oracle.adf.view.rich.model.BaseAttributeDescriptor` class.

   As the following example shows a sample class that adds a method to apply a custom style to the label for the `Sal` attribute.

   ```
   public class CustomizedDescriptor extends BaseAttributeDescriptor
   {
     public CustomizedDescriptor(BaseAttributeDescriptor base)
     {
       _base = base;
     }

     public Object getLabelStyle()
     {
   ```

```
    if (getName().equals("Sal"))
      return "color:red; font-weight:bold";
    else
      return null;
  }
```

2. Create a managed bean for the page.
3. In the managed bean, create a method that returns an instance of the custom descriptor. As the following example shows such a method.

   ```
   public CustomizedDescriptor getCustomizedAttributes(String attrName)
     {
     // create value expression for "#{attrName}" and evaluate it to get object
     // of "attr"
       BaseAttributeDescriptor attrMetadata = null;
       Class klass = null;

       try
       {
         klass =
   ClassLoaderUtils.loadClass("oracle.adf.view.rich.model.BaseAttributeDescripto
   r");
       }
       catch (ClassNotFoundException e)
       {
         throw new RuntimeException("Can not find class
   oracle.adf.view.rich.model.BaseAttributeDescriptor");
       }

       FacesContext facesContext = FacesContext.getCurrentInstance();
       ELContext elContext = facesContext.getELContext();
       ValueExpression valueExpression = createValueExpression(attrName, klass);

       if (valueExpression != null)
         attrMetadata =
   (BaseAttributeDescriptor)valueExpression.getValue(elContext);

       return new CustomizedDescriptor(attrMetadata);
     }
   ```

4. In the source editor for the page containing the dynamic component, bind the dynamic component's `attributeModel` property to the managed bean method that returns the customized attribute descriptor.

   As the following example shows an `af:dynamicComponent` tag where its `attributeModel` attribute has been set to an EL expression that references the `getCustomizedAttributes` method of the `attributeCustomBean` managed bean.

   ```
   <af:panelFormLayout id="pfl1">
     <af:iterator id="i1" value="#{bindings.EmpVO1.attributesModel.attributes}"
                  var="attr">
       <af:dynamicComponent value="#{bindings[attr.containerName]
   [attr.name].inputValue}"
   attributeModel="#{attributeCustomBean.getCustomizedAttributes('attr')}" /
   >
     </af:iterator>
   </af:panelFormLayout>
   ```

How to Access a Dynamic Component Programmatically

If you need to access a dynamic component instance programmatically to make changes based on a user action, you can do so by using a managed bean.

Before you begin:

It may be helpful to have an understanding of dynamic forms. For more information, see Creating a Form with Dynamic Components.

You may also find it helpful to understand other ADF functionality and features. For more information, see Additional Functionality for Databound Pages.

You will need to complete this task:

> Create a dynamic form as shown in How to Create a Dynamic Form or a dynamic table as shown in How to Create a Dynamic Table.

To programmatically access a dynamic component's binding instance:

1. Create a managed bean method that identifies dynamic component, casts the component to `RichDynamicComponent`, and then applies any changes.

   For more information, see Using a Managed Bean in a Fusion Web Application.

2. In the page containing the dynamic component, add a command component to the page and bind its `actionListener` property to the managed bean method.

The following example shows a managed bean method that changes the label style for a dynamic component.

```
public void changeLabelColor (ActionEvent event)
  {
    UIComponent component = event.getComponent().getParent();
    component = component.findComponent("pfl1");

    while(component != null)
    {
      if(component.getId().equals("it1"))
      {
        RichDynamicComponent rdc = (RichDynamicComponent)component;
        rdc.setLabelStyle("color:red");
        break;
      }
      else
        component = component.getChildren().get(0);
    }
  }
```

How to Access a Dynamic Component's Binding Instance Programmatically

If you need to access a runtime binding instance of a dynamic component programmatically, you can do so using a managed bean.

Before you begin:

It may be helpful to have an understanding of dynamic forms. For more information, see Creating a Form with Dynamic Components.

You may also find it helpful to understand other ADF functionality and features. For more information, see Additional Functionality for Databound Pages.

You will need to complete this task:

Create a dynamic form as shown in How to Create a Dynamic Form or a dynamic table as shown in How to Create a Dynamic Table.

To access a dynamic component's binding instance programmatically:

1. Create a managed bean method that accesses the binding container through the `FacesContext`, gets the `AttributesModel`, and then applies the desired business logic.

   For more information, see Using a Managed Bean in a Fusion Web Application.

2. In the page containing the dynamic component, add a command component to the page and bind its `actionListener` property to the managed bean method.

The following example illustrates a managed bean method that will set the `DeptNo` value of the current row to 9999.

```
public void setValueThroughBinding(ActionEvent event)
  {
    // get binding container
    FacesContext facesContext = FacesContext.getCurrentInstance();
    ELContext elContext = facesContext.getELContext();
    ValueExpression valueExpression = createValueExpression("bindings",
                                  DCBindingContainer.class);
    DCBindingContainer binding = (DCBindingContainer)
                             valueExpression.getValue(elContext);

    // get AttributesModel and attribute metadata list
    valueExpression = createValueExpression("bindings.EmpVO1.attributesModel",
                      AttributesModel.class);
    AttributesModel attributesModel = (AttributesModel)
                                  valueExpression.getValue(elContext);
    List<BaseAttributeDescriptor> attrList = attributesModel.getAttributes();

    // get AttributeMetadata for "Empno"
    BaseAttributeDescriptor deptNoAttributeBase = null;
    FacesAttributeDescriptor deptNoAttribute = null;
    for (BaseAttributeDescriptor attrDescriptor: attrList)
    {
      if ("Comm".equals(attrDescriptor.getName()))
      {
        deptNoAttributeBase = attrDescriptor;
        break;
      }
    }

    deptNoAttribute = (FacesAttributeDescriptor)deptNoAttributeBase;
    if (deptNoAttribute == null || deptNoAttribute.getContainerName() == null)
    {
      System.out.println("AttributesModel of Empno or its model name");
      System.out.println("can not be found.");
      return;
    }

    // get Deptno attribute value binding, set input value
    JUFormBinding fb = (JUFormBinding) binding.findExecutableBinding((String)
                                    deptNoAttribute.getContainerName());
    JUCtrlValueBinding valueBinding = (JUCtrlValueBinding)

fb.getControlBinding(deptNoAttribute.getName());
    valueBinding.setInputValue(Integer.valueOf(9999));

    System.out.println("Found JUCtrlValueBinding " + deptNoAttribute.getName()");
    System.out.println("set its value to 9999.");
  }
```

Modifying the UI Components and Bindings on a Form

The Data Controls panel is used to modify any type of form (except a dynamic form) and ADF bindings on a form. You can customize the display of the UI as per requirement.

Once you use the Data Controls panel to create any type of form (except a dynamic form), you can then delete attributes, change the order in which they are displayed, change the component used to display data, and change the attribute to which the components are bound.

> **Note:**
>
> You cannot change how a dynamic form displays using the following procedures. You must change display information on the view object or entity object instead.

How to Modify the UI Components and Bindings

You can modify certain aspects of the default components dropped from the Data Controls panel. You can use the Structure window to change the order in which components are displayed, to add new components or change existing components, or to delete components. You can use the Properties window to change or delete bindings, or to change the label displayed for a component.

Before you begin:

It may be helpful to have a general understanding of working with UI components after they have been generated. For more information, see Modifying the UI Components and Bindings on a Form.

You may also find it helpful to understand other ADF functionality and features. For more information, see Additional Functionality for Databound Pages.

You will need to complete these tasks:

- Create an application module that contains instances of the view objects that you want in your data model, as described in Creating and Modifying an Application Module. Alternatively, if you are using another type of business service, create a data control for that business service as described in Exposing Business Services with Data Controls in *Developing Applications with Oracle ADF Data Controls*.
- Create a JSF page as described in Creating a Web Page.

To modify default components and bindings:

1. Use the Structure window to do the following:
   - Change the order of the UI components by dragging them up or down the tree. A black line with an arrowhead denotes where the UI component will be placed.
   - Add a UI component. Right-click an existing UI component in the Structure window and choose to place the new component before, after, or inside the selected component. You then choose from a list of UI components.
   - Bind a UI component. Right-click an existing UI component in the Structure window and choose **Bind to ADF Control**. You can then select the object to which you want your component bound.
   - Rebind a UI component. Right-click an existing UI component in the Structure window and choose **Rebind to another ADF Control**. You can then select the new control object to which you want your component bound.
   - Delete a UI component. Right-click the component and choose **Delete**. If you wish to keep the component, but delete the binding, you need to use the Properties window. See the second bullet point in Step 2.
2. With the UI component selected in the Structure window, you can then do the following in the Properties window:
   - Add a non-ADF binding for the UI component. Enter an EL expression in the **Value** field, or use the dropdown menu and choose **Edit**.
   - Delete a binding for the UI component by deleting the EL expression.
   - Change the label for the UI component. By default, the label is bound to the binding's `label` property of its hint. This property allows your page to use the UI control hints for labels that you have defined for your entity object attributes or view object attributes. The UI hints allow you to change the value once and have it appear the same on all pages that display the label.

     You can change the label just for the current page. To do so, select the `label` attribute. You can enter text or an EL expression to bind the label value to something else, for example, a key in a properties or resource file.

     For example, the `inputText` component used to display the name of a product might have the following for its `Label` attribute:

     ```
     #{bindings.ProductName.hints.label}
     ```

     However, you could change the expression to instead bind to a key in a properties file, for example:

     ```
     #{properties['productName']}
     ```

     In this example, `properties` is a variable defined in the JSF page used to load a properties file.

What Happens When You Modify Attributes and Bindings

When you modify how an attribute is displayed by moving or changing the UI component, JDeveloper changes the corresponding code on the JSF page. When you use the binding editors to add or change a binding, JDeveloper adds the code to the JSF page, and also adds the appropriate elements to the page definition file.

32
Creating ADF Databound Tables

This chapter describes how to create tables with data modeled from ADF Business Components, using ADF data controls and ADF Faces components, including editable tables, input tables, and dynamic tables.
This chapter includes the following sections:

About Creating ADF Databound Tables

A table is used to display tabular data. You can bind the ADF Faces table component to the complete collection of data objects at a time from the collection. You can create and modify data in the table.

Unlike forms, tables allow you to display more than one data object from a collection at a time.

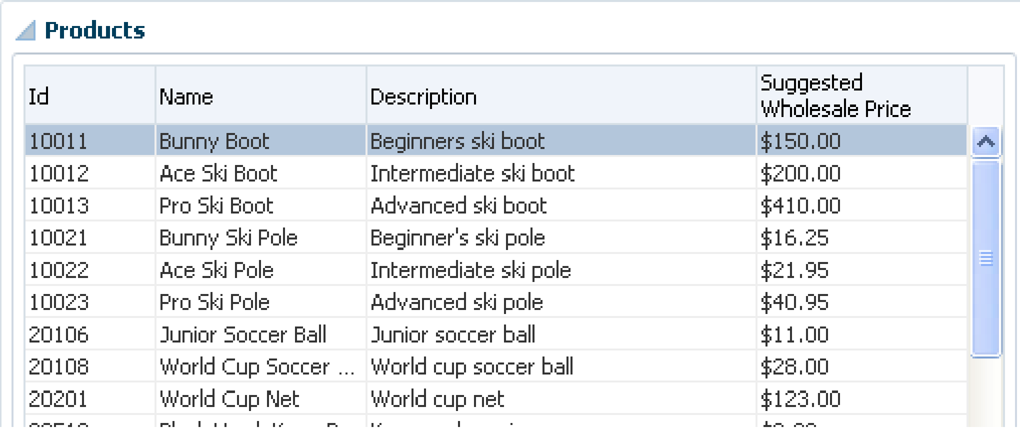
You can create tables that simply display data, or you can create tables that allow you to edit or create data. Once you drop a collection as a table, you can add command buttons bound to actions that execute some logic on a selected row. You can also modify the default components to suit your needs.

You can also display a collection of objects in a list that uses a grid to display each object's attributes, similar to a simple table.

ADF Databound Tables Use Cases and Examples

Figure 32-1 shows the Products table on the Inventory Control tab of the Summit sample application for **Oracle ADF**. This table is used to display all of the products in the inventory and enables the user to select a product to see its inventory status in an adjoining graph.

Figure 32-1 Read-only Table

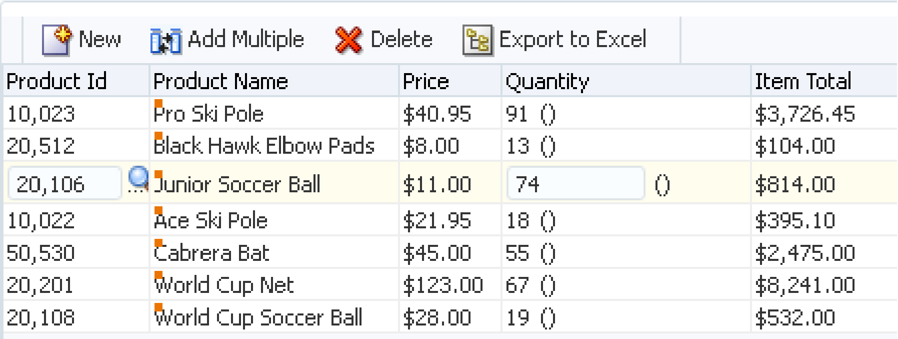


Products

Id	Name	Description	Suggested Wholesale Price
10011	Bunny Boot	Beginners ski boot	$150.00
10012	Ace Ski Boot	Intermediate ski boot	$200.00
10013	Pro Ski Boot	Advanced ski boot	$410.00
10021	Bunny Ski Pole	Beginner's ski pole	$16.25
10022	Ace Ski Pole	Intermediate ski pole	$21.95
10023	Pro Ski Pole	Advanced ski pole	$40.95
20106	Junior Soccer Ball	Junior soccer ball	$11.00
20108	World Cup Soccer ...	World cup soccer ball	$28.00
20201	World Cup Net	World cup net	$123.00

You can create a table that simply displays information, as shown in Figure 32-1, or you can allow the user to edit the information and create new records, as shown in Figure 32-2.

Figure 32-2 Edit Table



Product Id	Product Name	Price	Quantity	Item Total
10,023	Pro Ski Pole	$40.95	91 ()	$3,726.45
20,512	Black Hawk Elbow Pads	$8.00	13 ()	$104.00
20,106	Junior Soccer Ball	$11.00	74 ()	$814.00
10,022	Ace Ski Pole	$21.95	18 ()	$395.10
50,530	Cabrera Bat	$45.00	55 ()	$2,475.00
20,201	World Cup Net	$123.00	67 ()	$8,241.00
20,108	World Cup Soccer Ball	$28.00	19 ()	$532.00

Additional Functionality for Databound Tables

You may find it helpful to understand other ADF Faces features before you implement your table and tree components. Additionally, once you have added a tree or table component to your page, you may find that you need to add functionality such as validation and accessibility. Following are links to other functionality that table and tree components can use.

- **ADF view objects**: Much of how the components display and function in the table is controlled by the corresponding **view objects**. See Defining SQL Queries Using View Objects.
- **ADF application modules**: The Data Controls panel, from which you drag databound components to your pages, is populated with representations of view objects that you have added to **application modules**. See Implementing Business Services with Application Modules.
- **Adapter-based data controls**: If you are using other types of business services, such as EJB components or web services, you can create data control for those

business services as described in Creating and Configuring EJB Data Controls of the *Developing Applications with Oracle ADF Data Controls*.

- **ADF Model and data binding**: When you create databound tables in an ADF web application, you use ADF Model and data binding. See Using ADF Model in a Fusion Web Application.
- **ADF Faces**: For detailed information about developing with ADF Faces, see Introduction to ADF Faces in *Developing Web User Interfaces with Oracle ADF Faces*.
- **Command buttons**: You may want to add command button or links that invoke some functionality against the data set. For more advanced functionality than what is covered in this chapter, see Using Command Components to Invoke Functionality in the View Layer.
- **Filtered tables**: You can create tables that provide filtering, in effect executing a query-by-example search against the data in the table. See Creating Standalone Filtered Search Tables from Named View Criteria.
- **Task flows**: If your table takes part in a transaction, for example an input table, then you may need to use an ADF task flow to invoke certain operations before or after the table is rendered. See Creating ADF Task Flows .
- **Validation**: You may want certain fields to be validated before they are submitted to the data store. See Defining Validation and Business Rules Declaratively and Using Validation in the ADF Model Layer .
- **Active data**: If your application uses active data, then you can have the data in your tables and trees update automatically, whenever the data in the data source changes. See Using the Active Data Service .

> **Note:**
>
> If you wish to use active data, and your application uses ADF Business Components, then your tables must conform to the following:
>
> – The binding represents homogeneous data (that is, only one rule), although an accessor can still access a like accessor.
>
> – The binding rule contains a single attribute.
>
> – The table does not use filtering.
>
> – The tree component's `nodeStamp` facet contains a single `outputText` tag and contains no other tags.

Creating a Basic Table

In order to create a table using data control, you need to bind the ADF Faces table component to a collection of data objects. With the help of JDeveloper, you can perform this declaratively by dragging and dropping a collection from the Data Controls panel.

Unlike with forms where you bind the individual UI components that make up a form to the individual attributes on the collection, with a table you bind the ADF Faces `table` component to the complete collection or to a range of *n* data objects at a time from the collection. The individual components used to display the data in the columns are then bound to the attributes. The iterator binding handles displaying the correct data for each object, while the `table` component handles displaying each object in a row. JDeveloper allows you to do this declaratively, so that you don't need to write any code.

How to Create a Basic Table

To create a table using a data control, you bind the `table` component to a collection. JDeveloper allows you to do this declaratively by dragging and dropping a collection from the Data Controls panel.

Before you begin:

It may be helpful to have an understanding of ADF Faces databound tables. For more information, see Creating a Basic Table.

You may also find it helpful to understand other ADF functionality and features. For more information, see Additional Functionality for Databound Tables.

You will need to complete these tasks:

- Create an **application module** that contains instances of the view objects that you want in your data model, as described in Creating and Modifying an Application Module. Alternatively, if you are using another type of business service, create a data control for that business service as described in Exposing Business Services with Data Controls in *Developing Applications with Oracle ADF Data Controls*.
- Create a JSF page as described in Creating a Web Page.

To create a databound table:

1. From the Data Controls panel, select a collection.

   For example, to create a simple table in the Summit ADF sample application that displays all of the customers, you would select the `Customers` collection.

2. Drag the collection onto a JSF page, and from the context menu, choose **Table/ List View > ADF Table**.

3. The Create Table dialog shows each attribute in the collection, and allows you to determine how these attributes will behave and appear as columns in your table.

   Using this dialog, you can do the following:

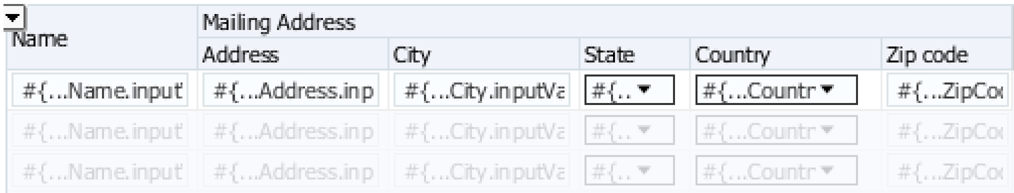
   - Make the table read-only.
   - Allow the ADF Model layer to handle selection by selecting the **Single Row** or **Multiple Rows** radio button. Selecting either of these options means that the iterator binding will access the iterator to determine the selected row or rows. Select one of these options unless you do not want the table to allow selection.
   - Allow the ADF Model layer to handle column sorting by selecting the **Enabling Sorting** checkbox. Selecting this option means that the iterator binding will access the iterator, which will perform an order-by query to determine the order. Select this option unless you do not want to allow column sorting.
   - Allow the columns in the table to be filtered using entered criteria by selecting the **Enabling Filtering** checkbox. Selecting this option allows the user to enter criteria in text fields above each column. That criteria is then used to build a Query-by-Example (QBE) search on the collection, so that the table

will display only the results returned by the query. For more information, see Creating Standalone Filtered Search Tables from Named View Criteria.

> **Note:**
>
> For JPA-based data controls, filtering does not work on tables that are created from nested collections in the data control. If you drop such a collection on a page, you should leave this checkbox unselected to avoid runtime errors. For more information on JPA-based data controls, see About EJB Data Controls in *Developing Applications with Oracle ADF Data Controls*.

- Group columns for selected attributes together under a parent column, by selecting the desired attributes (shown as rows in the dialog), and clicking the **Group** button. Figure 32-3 shows how five grouped columns appear in the visual editor after the table is created.

Figure 32-3 Grouped Columns in an ADF Faces Table



Name	Mailing Address				
	Address	City	State	Country	Zip code
#{...Name.input	#{...Address.inp	#{...City.inputVa	#{.. ▼	#{...Countr ▼	#{...ZipCo
#{...Name.input	#{...Address.inp	#{...City.inputVa	#{.. ▼	#{...Countr ▼	#{...ZipCo
#{...Name.input	#{...Address.inp	#{...City.inputVa	#{.. ▼	#{...Countr ▼	#{...ZipCo

- Change the display label for a column. By default, the label is bound to the `labels` property for any control hint defined for the attribute on the table binding. This binding allows you to change the value of a label text once on the view object, and have the change appear the same on all pages that display the label.

  Instead of using this default, you can enter text or an EL expression to bind the label value to something else, for example, a key in a resource file.

- Change the value binding for a column. You can change the column to be bound to a different attribute. If you simply want to rearrange the columns, you should use the order buttons. If you do change the attribute binding for a column, the label for the column also changes.
- Change the UI component used to display an attribute. The UI components are set based on the table you selected when you dropped the collection onto the page, on the type of the corresponding attribute (for example, `inputDate` components are used for attributes that are dates), and on whether or not default components were set as control hints on the corresponding view object. You can change to another component using the dropdown list.

> **Tip:**
>
> If one of the attributes for your table is also a primary key, you may want to choose a UI component that will not allow a user to change the value.

> **Tip:**
>
> If you want to use a component that is not listed in the dropdown menu, use this dialog to select the `outputText` component, and then manually add the other tag to the page.

- Change the order of the columns using the order buttons.
- Add a column using the **Add** icon. There's no limit to the number of columns you can add. When you first click the icon, JDeveloper adds a new column line at the bottom of the dialog and populates it with the values from the first attribute in the bound collection; subsequent new columns are populated with values from the next attribute in the sequence, and so on.
- Delete a column using the **Delete** icon.
- Make the table dynamic by selecting the **Generate columns dynamically at runtime** checkbox. For more information on dynamic tables, see Creating a Table with Dynamic Components.

4. Once the table is dropped on the page, you can use the Properties window to set other display properties of the table. For example, you may want to set the width of the table to a certain percentage or size. For more information about display properties, see the Using Tables, Trees, and Other Collection-Based Components chapter of *Developing Web User Interfaces with Oracle ADF Faces*.

   > **Tip:**
   >
   > When you set the table width to 100%, the table will not include borders, so the actual width of the table will be larger. To have the table set to 100% of the container width, expand the **Style** section of the Properties window, select the **Box** tab, and set the `Border Width` attribute to 0 pixels.

5. By default, the table will enforce delayed scrolling behavior which waits until the user stops scrolling and the position of the scroller is established to scroll the table rows. You can use the Property window that you display for the table iterator binding to configure smooth scrolling behavior. For more information, see What You May Need to Know About Table Scrolling Behavior and Row Count.
6. If you want the user to be able to edit information in the table and save any changes, you need to provide a way to submit and persist those changes. For more information, see Creating an Editable Table. For procedures on creating tables that allow users to input data, see Creating an Input Table.

What Happens When You Create a Table

Dropping a table from the Data Controls panel has the same effect as dropping a text field or form. Briefly, JDeveloper does the following:

- Creates the bindings for the table and adds the bindings to the page definition file
- Adds the necessary code for the UI components to the JSF page

For more information, see What Happens When You Create a Text Field .

Iterator and Value Bindings for Tables

When you drop a table from the Data Controls panel, a tree value binding is created. A tree consists of a hierarchy of nodes, where each subnode is a branch off a higher level node. In the case of a table, it is a flattened hierarchy, where each attribute (column) is a subnode off the table. Like an attribute binding used in forms, the tree value binding references the iterator binding, while the iterator binding references an iterator for the data collection, which facilitates iterating over the data objects in the collection. Instead of creating a separate binding for each attribute, only the tree binding to the table node is created. In the tree binding, the AttrNames element within the nodeDefinition element contains a child element for each attribute that you want to be available for display or reference in each row of the table.

The tree value binding is an instance of the FacesCtrlHierBinding class that extends the core JUCtrlHierBinding class to add two JSF specific properties:

- collectionModel: Returns the data wrapped by an object that extends the javax.faces.model.DataModel object that JSF and ADF Faces use for collection-valued components like tables.
- treeModel: Extends collectionModel to return data that is hierarchical in nature. For more information, see Displaying Master-Detail Data.

The following example shows the value binding for the table created when you drop the Summit sample application for ADF task flows's ProductVO1 collection.

```
<bindings>
  <tree IterBinding="ProductVO1Iterator" id="ProductVO1">
    <nodeDefinition
            DefName="oracle.summit.model.views.ProductVO"
            Name="ProductVO10">
      <AttrNames>
        <Item Value="Id"/>
        <Item Value="Name"/>
        <Item Value="ShortDesc"/>
        <Item Value="LongtextId"/>
        <Item Value="ImageId"/>
        <Item Value="SuggestedWhlslPrice"/>
        <Item Value="WhlslUnits"/>
      </AttrNames>
    </nodeDefinition>
  </tree>
</bindings>
```

Only the table component needs to be bound to the model (as opposed to the columns or the text components within the individual cells), because only the table needs access to the data. The tree binding for the table drills down to the individual structure attributes in the table, and the table columns can then derive their information from the table component.

Code on the JSF Page for an ADF Faces Table

When you use the Data Controls panel to drop a table onto a JSF page, JDeveloper inserts an ADF Faces table component, which contains an ADF Faces column component for each attribute named in the table binding. Each column then contains another component (such as an inputText or outputText component) bound to the attribute's value. Each column's heading is bound to the `labels` property for the control hint of the attribute.

> **Tip:**
>
> If an attribute is marked as hidden on the associated view or entity object, no corresponding UI is created for it.

The following example shows a code excerpt from a table created by dropping the `ProductsVO1` collection as a read- only table.

```
<af:table value="#{bindings.ProductVO1.collectionModel}" var="row"
          rows="#{bindings.ProductVO1.rangeSize}"
          emptyText="#{bindings.ProductVO1.viewable ? 'No data to display.' :
'Access Denied.'}"
          fetchSize="#{bindings.ProductVO1.rangeSize}" rowBandingInterval="0"
          selectedRowKeys="#{bindings.ProductVO1.collectionModel.selectedRow}"
          selectionListener="#{bindings.ProductVO1.collectionModel.makeCurrent}"
          rowSelection="single" id="t1">
    <af:column headerText="#{bindings.ProductVO1.hints.Id.label}" id="c1">
        <af:outputText value="#{row.Id}"
                  shortDesc="#{bindings.ProductVO1.hints.Id.tooltip}"
                  id="ot1">
            <af:convertNumber groupingUsed="false"
                         pattern="#{bindings.ProductVO1.hints.Id.format}"/>
        </af:outputText>
    </af:column>
    <af:column headerText="#{bindings.ProductVO1.hints.Name.label}" id="c2">
        <af:outputText value="#{row.Name}"
                       shortDesc="#{bindings.ProductVO1.hints.Name.tooltip}"
                       id="ot2"/>
    </af:column>
    <af:column headerText="#{bindings.ProductVO1.hints.ShortDesc.label}" id="c3">
        <af:outputText value="#{row.ShortDesc}"
                  shortDesc="#{bindings.ProductVO1.hints.ShortDesc.tooltip}"
                  id="ot3"/>
    </af:column>
.
.
.
</af:table>
```

The tree binding iterates over the data exposed by the iterator binding. Note that the table's value is bound to the `collectionModel` property, which accesses the `collectionModel` object. The table wraps the result set from the iterator binding in a `collectionModel` object. The `collectionModel` allows each item in the collection to be available within the table component using the `var` attribute.

In the example, the table iterates over the rows in the current range of the `ProductVO1` iterator binding. The iterator binding binds to a **row set iterator** that keeps track of the current row. When you set the `var` attribute on the table to `row`, each column then accesses the current data object for the current row presented to the table tag using the `row` variable, as shown for the value of the `af:outputText` tag:

```
<af:outputText value="#{row.Id}"/>
```

When you drop an ADF Table and do not specify it as a read-only table, input components are generated instead of output components, as shown in the following example. The value of the input component is implicitly bound to a specific row in the binding container through the bindings property, instead of being bound to the row variable. Additionally, JDeveloper adds validator components for each input component. By using the bindings property, any raised exception can be linked to the corresponding binding object or objects. The controller iterates through all exceptions in the binding container and retrieves the binding object to get the client ID when creating FacesMessage objects. This retrieval allows the table to display errors for specific cells. This strategy is used for all input components, including selection components such as lists.

```
<af:table value="#{bindings.ProductVO1.collectionModel}" var="row"
          rows="#{bindings.ProductVO1.rangeSize}"
          emptyText="#{bindings.ProductVO1.viewable ? 'No data to display.' :
          'Access Denied.'}"
          fetchSize="#{bindings.ProductVO1.rangeSize}" rowBandingInterval="0"
          id="t1">
    <af:column headerText="#{bindings.ProductVO1.hints.Id.label}" id="c1">
        <af:inputText value="#{row.bindings.Id.inputValue}"
                      label="#{bindings.ProductVO1.hints.Id.label}"
                      required="#{bindings.ProductVO1.hints.Id.mandatory}"
                      columns="#{bindings.ProductVO1.hints.Id.displayWidth}"
                      maximumLength="#{bindings.ProductVO1.hints.Id.precision}"
                      shortDesc="#{bindings.ProductVO1.hints.Id.tooltip}"
                      id="it1">
            <f:validator binding="#{row.bindings.Id.validator}"/>
            <af:convertNumber groupingUsed="false"
                              pattern="#{bindings.ProductVO1.hints.Id.format}"/>
        </af:inputText>
    </af:column>
```

For more information about using ADF Faces validators and converters, see the Validating and Converting Input chapter of *Developing Web User Interfaces with Oracle ADF Faces*.

Table 32-1 shows the other attributes defined by default for ADF Faces tables created using the Data Controls panel.

Table 32-1 ADF Faces Table Attributes and Populated Values

Attribute	Description	Default Value
rows	Determines how many rows to display at one time.	An EL expression that, by default, evaluates to the rangeSize property of the associated iterator binding, which determines how many rows of data are fetched from a data control at one time. Note that the value of the rows attribute must be equal to or less than the corresponding iterator's rangeSize value, as the table cannot display more rows than are returned.
first	Index of the first row in a range (based on 0).	An EL expression that evaluates to the rangeStart property of the associated iterator binding.

Table 32-1 (Cont.) ADF Faces Table Attributes and Populated Values

Attribute	Description	Default Value
`emptyText`	Text to display when there are no rows to return.	An EL expression that evaluates to the viewable property on the iterator. If the table is viewable, the attribute displays **No data to display** when no objects are returned. If the table is not viewable (for example, if there are authorization restrictions set against the table), it displays **Access Denied**.
`fetchSize`	Number of rows of data fetched from the data source.	An EL expression that, by default, evaluates to the `rangeSize` property of the associated iterator binding. For more information about the `rangeSize` property, see Iterator RangeSize Attribute Defined. This attribute can be set to a larger number than the `rows` attribute. Note that to improve scrolling behavior in a table, when the table's iterator binding is expected to manage a data set consisting of over 200 items, and the view object is configured to use range paging, the iterator actually returns a set of ranges instead of just one range. For more information about using range paging for view objects, see Using Range Paging to Efficiently Scroll Through Large Result Sets. For more information on range paging for JPA-based data controls, see Paginated Fetching of Data in EJB Data Controls in *Developing Applications with Oracle ADF Data Controls*.
`selectedRowKeys`	The selection state for the table.	An EL expression that by default, evaluates to the selected row on the collection model.
`selectionListener`	Reference to a method that listens for a selection event.	An EL expression that by default, evaluates to the `makeCurrent` method on the collection model.
`rowSelection`	Determines whether rows are selectable.	Set to `single` to allow one row to be selected at a time. Set to `multiple` to allow more than one row to be selected.
Column Attributes		
`sortProperty`	Determines the property on which to sort the column.	Set to the columns corresponding attribute binding value.
`sortable`	Determines whether a column can be sorted.	Set to `false`. When set to `true`, the iterator binding will access the iterator to determine the order.
`headerText`	Determines the text displayed at the top of the column.	An EL expression that, by default, evaluates to the label control hint set on the corresponding attribute.

What You May Need to Know About Setting the Current Row in a Table

When you use tables in an application and you allow the ADF Model layer to manage row selection, the current row is determined by the iterator. When a user selects a row in an ADF Faces table, the row in the table is shaded, and the component notifies the iterator of the selected row. To do this, the `selectedRowKeys` attribute of the table is bound to the collection model's selected row, as shown in the following example.

```
<af:table value="#{bindings.ProductVO1.collectionModel}" var="row"
.
.
.
          selectedRowKeys="#{bindings.ProductVO1.collectionModel.selectedRow}"
          selectionListener="#{bindings.ProductVO1.collectionModel.makeCurrent}"
          rowSelection="single">
```

This binding binds the selected keys in the table to the selected row of the collection model. The `selectionListener` attribute is then bound to the collection model's `makeCurrent` property. This binding makes the selected row of the collection the current row of the iterator.

> **Note:**
>
> If you create a custom selection listener, you must create a method binding to the `makeCurrent` property on the collection model (for example `#{binding.Products.collectionModel.makeCurrent}`) and invoke this method binding in the custom selection listener before any custom logic.

Although a table can handle selection automatically, there may be cases where you need to programmatically set the current row for an object on an iterator.

You can call the `getKey()` method on any view row to get a `Key` object that encapsulates the one or more key attributes that identify the row. You can also use a `Key` object to find a view row in a row set using the `findByKey()` method. At runtime, when either the `setCurrentRowWithKey` or the `setCurrentRowWithKeyValue` built-in operation is invoked by name by the data binding layer, the `findByKey()` method is used to find the row based on the value passed in as a parameter before the found row is set as the current row.

The `setCurrentRowWithKey` and `setCurrentRowWithKeyValue` operations both expect a parameter named `rowKey`, but they differ precisely by what each expects that `rowKey` parameter value to be at runtime:

setCurrentRowWithKey Operation
`setCurrentRowWithKey` expects the `rowKey` parameter value to be the **serialized string representation** of a view row key. This is a hexadecimal-encoded string that looks similar to this:

```
000200000002C20200000002C102000000010000010A5AB7DAD9
```

The serialized string representation of a key encodes all of the key attributes that might comprise a view row's key in a way that can be conveniently passed as a single value in a browser URL string or form parameter. At runtime, if you inadvertently pass a parameter value that is not a legal serialized string key, you may receive exceptions like `oracle.jbo.InvalidParamException` or `java.io.EOFException` as a result. In your web page, you can access the value of the serialized string key of a row by referencing the `rowKeyStr` property of an ADF control binding (for example. `#{bindings.SomeAttrName.rowKeyStr}`) or the row variable of an ADF Faces table (e.g. `#{row.rowKeyStr}`).

setCurrentRowWithKeyValue Operation
The `setCurrentRowWithKeyValue` operation expects the `rowKey` parameter value to be the literal value representing the key of the view row. For example, its value would be simply "`201`" to find product number `201`.

> **Note:**
>
> If you write custom code in an application module class and need to find a row based on a serialized string key passed from the client, you can use the `getRowFromKey()` method in the `JboUtil` class in the `oracle.jbo.client` package:
>
> ```
> static public Row getRowFromKey(RowSetIterator rsi, String sKey)
> ```
>
> The first parameter is the view object instance in which you'd like to find the row. The second parameter is the serialized string format of the key.

What You May Need to Know About Getting Row Data Programmatically

`CollectionModel` supports a version of the `getRowData` method that takes a row key parameter and thus allows you to get row data without changing currency.

For example, you can replace this code:

```
CollectionModelInstance.setRowKey(RowKeyInstance);
JUCtrlHierNodeBinding rowData = (JUCtrlHierNodeBinding)
   CollectionModelInstance.getRowData();
```

with this code:

```
JUCtrlHierNodeBinding rowData = (JUCtrlHierNodeBinding)
   CollectionModelInstance.getRowData(RowKeyInstance);
```

This is particularly useful when your view object uses range paging, since it enables you to avoid having to use the `setRowKey` method. (If you use `setRowKey`, you might also need to provide custom code to ensure that the row that you specify with that method is within the current range of rows.)

What You May Need to Know About Table Scrolling Behavior and Row Count

When you add the table component to a page, you size the table to display fewer rows than the fetch size set on the backing view object and therefore only a portion of the total row count. At runtime, when the user scrolls through the result set and the total number of rows is much larger than table rows, it is usually desirable to wait until the user stops scrolling and the final position of the scroller is established to render the corresponding rows. This default behavior is called delayed scrolling because the table rows don’t change until the position of the scroller is established. In contrast, when few fetches are needed to scroll through the result set (because the row count is a small multiple of the backing view object’s fetch size), smooth scrolling may be desired to allow the user to view table rows corresponding to the changing position of the scroller.

You can configure the desired table scrolling behavior using the **RowCountThreshold** attribute on the table’s iterator binding. The default attribute value (0) gets the estimated row count when the table is first rendered and supports delayed scrolling through a large result set. In this case, the size of the scroller will be determined from the start and will not change as the user scrolls through the result set.

To support smooth scrolling, set the **RowCountThreshold** attribute on the iterator binding of the table component to a value less than 0 (for example, -1). This will enforce smooth scrolling through a small result set and defer getting an estimated row count until the last set of rows is fetched. In this case, the scroller size and position will be determined by the number of rows fetched by the table, and the scroller will get smaller as the user scrolls through the result set.

Alternatively, you can set the **RowCountThreshold** attribute to a value greater than 0 when you want to configure a threshold for smooth scrolling to begin. When the table renders, it executes the estimated row count and if the row count is less than the value you specify, the framework will enforce the default behavior (delay row scrolling until the scroller position is established). If the estimated row count exceeds the threshold value, the table enforces the smooth scrolling behavior.

Creating an Editable Table

The process of creating an editable table is similar to that of a basic table apart from the addition of command buttons bound to operations. You can add an ADF Faces component if you need your table to contain a toolbar and place the command buttons on it.

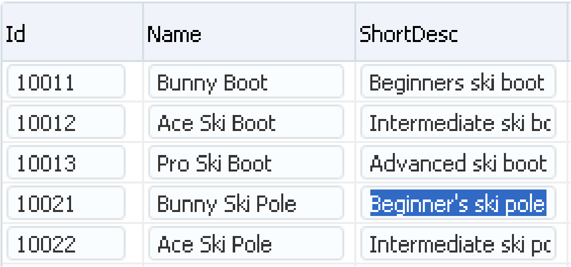
You can create a table that allows the user to edit information within the table, and then commit those changes to the data source. You can use built-in data control operations to create command buttons for changing the data, and place those buttons in a toolbar in the table. For example, you might use the `Delete` operation to create a button that allows a user to delete a record from the current range.

It is important to note that the collection-specific operations are executed only against objects in the ADF cache. You need to use the `Commit` operation on the root data control to actually commit any changes to the data source. You use the data control's `Rollback` operation to roll back any changes made to the cached object. If the page is part of a transaction within a bounded task flow, you would most likely use these operations to resolve the transaction in a task flow return activity. See Managing Transactions in Task Flows.

> **Note:**
>
> For data controls which do not have the `Commit` and `Rollback` operations available, such as data controls based on web services or stateless EJB session beans, you need to provide custom methods to save the data to the data source. For JPA-based data controls, you can use the `mergeEntity` or `persistEntity` methods to save changes, but these methods only work on a single row. For more information on working with JPA-based data controls, see About Commit Models for EJB Session Beans in *Developing Applications with Oracle ADF Data Controls*.

When you decide to use editable components to display your data, you have the option of the table displaying all rows as editable immediately, or displaying all rows as read-only until the user double-clicks within the row. Figure 32-4 shows a table whose rows all have editable fields. The page renders using the components that were added to the page (for example, `inputText` and `inputDate`, components).

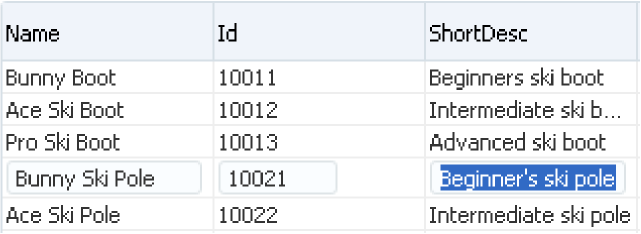
Figure 32-4 Table with Editable Fields



Id	Name	ShortDesc
10011	Bunny Boot	Beginners ski boot
10012	Ace Ski Boot	Intermediate ski bc
10013	Pro Ski Boot	Advanced ski boot
10021	Bunny Ski Pole	Beginner's ski pole
10022	Ace Ski Pole	Intermediate ski pc

Figure 32-5 shows the same table, but configured so that the user must double-click (or single-click if the row is already selected) a row in order to edit or enter data. Note that `outputText` components are used to display the data in the nonselected rows, even though the same input components as in Figure 32-4 were used to build the page. The only row that actually renders those components is the row selected for editing.

Figure 32-5 Click to Edit a Row



Name	Id	ShortDesc
Bunny Boot	10011	Beginners ski boot
Ace Ski Boot	10012	Intermediate ski b...
Pro Ski Boot	10013	Advanced ski boot
Bunny Ski Pole	10021	Beginner's ski pole
Ace Ski Pole	10022	Intermediate ski pole

For more information about how ADF Faces table components handle editing, see Editing Data in Tables, Trees, and Tree Tables in *Developing Web User Interfaces with Oracle ADF Faces*.

How to Create an Editable Table

To create an editable table, you follow similar procedures to creating a basic table, then you add command buttons bound to operations. However, in order for the table to contain a toolbar, you need to add an ADF Faces component that associates the toolbar with the items in the collection used to build the table.

Before you begin:

It may be helpful to have an understanding of editable databound tables. For more information, see Creating an Editable Table.

You may also find it helpful to understand other ADF functionality and features. For more information, see Additional Functionality for Databound Tables.

You will need to complete these tasks:

- Create an application module that contains instances of the view objects that you want in your data model, as described in Creating and Modifying an Application Module. Alternatively, if you are using another type of business service, create a data control for that business service as described in Exposing Business Services with Data Controls in *Developing Applications with Oracle ADF Data Controls*.
- Create a JSF page as described in Creating a Web Page.

To create an editable table:

1. From the Data Controls panel, select a collection.

   For example, to create a simple table in the Summit ADF task flow sample application that will allow you to edit products in the system, you would select the `ProductVO1` collection.

2. Drag the collection onto a JSF page, and from the context menu, choose **Table / List View > ADF Table**.

   This creates an editable table using input components.

3. Use the ensuing Create Table dialog to determine how the attributes should behave and appear as columns in your table. Be sure to select the **Single Row** or **Multiple Rows** radio button in order to allow the user to select a row to edit.

   For more information about using this dialog to configure the table, see How to Create a Basic Table.

4. With the table selected in the Structure window, expand the **Behavior** section of the Properties window and set the **EditingMode** attribute. If you want all the rows to be editable select **editAll**. If you want the user to click into a row to make it editable, select **clickToEdit**.

5. From the Structure window, right-click the table component and choose **Surround With**.

6. In the Surround With dialog, ensure that **ADF Faces** is selected in the dropdown list, select the **Panel Collection** component, and click **OK**.

   The `panelCollection` component's toolbar facet will hold the toolbar which, in turn, will hold the command components used to update the data.

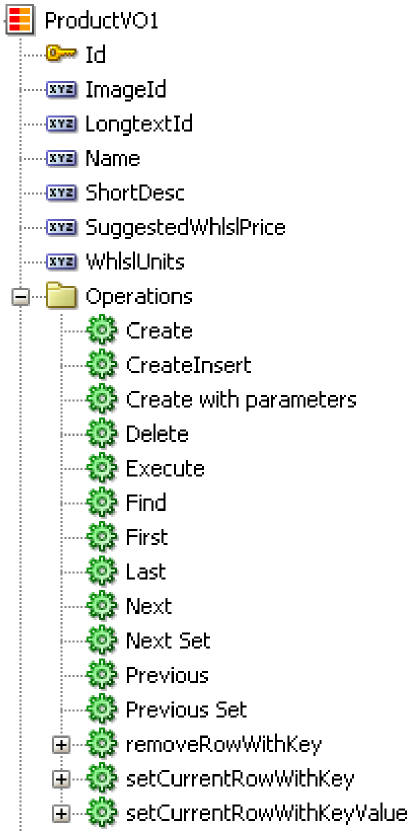
7. In the Structure window, right-click the `panelCollection`'s **f:facet - toolbar** facet node and choose **Insert inside f:facet - toolbar** > **Toolbar**.

This creates a toolbar that already contains a default menu that allows users to change how the table is displayed and a **Detach** link that detaches the entire table and displays it such that it occupies the majority of the space in the browser window. For more information about the `panelCollection` component, see the Displaying Table Menus, Toolbars, and Status Bars section of *Developing Web User Interfaces with Oracle ADF Faces*.

8. From the Data Controls panel, select the operation associated with the collection of objects on which you wish the operation to execute, drag it onto the `toolbar` component in the Structure window, and choose **Create > ADF Button** from the context menu. This will place the databound command component inside the toolbar.

   For example, if you want to be able to delete a product record, you would drag the `Delete` operation associated with the `ProductVO1` collection. Figure 32-6 shows the operations associated with he `ProductVO1` collection.

Figure 32-6 Operations Associated with a Collection



- ProductVO1
  - Id
  - ImageId
  - LongtextId
  - Name
  - ShortDesc
  - SuggestedWhlsPrice
  - WhlsUnits
  - Operations
    - Create
    - CreateInsert
    - Create with parameters
    - Delete
    - Execute
    - Find
    - First
    - Last
    - Next
    - Next Set
    - Previous
    - Previous Set
    - removeRowWithKey
    - setCurrentRowWithKey
    - setCurrentRowWithKeyValue

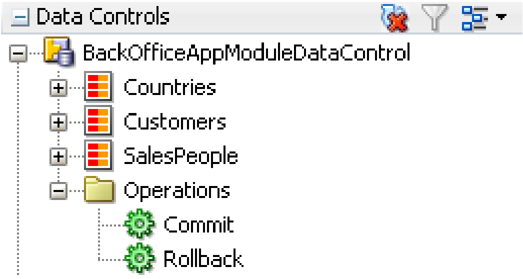
9. If you need a button for submitting changes to the cache, right-click the **af:toolbar** component in the Structure window and choose **Insert Inside Toolbar** > **Button**.

> **Note:**
>
> You can use other means to submit changes to the cache, such as setting the `autoSubmit` property to `true` for the input components in the table. For more information, see Using Partial Triggers in *Developing Web User Interfaces with Oracle ADF Faces*.

10. If the page is not part of a transaction within a bounded task flow, then you need to create buttons that allow the user to either commit or rollback the changes. From the Data Controls panel, drag the `Commit` and `Rollback` operations associated with the root-level data control, and drop them as either a button or link into the toolbar.

    Figure 32-7 shows the commit and roll back operations for the `BackOfficeAppModuleDataControl` data control.

    Figure 32-7 Commit and Rollback Operations for a Data Control



Data Controls
BackOfficeAppModuleDataControl
Countries
Customers
SalesPeople
Operations
Commit
Rollback

    If the page is part of a transaction within a bounded task flow, then you can simply enter `Commit` and `Rollback` as the values for the transaction resolution when creating the task flow return activity. For more information, see Managing Transactions in Task Flows.

> **Tip:**
>
> To create a table that allows you to insert a new record into the data store, see Creating an Input Table.

What Happens When You Create an Editable Table

Creating an editable table is similar to creating a form used to edit records. Action bindings are created for the operations dropped from the Data Controls panel. For details, see What Happens When You Create Command Components Using Operations.

Creating an Input Table

In order to create an input table, you need to first create an editable table and add a command component for creating new blank rows used to create new records.

You can create a table that allows users to insert a new blank row into a table and then add values for each column (any default values set on the corresponding entity or view object will be automatically populated).

How to Create an Input Table

When you create an input table, you want the user to see the new blank row in the context of the other rows within the current row set. To allow this insertion, you use the `CreateInsert` operation. As opposed to the `Create` operation, the `CreateInsert` operation actually creates the new row within the row set instead of only in the cache. In addition, you need to configure the table to respond to that user action. ADF Faces components can be set so that one component refreshes based on an interaction with another component, without the whole page needing to be refreshed. This is known as **partial page rendering**.

> **Note:**
>
> The `CreateInsert` operation is not available for adapter-based data controls, such as those based web services or Java classes. However, for JPA-based data controls, the `Create` operation adds the new row to the row set like `CreateInsert` does for data controls based on application modules. For more information, see Data Control Built-in Operations in *Developing Applications with Oracle ADF Data Controls*.

Before you begin:

It may be helpful to have an understanding of ADF Faces input tables. For more information, see Creating an Input Table.

You may also find it helpful to understand other ADF functionality and features. For more information, see Additional Functionality for Databound Tables.

You will need to complete these tasks:

- Create an editable table, as described in Creating an Editable Table.
- If your table is not part of a bounded task flow, include buttons bound to the `Commit` and `Rollback` operations.

To create an input table:

1. From the Data Controls panel, drag the **CreateInsert** operation associated with the dropped collection and drop it as a toolbar button into the toolbar. You may want to change the ID to something more recognizable, such as `CreateNewRow`. This will make it easier to identify when you need to select it as the partial trigger.

> **Note:**
>
> If the collection is a child to another collection (for example, in the Summit ADF sample application the `OrdersForCustomer` collection is a child to the `Customers` collection), then instead of using the `CreateInsert` operation, you need to use the `Create with Parameters` operation so that the record to create for the child collection is associated with the parent collection.

2. In the Structure window, select the table component. In the Properties window, expand the **Behavior** section.
3. In the Properties window, mouse over the **PartialTriggers** attribute, click the dropdown menu, and choose **Edit**.
4. In the Edit Property dialog, expand the toolbar facet for the `panelCollection` component and then expand the toolbar that contains the **CreateInsert** command component. Select that component and shuttle it to the **Selected** panel. Click **OK**. This sets that component to be the trigger that will cause the table to refresh.

What Happens When You Create an Input Table

When you use the `CreateInsert` operation to create an input table, JDeveloper:

- Creates an iterator binding for the collection, an action binding for the `CreateInsert` operation, and attribute bindings for the table. The `CreateInsert` operation is responsible for creating the new row in the row set. If you created command buttons or links using the `Commit` and `Rollback` operations, JDeveloper also creates an action bindings for those operations.
- Inserts code in the JSF page for the table using ADF Faces `table`, `column`, and `inputText` components, and in the case of the operations, `button` components.

The following example shows the page definition file for an input table created from the `ProductVO1` collection (some attributes were deleted in the Create Table dialog when the collection was dropped).

```
<executables>
  <iterator Binds="ProductVO1" RangeSize="25"
            DataControl="BackOfficeAppModuleDataControl"
id="ProductVO1Iterator"/>
</executables>
  <tree IterBinding="ProductVO1Iterator" id="ProductVO1">
    <nodeDefinition
            DefName="oracle.summit.model.views.ProductVO"
            Name="ProductVO10">
      <AttrNames>
        <Item Value="Id"/>
        <Item Value="Name"/>
        <Item Value="ShortDesc"/>
        <Item Value="LongtextId"/>
        <Item Value="ImageId"/>
        <Item Value="SuggestedWhlslPrice"/>
        <Item Value="WhlslUnits"/>
      </AttrNames>
    </nodeDefinition>
  </tree>
  <action IterBinding="ProductVOIterator" id="CreateInsert"
          RequiresUpdateModel="true" Action="createInsertRow"/>
  <action id="Commit" RequiresUpdateModel="true" Action="commitTransaction"
          DataControl="BackOfficeAppModuleDataControl"/>
  <action id="Rollback" RequiresUpdateModel="false"
          Action="rollbackTransaction"
          DataControl="BackOfficeAppModuleDataControl"/>
</bindings>
```

The following example shows the code added to the JSF page that provides partial page rendering, using the `CreateInsert` command toolbar button as the trigger to refresh the table.

```
<af:form>
  <af:panelCollection id="pc1">
    <f:facet name="menus"/>
    <f:facet name="toolbar">
      <af:toolbar id="tb1">
        <af:button actionListener="#{bindings.CreateInsert.execute}"
                                 text="CreateInsert"
                                 disabled="#{!bindings.CreateInsert.enabled}"
                                 id="CreateInsert"/>
        <af:button actionListener="#{bindings.Commit.execute}"
                                 text="Commit"
                                 disabled="false" id="b2"/>
        <af:button actionListener="#{bindings.Rollback.execute}"
                                 text="Rollback"
                                 disabled="#{!bindings.Rollback.enabled}"
                                 immediate="true" id="b3">
          <af:resetActionListener/>
        </af:button>
      </af:toolbar>
    </f:facet>
    <f:facet name="statusbar"/>
    <af:table value="#{bindings.ProductVO1.collectionModel}" var="row"
              rows="#{bindings.ProductVO1.rangeSize}"
              emptyText="#{bindings.ProductVO1.viewable ? \'No data to
display.\' :
                                                      \'Access Denied.\'}"
              fetchSize="#{bindings.ProductVO1.rangeSize}"
              rowSelection="single" partialTriggers="CreateInsert" id="t1">
      <af:column sortProperty="Id" sortable="false"
                 headerText="#{bindings.ProductVO1.hints.Id.label}"
                 id="c1">
        <af:inputText value="#{row.ProductId}" simple="true"
                      required="#{bindings.ProductVO1.hints.Id.mandatory}"
                      columns="#{bindings.ProductVO1.hints.Id.displayWidth}"
                      maximuumLength="#{bindings.ProductVO1.hints.Id.precision}"
                     id="it1"/>
      </af:column>
.
.
.
    </af:table>
  </af:panelCollection>
</af:form>
```

What Happens at Runtime: How CreateInsert and Partial Page Refresh Work

When the button bound to the `CreateInsert` operation is invoked, the action executes, and a new instance for the collection is created and inserted as the page is rerendered. Because the button was configured to be a trigger that causes the table to refresh, the table redraws with the new empty row shown at the top. When the user clicks the button bound to the `Commit` action, the newly created rows in the row set are inserted into the database. For more information about partial page refresh, see the Rerendering Partial Page Content chapter in *Developing Web User Interfaces with Oracle ADF Faces*.

What You May Need to Know About Creating a Row and Sorting Columns

If your table columns allow sorting, and the user has sorted on a column before inserting a new row, then that new row will not be sorted. To have the column sort with the new row, the user must first sort the column opposite to the desired sort, and then resort. This is because the table assumes the column is already sorted, so clicking on the desired sort order first will have no effect on the column.

For example, say a user had sorted a column in ascending order, and then added a new row. Initially, that row appears at the top. If the user first clicks to sort the column again in ascending order, the table will not resort, as it assumes the column is already in ascending order. The user must first sort on descending order and then ascending order.

If you want the data to automatically sort on a specific column in a specific order after inserting a row, then programmatically queue a `SortEvent` after the commit, and implement a handler to execute the sort.

If you want to allow sorting on a column for which the view object attribute is defined by a custom domain data type, you must implement the `compareTo()` method for the table's sort button click event:

```
public int compareTo(Object theObject)
```

The method will return `0` for values that are equal, less then `0` if the current object is less than `theObject`, and greater than `0` if the current object is greater. The object comparison for attributes behaves just like String comparisons.

What You May Need to Know About Create and CreateInsert

When you use the `Create` or `CreateInsert` operation to declaratively create a new row, it performs the following lines of code:

```
// create a new row for the view object
Row newRow = yourViewObject.createRow();
// mark the row as being "initialized", but not yet new
newRow.setNewRowState(Row.STATUS_INITIALIZED);
```

However, if you are using the `CreateInsert` operation, it performs the additional line of code to insert the row into the row set:

```
// insert the new row into the view object's default rowset
yourViewObject.insertRow(newRow);
```

When you create a row in an entity-based view object, the `Transaction` object associated with the current application module immediately takes note of the fact. The new entity row that gets created behind the view row is already part of the `Transaction`'s list of pending changes. When a newly created row is marked as having the *initialized* state, it is removed from the `Transaction`'s pending changes list and is considered a blank row in which the end user has not yet entered any data values. The term *initialized* is appropriate since the end user will see the new row initialized with any default values that the underlying entity object has defined. If the user never enters any data into any attribute of that initialized row, then it is as if the row never existed. At transaction commit time, since that row is not part of the `Transaction`'s pending changes list, no `INSERT` statement will be attempted for it.

As soon as at least one attribute in an initialized row is set, it automatically transitions from the initialized status to the new status (`Row.STATUS_NEW`). At that time, the underlying entity row is enrolled in the `Transaction`'s list of pending changes, and the new row will be permanently saved the next time you commit the transaction.

Note:

If the end user performs steps that result in creating many initialized rows but never populating them, it might seem likely to cause a slow memory leak. However, the memory used by an initialized row that never transitions to the new state will eventually be reclaimed by the Java virtual machine's garbage collector.

What You May Need to Know About Saving Multiple Rows in a JPA-Based Data Control

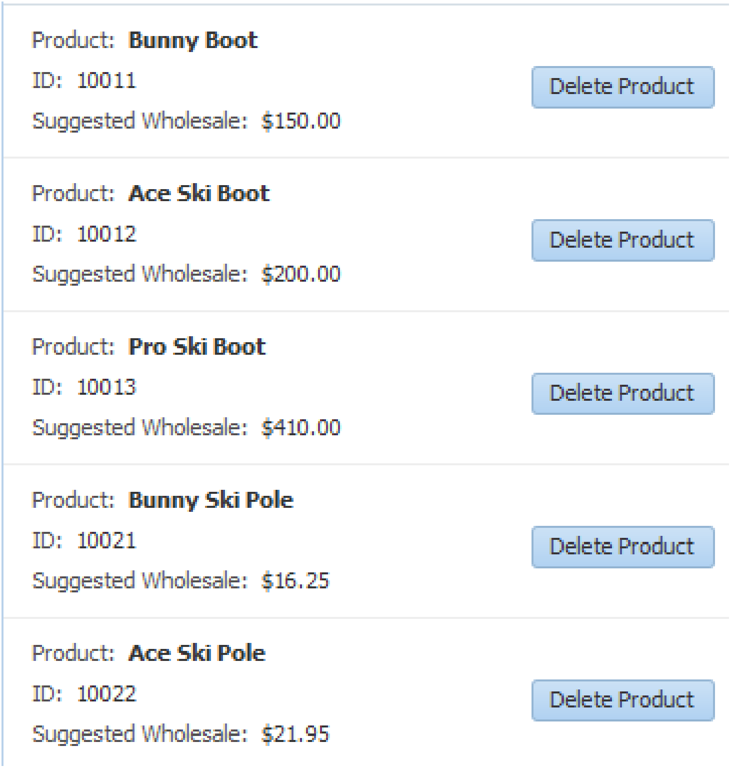
EJB data controls and JPA-based bean data controls do not have the built-in `CreateInsert` operation available. However, it is possible to replicate the behavior of `CreateInsert` with the `Create` operation in these data controls by setting the data control's `EagerPersist` property to `true`. This property is set to `true` by default for data controls based on stateful session beans that use container-managed transactions and an explicit commit model. For more information, see About Commit Models for EJB Session Beans in *Developing Applications with Oracle ADF Data Controls*

Creating a List View of a Collection

The listview component uses a model to display data in a list. You can choose between a panelGroupLayout or a panelGridLayout component to arrange the components for your list item.

You can use the `listView` component when you need to display a collection of objects in a list. The attributes of each object are displayed in a group or grid within a single row and single column. Figure 32-8 shows a collection of products displayed using a `listView` component.

Figure 32-8 A Collection Displayed in a List in a Single Column



Product: **Bunny Boot**
ID: 10011
Suggested Wholesale: $150.00

Product: **Ace Ski Boot**
ID: 10012
Suggested Wholesale: $200.00

Product: **Pro Ski Boot**
ID: 10013
Suggested Wholesale: $410.00

Product: **Bunny Ski Pole**
ID: 10021
Suggested Wholesale: $16.25

Product: **Ace Ski Pole**
ID: 10022
Suggested Wholesale: $21.95

In this `listView`, a child `listItem` component contains a `panelGridLayout` component that, in turn, contains `outputFormatted` components that display the data and a button component that allows users to delete the item. The `listView` component is bound to the collection, in the same way that a table is bound to a collection, using the `value` attribute. The `listItem` component is a direct child to the `listView` component, and is the container for the components that display the data, in the same way that the column component holds the data components for a table.

If your collection has a **master-detail relationship** with another collection, you can group the child collection under the parent collection using grouping functionality in the `listView` component. For more information, see Using List Views to Display Master-Detail Objects.

For complete information about the `listView` and `listItem` components, see the Displaying a Collection in a List section of the *Developing Web User Interfaces with Oracle ADF Faces*. For complete information about using the `panelGridLayout` component to group items in grid within the list (as shown in Figure 32-8), see the Arranging Content in a Grid section of the *Developing Web User Interfaces with Oracle ADF Faces*. You can also use a `panelGroupLayout` component to group components when you don't need a grid layout. For more information, see the Grouping Related Items section of the *Developing Web User Interfaces with Oracle ADF Faces*.

How to Create a Databound List View

You use the Create List View wizard to create your list. This wizard allows you to choose between a `panelGroupLayout` or a `panelGridLayout` component to arrange the components for your list item. If you are creating the list for a child of a master-detail relationship, you can also use the wizard to configure the headers for the list.

Before you begin:

It may be helpful to have an understanding of ADF Faces list views. For more information, see Creating a List View of a Collection.

You may also find it helpful to understand other ADF functionality and features. For more information, see Additional Functionality for Databound Tables.

You will need to complete these tasks:

- Create an application module that contains instances of the view objects that you want in your data model, as described in Creating and Modifying an Application Module. Alternatively, if you are using another type of business service, create a data control for that business service as described in Exposing Business Services with Data Controls in *Developing Applications with Oracle ADF Data Controls*.
- Create a JSF page as described in Creating a Web Page.

To create a list view:

1. From the Data Controls panel, drag a collection onto the page and choose **Table/List View > ADF List View**. If you want to create a list with headers from a parent collection, drag a collection that is a child to another collection.
2. Use the Create List View wizard to select your layout component to organize your display of the data, and to select the attributes and components to display. If you are creating a group, on the first page of the wizard, select **Include header using parent collection**.

   For more information about using the wizard, press the F1 key or click **Help**.
3. Use the Properties window to further configure the layout components.

   For complete information about using the `panelGridLayout` component, see the Arranging Content in a Grid section of the *Developing Web User Interfaces with Oracle ADF Faces*. For more information about the `panelGroupLayout` component, see the Grouping Related Items section of the *Developing Web User Interfaces with Oracle ADF Faces*.

What Happens When You Create a Databound List View

When you use the wizard to create a `listView` component, JDeveloper creates and configures the `listView`, `listItem`, and layout components. The bindings are the same as for a table—the `listView` component is bound to the collection dropped from the Data Controls panel. The value of the `var` attribute on this component is used by the `outputFormatted` components to access the data, in the same way as a table. For more information about the page definition code and the JSP code for collections, see What Happens When You Create a Table.

The following example shows the page code for the simple table shown in Figure 32-8.

```
<af:listView value="#{bindings.SProductView1.collectionModel}" var="item"
             emptyText="#{bindings.SProductView1.viewable ? 'No data to display.'
                                                           : 'Access Denied.'}"
             fetchSize="#{bindings.SProductView1.rangeSize}" id="lv1">
    <af:listItem id="li1">
        <af:panelGridLayout id="pgl2" dimensionsFrom="parent">
            <af:gridRow marginTop="5px" height="15px" id="gr2">
```

```
                <af:gridCell marginStart="5px" width="50%" id="gc2">
                    <af:panelLabelAndMessage label="Product:" id="plam1"
                                                  inlineStyle="font-
weight:bold;">
                        <af:outputFormatted
                               value="#{item.bindings.Name.inputValue}"
id="of1"/>
                    </af:panelLabelAndMessage>
                </af:gridCell>
                <af:gridCell marginStart="5px" width="50%" marginEnd="5px"
                             id="gc3"/>
            </af:gridRow>
            <af:gridRow marginTop="5px" height="15px" id="gr3">
                <af:gridCell marginStart="5px" width="50%" id="gc4">
                    <af:panelLabelAndMessage label="ID:" id="plam2">
                        <af:outputFormatted
value="#{item.bindings.Id.inputValue}"
                                            id="of2">
                            <af:convertNumber groupingUsed="false"

pattern="#{bindings.SProductView1.hints.Id.format}"/>
                        </af:outputFormatted>
                    </af:panelLabelAndMessage>
                </af:gridCell>
                <af:gridCell marginStart="5px" width="50%" marginEnd="5px"
                             id="gc5">
                    <af:button actionListener="#{bindings.Delete.execute}"
                               text="Delete Product"
                               disabled="#{!bindings.Delete.enabled}" id="b1"/>
                </af:gridCell>
            </af:gridRow>
            <af:gridRow marginTop="5px" height="15px" marginBottom="5px"
id="gr4">
                <af:gridCell marginStart="5px" width="50%" id="gc6">
                    <af:panelLabelAndMessage label="Suggested Wholesale:"
                                             id="plam3">
                        <af:outputFormatted

value="#{item.bindings.SuggestedWhlslPrice.inputValue}"
                           id="of3">
                            <af:convertNumber groupingUsed="false"
           pattern="#{bindings.SProductView1.hints.SuggestedWhlslPrice.format}"/>
                        </af:outputFormatted>
                    </af:panelLabelAndMessage>
                </af:gridCell>
                <af:gridCell marginStart="5px" width="50%" marginEnd="5px"
                             id="gc7"/>
            </af:gridRow>
        </af:panelGridLayout>
    </af:listItem>
</af:listView>
```

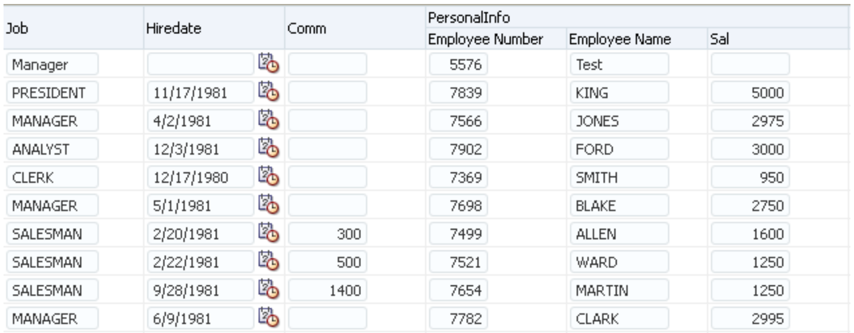
Creating a Table with Dynamic Components

A table can be created using ADF Faces dynamic components. The dynamic component in the table helps you to determine the binding metadata and the tags used to display the bound content each time you run it.

Instead of creating static databound tables where you provide tags for each component directly in the page, you can use a dynamic component to create tables where the binding metadata, columns, and the components used to display the bound content are determined at runtime.

Figure 32-9 shows a dynamic table at runtime that was created by setting UI hints for attributes on the `CustomerVO` view object and then dropping the `Customers` data control collection as a dynamic table. Among the UI hints set were `LABEL` and `DISPLAYHINT` (the latter of which can be set to `Hide` in order to not include the given attribute in the dynamic form).

Figure 32-9 Dynamic Table Including a Column Group Based on Category Hint



Job	Hiredate	Comm	PersonalInfo		
			Employee Number	Employee Name	Sal
Manager			5576	Test	
PRESIDENT	11/17/1981		7839	KING	5000
MANAGER	4/2/1981		7566	JONES	2975
ANALYST	12/3/1981		7902	FORD	3000
CLERK	12/17/1980		7369	SMITH	950
MANAGER	5/1/1981		7698	BLAKE	2750
SALESMAN	2/20/1981	300	7499	ALLEN	1600
SALESMAN	2/22/1981	500	7521	WARD	1250
SALESMAN	9/28/1981	1400	7654	MARTIN	1250
MANAGER	6/9/1981		7782	CLARK	2995

For more general information on dynamic components, see About Dynamic Components.

How to Create a Dynamic Table

To create a dynamic table, you drop a collection from the data controls panel as an ADF Table and specify that the fields be generated dynamically.

Before you begin:

It may be helpful to have an understanding of the dynamic component feature and dynamic databound tables. For more information, see About Dynamic Components and Creating a Table with Dynamic Components.

You may also find it helpful to understand other ADF functionality and features. For more information, see Additional Functionality for Databound Tables.

You will need to complete these tasks:

- Create an application module that contains instances of the view objects that you want in your data model, as described in Creating and Modifying an Application Module. Alternatively, if you are using another type of business service, create a data control for that business service as described in Exposing Business Services with Data Controls in *Developing Applications with Oracle ADF Data Controls*.
- Create a JSF page as described in Creating a Web Page.

UI hints determine things such as the type of UI component to use to display the attribute, the label, the tooltip, whether the field should be automatically submitted, and so forth. You can also determine whether a given attribute is displayed or hidden. For the procedure to create UI hints, see Defining UI Hints for View Objects.

To create a dynamic table:

1. Set UI Hints either on the view object or on the data control structure definition file that corresponds to the collection for which you are creating the dynamic form.
2. Optionally, for the `Category` hint on each view object attribute, specify a category or click the **New Category** icon to create a new category.

   Any attributes that have the same category can be grouped together in dynamic tables that you create based on that view object.
3. Optionally, for any categories that you have created set the UI hints for the category's label and tooltip. These can be set in the **UI Categories** tab of the view object's overview editor.
4. From the Data Controls panel, select the collection that represents the view object.
5. Drag the collection onto the page, and from the context menu, choose **Table/List View > ADF Table**.
6. In the Create Table dialog, select the **Generate columns dynamically at runtime** checkbox.
7. If you have specified any categories for attributes select the **Include Column Groups** checkbox in order for the form to organize the display of columns into any checkbox in order for the table to group the columns according to those categories.

What Happens When You Use Dynamic Components

When you drop a collection as a dynamic table on a page, the following things happen:

- The page definition is populated with a `variableIterator` binding, an `iterator` binding to the iterator, and a `tree` value binding. See Bindings Created for the Dynamic Form.
- The JSF page is populated with an `af:table` tag and one or more `af:iterator`, `af:column` and `af:dynamicComponent` tags. In addition, if the **Include Column Groups** option is selected, `af:switcher` and `af:group` tags are added.

Tags Created for a Dynamic Table without Grouping

In the JSF page, JDeveloper inserts an `af:table` tag, within which it nests an `af:iterator` tag that iterates over the attributes of the collection. Within that `af:iterator` tag is nested an `af:column` tag, which contains an `af:dynamicComponent` tag. The `af:dynamicComponent` in turn displays the appropriate component for each column as determined at runtime.

The following example shows the code that is generated when you drop the `Countries` data control object as a dynamic table (but do not select the **Include Column Groups** option).

```
<af:table value="#{bindings.Countries.collectionModel}" var="row"
          rows="#{bindings.Countries.rangeSize}"
          emptyText="#{bindings.Countries.viewable ? 'No data to display.' :
                                                    'Access Denied.'}"
          rowBandingInterval="0" fetchSize="#{bindings.Countries.rangeSize}"
          id="t1">
    <af:iterator id="i1"
                 value="#{bindings.Countries.attributesModel.attributes}"
                 var="column">
        <af:column headerText="#{column.label}" id="c1">
```

```
            <af:dynamicComponent id="d2" attributeModel="#{column}"

value="#{row.bindings[column.name].inputValue}"/>
        </af:column>
    </af:iterator>
</af:table>
```

The value attribute of the af:iterator tag uses an EL expression that evaluates to the attributesModel.attributes property of the collection's tree binding. The attributesModel property is used to retrieve the data object's attributes and their metadata, such as component type, label, tooltip, and other properties of the real component to be rendered. The attributes property of attributesModel signifies that a flat (unhierarchical) list of displayable attributes and their metadata is provided.

The attributeModel attribute of the af:dynamicComponent tag is bound to the EL expression #{column}, which references the variable that is defined in the iterator's var attribute and that serves as a pointer to the current attribute of the data control collection and its corresponding metadata. The EL expression for the dynamicComponent's value attribute also references the variable column.

Tags Created for a Dynamic Table with Grouping

If you have selected the **Include Column Groups** checkbox in the Create Table dialog, the generated JSF page includes the af:switcher and af:group tags, in addition to the tags described in Tags Created for a Dynamic Table without Grouping.

The af:switcher tag is nested directly within an af:column tag. Within the af:switcher tag are nested facet tags named GROUP and ATTRIBUTE. The GROUP facet contains an af:group tag, within which is an af:outputText tag to display the group name and an af:iterator tag, which contains an af:column tag that in turn contains an af:dynamicComponent tag. The ATTRIBUTE facet only contains an af:dynamicComponent tag.

For each attribute that the high-level iterator iterates over, the switcher dynamically determines whether to render a group or a column. If it renders a group, the iterator within the group then is used to render the columns within that group.

The following example shows the code that is generated if you create a dynamic form based on the Countries collection and choose to include column groups.

```
<af:table value="#{bindings.Countries.collectionModel}" var="row"
          rows="#{bindings.Countries.rangeSize}"
          emptyText="#{bindings.Countries.viewable ? 'No data to display.' :
'Access Denied.'}"
          rowBandingInterval="0" fetchSize="#{bindings.Countries.rangeSize}"
          id="t1">
    <af:iterator id="i1"

value="#{bindings.Countries.attributesModel.hierarchicalAttributes}"
                 var="column">
        <af:column headerText="#{column.label}" id="c1">
            <af:switcher id="sw1" facetName="#{column.descriptorType}"
                         defaultFacet="ATTRIBUTE">
                <f:facet name="GROUP">
                    <af:iterator id="gi1" value="#{column.descriptors}"
                                 var="nestedCol">
                        <af:column headerText="#{nestedCol.label}" id="c2">
                            <af:dynamicComponent id="gd1"
```

```
                                                   attributeModel="#{nestedCol}"

value="#{row.bindings[nestedCol.name].inputValue}"/>
                        </af:column>
                    </af:iterator>
                </f:facet>
                <f:facet name="ATTRIBUTE">
                    <af:dynamicComponent id="ad1" attributeModel="#{column}"
                                value="#{row.bindings[column.name].inputValue}"/>
                </f:facet>
            </af:switcher>
        </af:column>
    </af:iterator>
</af:table>
```

The `value` attribute of the `af:iterator` tag uses an EL expression that evaluates to the `attributesModel.hierarchicalAttributes` property of the collection's tree binding. The `attributesModel` property is used to retrieve the data object's attributes and their metadata, such as component type, label, tooltip, and other properties of the real component to be rendered. The `hierarchicalAttributes` property signifies that a hierarchical list of displayable attributes and their metadata is provided, including any categories that have been set for any attributes in the UI hints.

The `attributeModel` attribute of the `af:dynamicComponent` tag is set to the EL expression `#{column}`, which references the variable that is defined in the iterator's `var` attribute and that serves as a pointer to the current column (or category) of the data control collection and its corresponding metadata.

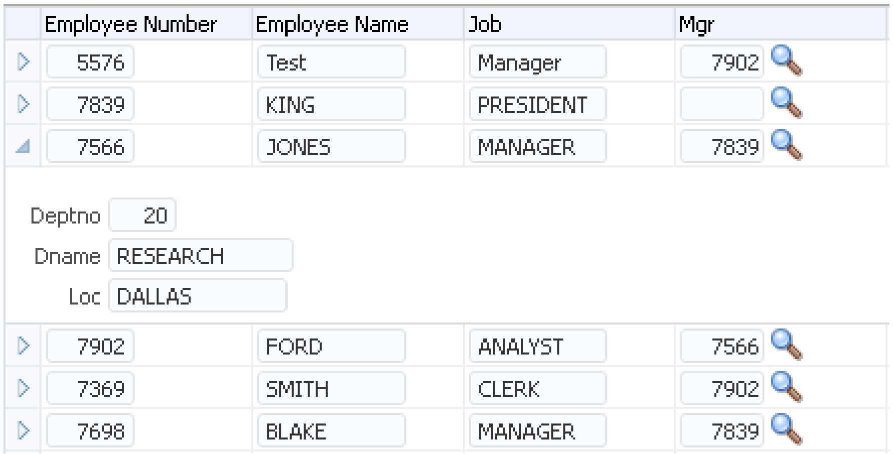
What Happens at Runtime: How Attribute Values Are Dynamically Determined

When a page with dynamic components is rendered, the bindings are created just as they are when items are dropped from the Data Controls panel at design time, except that they are created at runtime. For more information, see What Happens at Runtime: How Attribute Values Are Dynamically Determined.

How to Create a Dynamic Table with a detailStamp Facet

You can create a dynamic table that displays master content and includes `detailStamp` facet that enables a user to display detail for a given record. The detail content is obtained through the dynamic component's bindings to a **view link accessor** on the master view object. Figure 32-10 shows a part of a dynamic table that contains such a detail facet. For more information on using `detailStamp` facets in tables, see Adding Hidden Capabilities to a Table in *Developing Web User Interfaces with Oracle ADF Faces*.

Figure 32-10 Dynamic Table with detailStamp Facet



	Employee Number	Employee Name	Job	Mgr
▷	5576	Test	Manager	7902 🔍
▷	7839	KING	PRESIDENT	🔍
◢	7566	JONES	MANAGER	7839 🔍
	Deptno 20 Dname RESEARCH Loc DALLAS			
▷	7902	FORD	ANALYST	7566 🔍
▷	7369	SMITH	CLERK	7902 🔍
▷	7698	BLAKE	MANAGER	7839 🔍

Before you begin:

It may be helpful to have an understanding of dynamic tables. See Creating a Table with Dynamic Components.

You may also find it helpful to understand other ADF functionality and features. See Additional Functionality for Databound Tables.

You will need to complete these tasks:

- Create an application module that contains instances of the view objects and view links that establish the desired master-detail hierarchy, as described in How to Create a Master-Detail Hierarchy Based on View Objects Alone.
- Create a dynamic table as shown in How to Create a Dynamic Table.

To create a dynamic table with a detailStamp facet:

1. In the source editor for the JSF page, insert a `f:facet` tag within the dynamic table's `af:table` tag.
2. Within the `f:facet` tag, insert an `af:panelFormLayout` tag.
3. Within the `af:panelFormLayout` tag, insert an `af:iterator` tag.
4. Bind the iterator's `value` attribute to the `attributesModel.getLinkedViewAttributes(`*`LinkedViewName`*`)` property of your data object, where *`LinkedViewName`* is the name of a view link accessor that accesses the detail content.
5. Within the `af:iterator` tag, insert an `af:dynamicComponent` tag.
6. Bind the `af:dynamicComponent` tag's `attributeModel` attribute to the value of the `af:iterator` tag's `var` attribute.
7. Bind the `af:dynamicComponent` tag's `value` attribute to an expression that returns the detail data for the selected row.

The following example shows the JSF page code for a dynamic table that uses a `detailStamp` facet to display data retrieved through a view link accessor.

```
<af:table value="#{bindings.EmpVO3.collectionModel}" var="row" id="t1" ...>
  <f:facet name="detailStamp">
```

```
    <af:panelForm id="pgl1" layout="vertical">
      <af:iterator id="iter3" var="detail"

value="#{bindings.EmpVO3.attributesModel.getLinkedViewAttributes('DeptView')}">
        <af:dynamicComponent
value="#{row['DeptView'].bindings[detail.name].inputValue}"/>
attributeModel="#{detail}" id="dc1"/>
      </af:iterator>
    </af:panelForm>
  </f:facet>
  <af:iterator id="itr1" var="column"
      value="#{bindings.EmpVO3.attributesModel.attributes}">
    <af:column headerText="#{column.label}" id="dcc1">
      <af:dynamicComponent value="#{row.bindings[column.name].inputValue}"
attributeModel="#{column}" id="dc1"/>
    </af:column>
  </af:iterator>
</af:table>
```

Modifying the Attributes Displayed in the Table

You can modify and delete attributes displayed in a table that you don’t want to see in the UI.

Once you use the Data Controls panel to create a table, you can then delete attributes, change the order in which they are displayed, change the component used to display them, and change the attribute binding for the component. You can also add new attributes, or rebind the table to a new data control.

How to Modify the Displayed Attributes

You can modify the following aspects of a table that was created using the Data Controls panel:

- Change the binding for the label of a column
- Change the UI component bound to an attribute
- Add or delete columns that represent the attributes
- Reorder the columns in the table
- Enable selection and sorting

Before you begin:

It may be helpful to have an understanding of modifying databound tables. For more information, see Modifying the Attributes Displayed in the Table.

You may also find it helpful to understand other ADF functionality and features. For more information, see Additional Functionality for Databound Tables.

You will need to complete this task:

Create a databound table as described in Creating a Basic Table.

To change the attributes for a table:

1. In the Structure window, select the table component.

2. In the Properties window, expand the Columns section to change, add, or delete the attributes that are displayed as columns. You can also change the UI component used within the column to display the data.
3. Expand other sections of the Properties window to change the display and behavior of the table, such as filtering and sorting.

How to Change the Binding for a Table

Instead of modifying a binding, you can completely change the object to which the table is bound.

Before you begin:

It may be helpful to have an understanding of modifying databound tables. For more information, see Modifying the Attributes Displayed in the Table.

You may also find it helpful to understand other ADF functionality and features. For more information, see Additional Functionality for Databound Tables.

You will need to complete this task:

- Create a databound table as described in Creating a Basic Table.

To rebind a table:

1. Right-click the table in the Structure window and choose **Rebind to Another ADF Control**.
2. In the Bind to ADF Control dialog, select the new collection to which you want to bind the table.

   Note that changing the binding for the table will also change the binding for all the columns. You can then use the procedures in How to Modify the Displayed Attributes to modify those bindings.

> **Tip:**
>
> You can also rebind a table by dragging a different view object on top of the existing table.

What Happens When You Modify Bindings or Displayed Attributes

When you simply modify how an attribute is displayed by moving the UI component or changing the UI component, JDeveloper changes the corresponding code on the JSF page. When you use the binding editors to add or change a binding, JDeveloper adds the code to the JSF page, and also adds the appropriate elements to the page definition file.

33 Using Command Components to Invoke Functionality in the View Layer

This chapter describes how to invoke custom methods on ADF Business Components using ADF Faces command components and ADF Model data bindings in the Fusion web application. Specifically, it describes how you can use ADF Faces components to invoke ADF Business Components client interface methods, including how to pass parameters to a method and how to override the declarative method created when you drag and drop a method from the Data Controls panel.
This chapter includes the following sections:

About Command Components

Command components are represented by the UICommand component, which performs an action when it is activated. ADF Faces command components deliver action events when the components are activated.

Once you create a basic page and add navigation capabilities, you may want to add more complex features, such as passing parameters between pages or providing the ability to override declarative actions. **Oracle ADF** provides many features that allow you to add this complex functionality using very little actual code.

Some of the implementation methods in this chapter are intended for page-level designs. You may be able to perform many of the same functions by using task flows. See Getting Started with ADF Task Flows.

Command Component Use Cases and Examples

You can use command components to perform an action, such as passing parameters, overriding declarative actions, or performing some logic. You can bind a command component to a custom method in the Data Controls panel. For example, if you have a custom method called `deleteOrderItem()`, you can create a button bound to this method so that when the user presses this button, the order item will be deleted.

You can also use the `setPropertyListener` tag within a command component to assign values when an event fires. For example, assume you have a button that initiates a search and displays the results on another page depending on a parameter value stored in a managed bean. You can use the `setPropertyListener` property to pass this parameter value to the method that checks this value.

You can add code to a declarative method that is bound to a command component. For example, after you have created a declarative Commit button, you can add code to a managed bean and then override the Commit operation for additional processing

Additional Functionality for Command Components

You may find it helpful to understand other Oracle ADF features before you work with command components. Following are links to other functionality that may be of interest.

- For information about creating custom methods that can be added to the page as a command component, see Customizing an Application Module with Service Methods and Publishing Custom Service Methods to UI Clients.
- For information about creating command components from operations, see What Happens When You Create Command Components Using Operations.
- If you want to use EL expressions in your methods, see EL Expressions Created to Bind to Operations.
- For more information about the command button's actionListener attribute, see What Happens at Runtime: How Action Events and Action Listeners Work.
- For more information about ADF bindings, see Using ADF Model in a Fusion Web Application.
- If you are using task flows, you can use the task flow parameter passing mechanism. For more information, see Using Parameters in Task Flows.
- If you considering using task flow call methods, see Using Method Call Activities.
- If you are using managed beans, see Using a Managed Bean in a Fusion Web Application.

Creating Command Components to Execute Methods

The Data Controls panel contains custom methods that invoke application logic from anywhere within an ADF application's control flow. You can drag a method and drop it as a command button to execute the method.

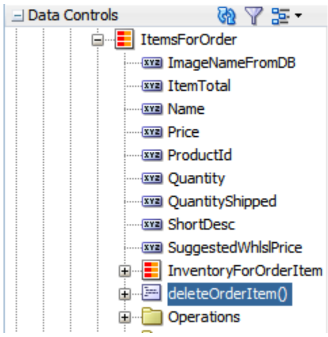
When your application contains custom methods, these methods appear in the Data Controls panel. You can then drag these methods and drop them as command buttons. When a user clicks the button, the method is executed.

For more information about creating custom methods, see Customizing an Application Module with Service Methods, and Publishing Custom Service Methods to UI Clients.

> **Note:**
>
> If you are using task flows, you can call methods directly from the task flow definition. For more information, see Using Method Call Activities.

For example, the ItemsForOrder view object in the Summit sample application for Oracle ADF contains the deleteOrderItem() method. This method updates the items in the shopping cart. To allow the user to execute this method, you drag the `deleteOrderItem()` method from the Data Controls panel, as shown in Figure 33-1.

Figure 33-1   Methods in the Data Controls Panel



In order to perform the required business logic, many methods require a value for their parameter or parameters. This means that when you create a button bound to the method, you need to create an EL expression to specify where the value for the parameter(s) is to be retrieved from. For example, if you use the `populateInventoryForProduct(String productId)` method, you need to configure the method action binding to retrieve a product ID so that the method can display the inventory for the product with that ID.

How to Create a Command Component Bound to a Custom Method

You create a command component bound to custom method by dragging the custom method from the Data Controls panel to a form.

Before you begin:

It may be helpful to have an understanding of custom methods. For more information, see Creating Command Components to Execute Methods.

You may also find it helpful to understand functionality that can be added using other command components. For more information, see Additional Functionality for Command Components.

You will need to complete these tasks:

- Create an **application module** that contains instances of the **view objects** and any custom methods that you want in your data model, as described in Creating and Modifying an Application Module. Alternatively, if you are using another type of business service, create a data control for that business service as described in Exposing Business Services with Data Controls in *Developing Applications with Oracle ADF Data Controls*.
- Create a JSF page as described in Creating a Web Page.

To create a command component bound to a custom method:

1. From the Data Controls panel, drag the method onto the page.

> **Tip:**
>
> If you are dropping a button for a method that needs to work with data in a table or form, that button must be dropped inside the table or form.

2. From the context menu, choose **Create > Methods** > **ADF Button**.

   If the method takes parameters, the Edit Action Binding dialog opens. In the Edit Action Binding dialog, enter values for each parameter or click the **Show EL Expression Builder** menu selection in the **Value** column of **Parameters** to launch the EL Expression Builder.

What Happens When You Create Command Components Using a Method

When you drop a method as a command button, JDeveloper:

- Defines a method action binding for the method.
- If the method takes any parameters, JDeveloper creates `NamedData` elements that hold the parameter values.
- Inserts code in the JSF page for the ADF Faces command component.
- Binds the button to the method using `actionListener`.
- Uses the return value from the method call.

Action Bindings

JDeveloper adds an action binding for the method. Action bindings use the `RequiresUpdateModel` property, which determines whether or not the model needs to be updated before the action is executed. For command operations, this property is set to `true` by default, which means that any changes made at the view layer must be moved to the model before the operation is executed.

Method Parameters

When you drop a method that takes parameters onto a JSF page, JDeveloper creates a method action binding. This binding is what causes the method to be executed when a user clicks the command component. When the method requires parameters to run, JDeveloper also creates `NamedData` elements for each parameter. These elements represent the parameters of the method.

For example, in the Summit sample application for ADF task flows, the `populateInventoryForProduct(String)` method action binding contains a `NamedData` element for the `String` parameter. This element is bound to the value specified when you created the action binding. The following example shows the method action binding created when you drop the `populateInventoryForProduct(String)` method, and bind the `String` parameter (named `productId`) to the appropriate variable.

```
<methodAction id="populateInventoryForProduct" RequiresUpdateModel="true"
              Action="invokeMethod"
              MethodName="populateInventoryForProduct" IsViewObjectMethod="true"
              DataControl="BackOfficeAppModuleDataControl"
```

```
            InstanceName="data.BackOfficeAppModuleDataControl.InventoryVO1">
    <NamedData NDName="productId" NDType="java.lang.String"/>
</methodAction>
```

ADF Faces Component Code

JDeveloper adds code for the ADF Faces component to the JSF page. This code is similar to the code for any other command button, as described in EL Expressions Created to Bind to Operations. However, instead of being bound to the `execute` method of the action binding for a built-in operation, the button is bound to the `execute` method of the method action binding for the method that was dropped.

EL Expressions Used to Bind to Methods

Like creating command buttons using operations, when you create a command button using a method, JDeveloper binds the button to the method using the `actionListener` attribute. The button is bound to the `execute` property of the action binding for the given method using an EL expression. This EL expression causes the binding's method to be invoked on the application module. For more information about the command button's `actionListener` attribute, see What Happens at Runtime: How Action Events and Action Listeners Work.

> **Tip:**
>
> Instead of binding a button to the `execute` method on the action binding, you can bind the button to the method in a backing bean that overrides the `execute` method. Doing so allows you to add logic before or after the original method runs. For more information, see Overriding Declarative Methods.

Like navigation operations, the `disabled` property on the button uses an EL expression to determine whether or not to display the button. The following example shows the EL expression used to bind the command button to the `populateInventoryForProduct(String)` method.

```
<af:button actionListener="#{bindings.populateInventoryForProduct.execute}"
                       text="populateInventoryForProduct"
                       disabled="#{!
bindings.populateInventoryForProduct.enabled}"
                       id="b1"/>
```

> **Tip:**
>
> When you drop a command button component onto the page, JDeveloper automatically gives it an ID based on the number of the same type of component that was previously dropped. For example, `b1` and `b2` for the first two buttons dropped. If you change the ID to something more descriptive, you must manually update any references to it in any EL expressions in the page.

Using the Return Value from a Method Call

You can also use the return value from a method call. The following example shows a custom method that returns a string value.

```
/**
 * Custom method.
*/
    public String getHelloString() {
        return ("Hello World");
    }
```

The following example shows the code in the JSF page for the command button and an `outputText` component.

```
<af:button actionListener="#{bindings.getHelloString.execute}"
        text="getHelloString"
        disabled="#{!bindings.getHelloString.enabled}"
        id="helloButtonId"/>
<af:outputText value="#{bindings.return.inputValue}"
        id="helloOutputId"/>
```

When the user clicks the command button, it calls the custom method. The method returns the string "Hello World" to be shown as the value of the `outputText` component.

What Happens at Runtime: Command Button Method Bindings

When the user clicks the button, the method binding causes the associated method to be invoked, passing in the value bound to the `NamedData` element as the parameter. For example, if a user clicks a button bound to the `populateInventoryForProduct(String productId)` method, the method takes the value of the `productId` parameter and displays the inventory for that product.

Setting Parameter Values Using a Command Component

In order to determine complex ADF application functionality, you may need to set parameters to an action on a page. A managed bean can be used to successfully pass the parameter and a method on the bean checks the parameter.

There may be cases where an action on one page needs to set parameters that will be used to determine application functionality. For example, you can create a search button on one page that will navigate to a results table on another page. But the results table will display only if a parameter value is `false`.

You can use a managed bean to pass this parameter between the pages, and to contain the method that is used to check the value of this parameter. The managed bean is instantiated as the search page is rendered, and a method on the bean checks that parameter. If it is `null` (which it will be the first time the page is rendered), the bean sets the value to `true`.

For more information about creating custom methods, see Customizing an Application Module with Service Methods, and Publishing Custom Service Methods to UI Clients.

> **Note:**
>
> If you are using task flows, you can use the task flow parameter passing mechanism. See Using Parameters in Task Flows.

A `setPropertyListener` component with `type` property set to `action`, which is nested in the command button that executed this search, is then used to set this flag to `false`, thus causing the results table to display once the search is executed. For information about using managed beans, see Using a Managed Bean in a Fusion Web Application.

How to Set Parameters Using setPropertyListener Within a Command Component

You can use the `setPropertyListener` component to set values on other objects. This component must be a child of a command component.

Before you begin:

It may be helpful to have an understanding of how `setPropertyListener` and managed bean can be used to set a value. For more information, see Setting Parameter Values Using a Command Component.

You may also find it helpful to understand functionality that can be added using other command components. For more information, see Additional Functionality for Command Components.

You will need to complete this task:

- Create an application module that contains instances of the view objects and any custom methods that you want in your data model, as described in Creating and Modifying an Application Module. Alternatively, if you are using another type of business service, create a data control for that business service as described in Exposing Business Services with Data Controls in *Developing Applications with Oracle ADF Data Controls*.
- Create a JSF page as described in Creating a Web Page.

To use the setPropertyListener component:

1. In the Applications window, double-click the web page that contains the command component to which you want to add the listener.
2. In the design editor for the page, right-click the command component and choose **Insert Inside Button > ADF Faces**.
3. In the Insert ADF Faces Item dialog, select **Set Property Listener** and click **OK**.
4. In the Insert Set Property Listener dialog, enter the parameter value in the **From** field.

   You can bring up an expression builder for this field by hovering to the right of the field, clicking the icon that appears, and choosing **Expression Builder**.
5. Enter the parameter target in the **To** field.

You can bring up an expression builder for this field by hovering to the right of the field, clicking the icon that appears, and choosing **Expression Builder**.

> **Tip:**
>
> Consider storing the parameter value on a managed bean or in scope instead of setting it directly on the resulting page's page definition file. By setting it directly on the next page, you lose the ability to easily change navigation in the future. For more information, see Using a Managed Bean in a Fusion Web Application. Additionally, the data in a binding container is valid only during the request in which the container was prepared. The data may change between the time you set it and the time the next page is rendered.

6. From the **Type** dropdown menu, select **Action**.
7. Click **OK**.

What Happens When You Set Parameters Using a setPropertyListener

The `setPropertyListener` component lets the command component set a value before it navigates to the next page. When you set the `from` attribute either to the source of the value you need to pass or to the actual value, the component will be able to access that value. When you set the `to` attribute to a target, the command component is able to set the value on the target. The following example shows the code on the JSF page for a command component that takes the value `false` and sets it as the value of the `initialSearch` flag on the `searchResults` managed bean.

```
<af:button actionListener="#{bindings.Execute.execute}"
                 text=Search>
      <af:setPropertyListener from="#{false}"
                          to="#{searchResults.initialSearch}"/>
                          type="action"/>
</af:button>
```

What Happens at Runtime: setPropertyListener for a Command Component

When a user clicks the command component, before navigation occurs, the `setPropertyListener` component sets the parameter value. As the example in What Happens When You Set Parameters Using a setPropertyListener illustrates, the `setPropertyListener` takes the value `false` and sets it as the value for the `initialSearch` attribute on the `searchResults` managed bean. Now, any component that needs to know this value in determining whether or not to render can access it using the EL expression `#{searchResults.initialSearch}`.

Overriding Declarative Methods

You can override the functionality of an existing declarative method to define a behavior that is specific to the method requirement. In order to add logic after or before the declarative method , you will need a new method and property on a managed bean that provides access to the associated action binding.

When you drop an operation or method as a command button, JDeveloper binds the button to the `execute` method for the operation or method. However, there may be occasions when you need to add logic before or after the existing logic.

> **Note:**
>
> If you are using task flows, you can call custom methods from the task flow. For more information, see Getting Started with ADF Task Flows.

JDeveloper allows you to add logic to a declarative operation by creating a new method and property on a managed bean that provides access to the binding container. By default, this generated code executes the operation or method. You can then add logic before or after this code. JDeveloper automatically binds the command component to this new method, instead of to the `execute` property on the original operation or method. Now when the user clicks the button, the new method is executed.

For example, in the `Customers.jsff` fragment of the Summit ADF task flow sample application, you can click the New icon to create a new order. However, the declarative `CreateInsert` operation requires additional processing. When the button is clicked, a new tab needs to be displayed showing the order details. To provide this processing, logic is added to the `createNewOrder` method in the `CustomersBackingBean` managed bean.

In order to override a declarative method, you must have a managed bean to hold the new method to which the command component will be bound. If your page has a backing bean associated with it, JDeveloper adds the code needed to access the binding object to this backing bean. If your page does not have a backing bean, JDeveloper asks you to create one.

How to Override a Declarative Method

You can add a command component and override its declarative methods using a managed bean.

Before you begin:

It may be helpful to have an understanding of how to override declarative methods. For more information, see Overriding Declarative Methods.

You may also find it helpful to understand functionality that can be added using other command components. For more information, see Additional Functionality for Command Components.

You will need to complete these tasks:

- Create an application module that contains instances of the view objects and any custom methods that you want in your data model, as described in Creating and Modifying an Application Module. Alternatively, if you are using another type of business service, create a data control for that business service as described in

Exposing Business Services with Data Controls in *Developing Applications with Oracle ADF Data Controls*.

- Create a JSF page as described in Creating a Web Page.

> **Note:**
>
> You cannot override the declarative method if the command component currently has an EL expression as its value for the `Action` attribute, because JDeveloper will not overwrite an EL expression. You must remove this value before continuing.

To override a declarative method:

1. Drag the operation or method to be overridden onto the JSF page and drop it as a UI command component.

   The component is created and bound to the associated binding object in the ADF Model layer with the `ActionListener` attribute.

   For more information about creating command components using methods on the Data Controls panel, see Creating Command Components to Execute Methods.

   For more information about creating command components from operations, see What Happens When You Create Command Components Using Operations.

2. On the JSF page, double-click the component.

3. In the Bind Action Property dialog, identify the backing bean and the method to which you want to bind the component, using one of the following techniques:

   - If auto-binding has been enabled on the page (by selecting the **Automatically Expose UI Components in a New Managed Bean** option that is available on the Managed Bean tab of the Create JSF Page wizard), the backing bean is already selected for you.

     – To create a new method, enter a name for the method in the **Method** field, which initially displays a default name.

       or

     – To use an existing method, select a method from the dropdown list in the **Method** field.

     – Select **Use ADF Binding for Generation**.

   - If the page is not using auto-binding, you can select from an existing backing bean or create a new one.

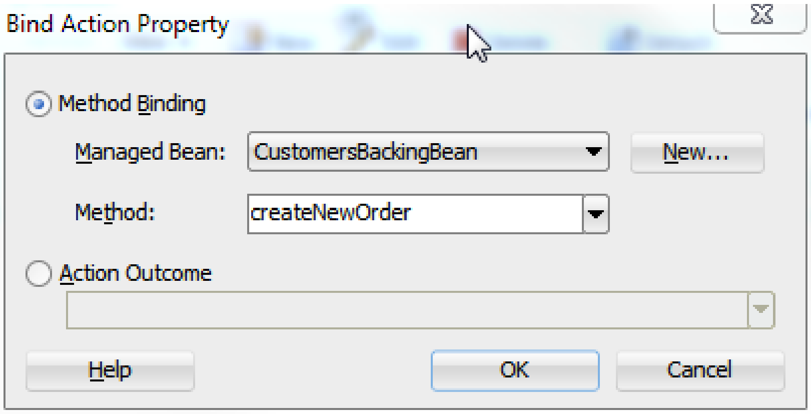
     – Click **New** to create a new backing bean. In the Create Managed Bean dialog, name the bean and the class, and set the bean's scope.

       or

     – Select an existing backing bean and method from the dropdown lists.

     Figure 33-2 shows the Bind Action Property dialog with a method from a managed bean specified for the method binding.

Figure 33-2 Bind Action Property Dialog



● Method Binding

Managed Bean: CustomersBackingBean

Method: createNewOrder

○ Action Outcome

> **Note:**
>
> Whenever there is a value for the `ActionListener` attribute on the command component, JDeveloper understands that the button is bound to the `execute` property of a binding. If you have removed that binding, you will not be given the choice to generate the ADF binding code. You will need to insert the code manually, or to set a dummy value for the `ActionListener` before double-clicking the command component.

4. After identifying the backing bean and method, click **OK** in the Bind Action Property dialog

   The managed bean opens in the source editor. The following example shows the code inserted into the bean in the Summit ADF task flow sample application. In this example, a command button is bound to the `CreateInsert` operation.

```
public void createNewOrder() {
        BindingContainer bindings = getBindings();
        OperationBinding operationBinding =
              bindings.getOperationBinding("CreateInsert");
        Object result = operationBinding.execute();
        if (!operationBinding.getErrors().isEmpty()) {
            return null;
        }
}
```

5. You can now add logic either before or after the binding object is accessed, as shown by the code in bold in the following example.

```
public void createNewOrder() {
//This method is called when creating a new order
//to create the record and switch the tab
  getSdi3().setDisclosed(false);
  getSdi4().setDisclosed(true);
  BindingContainer bindings = getBindings();
  OperationBinding operationBinding =
       bindings.getOperationBinding("CreateInsert");
  operationBinding.execute();
  AdfFacesContext adfFacesContext =
AdfFacesContext.getCurrentInstance();
```

```
  adfFacesContext.addPartialTarget(getSdi4());
}
```

In addition to any processing logic, you may also want to write conditional logic to return one of multiple outcomes. For example, you might want to return `null` if there is an error in the processing, or another outcome value if the processing was successful. A return value of `null` causes the navigation handler to forgo evaluating navigation cases and to immediately redisplay the current page.

> **Tip:**
>
> To trigger a specific navigation case, the outcome value returned by the method must exactly match the outcome value in the navigation rule, including case.

The command button is now bound to this new method using the `Action` attribute instead of the `ActionListener` attribute. If a value had previously existed for the `Action` attribute (such as an outcome string), that value is added as the return for the new method. If there was no value, the return is kept as `null`.

What Happens When You Override a Declarative Method

When you override a declarative method, JDeveloper adds a managed property to your backing bean with the managed property value of `#{bindings}` (the reference to the binding container), and it adds a strongly typed bean property to your class of the `BindingContainer` type, which the JSF runtime will then set with the value of the managed property expression `#{bindings}`. JDeveloper also adds logic to the UI command action method. This logic includes the strongly typed `getBindings()` method used to access the current binding container.

The code does the following:

- Accesses the binding container.
- Finds the binding for the associated method, and executes it.
- Adds a return for the method that can be used for navigation. By default, the return is `null`. If an outcome string had previously existed for the button's `Action` attribute, that attribute is used as the return value. You can change this code as needed.

JDeveloper automatically rebinds the UI command component to the new method using the `Action` attribute, instead of the `ActionListener` attribute. The following example shows the code when a `CreateInsert` operation is declaratively added to a page.

```
<af:button actionListener="#{bindings.CreateInsert.execute}"
                 text="CreateInsert"
                 disabled="#{!bindings.CreateInsert.enabled}"/>
```

The following example shows the code for the button tag after the method on the page's backing bean is overridden. Note that the `action` attribute is now bound to the backing bean's method.

```
<af:button text="New"
           action="#{backingBeanScope.CustomersBackingBean.createNewOrder}"/>
```

> **Tip:**
>
> If when you click the button that uses the overridden method you receive this error:
>
> `SEVERE: Managed bean main_bean could not be created The scope of the referenced object: '#{bindings}' is shorter than the referring object`
>
> it is because the managed bean that contains the overriding method has a scope that is greater than `request` (that is, either `session` or `application`). Because the data in the binding container referenced in the method has a scope of `request`, the scope of this managed bean must be set to the same or a lesser scope.

34
Displaying Master-Detail Data

This chapter describes how to create master-detail objects with data modeled from ADF Business Components, using ADF data controls and ADF Faces components. It describes how to display master-detail data by using prebuilt master-detail widgets, tables, trees and tree tables and how to work with selection events.
This chapter includes the following sections:

For information about using a selection list to populate a collection with a key value from a related master or detail collection, see Creating Databound Selection Lists and Shuttles .

About Displaying Master-Detail Data

A master-detail relationship is a one-to-many type relationship. Master-detail relationships using Oracle ADF allows you to view data from related tables at the same time.

There are many instances where data needs to be presented in a hierarchical manner. In a **master-detail relationship**, when the user changes the selected item in the master data object, the data set displayed in the detail data object changes with it. For example, selecting the television categories in the master object would display all the models of televisions in the detail object.

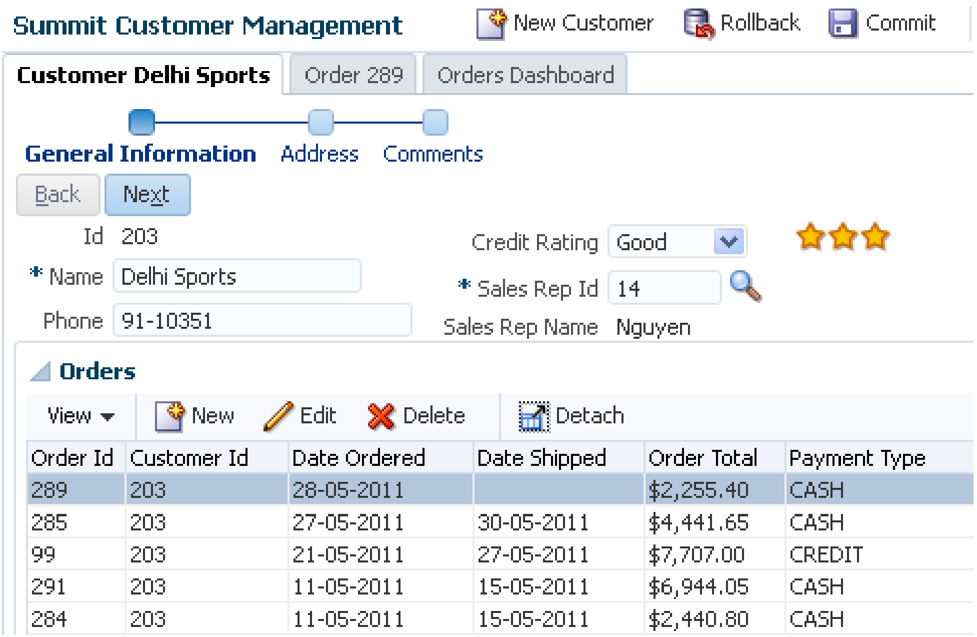
When using ADF Model in combination with ADF Faces UI components, you can declaratively create master-detail pages that display the data from multiple objects simultaneously when those objects have a master-detail relationship defined. ADF iterator bindings automatically manage the synchronization of the detail data objects displayed for a selected master data object.

In ADF Business Components, a master-detail relationship is established when a view link is created to associate two **view object instances**. As described in About View Objects, a **view link** represents the relationship between two view objects, which is usually, but not necessarily, based on a foreign-key relationship between the underlying data tables. The view link associates a row of one view object instance (the master object) with one or more rows of another view object instance (the detail object).

Master-Detail Tables, Forms, and Trees Use Cases and Examples

When objects have a master-detail relationship, you can declaratively create pages that display the data from both objects simultaneously. For example, the page shown in Figure 34-1 displays an entry for a customer in a form at the top of the page and a table of the customer's orders at the bottom of the page. This is possible because the objects have a master-detail relationship. In this example, the `Customers` view object is the master object and `OrdersForCustomer` is the detail object. ADF iterators automatically manage the synchronization of the detail data objects displayed for a selected master data object. Iterator bindings simplify building user interfaces that allow scrolling and paging through collections of data and drilling-down from summary to detail information.

Figure 34-1   Master Form and Detail Table



Id 203
* Name Delhi Sports
Phone 91-10351
Credit Rating Good
* Sales Rep Id 14
Sales Rep Name Nguyen

Order Id	Customer Id	Date Ordered	Date Shipped	Order Total	Payment Type
289	203	28-05-2011		$2,255.40	CASH
285	203	27-05-2011	30-05-2011	$4,441.65	CASH
99	203	21-05-2011	27-05-2011	$7,707.00	CREDIT
291	203	11-05-2011	15-05-2011	$6,944.05	CASH
284	203	11-05-2011	15-05-2011	$2,440.80	CASH

You display master and detail objects in forms, tables, trees, and tree tables. You can display these objects on the same page or on separate pages. For example, you can display the master object in a table on one page and detail objects in a read-only form on another page.

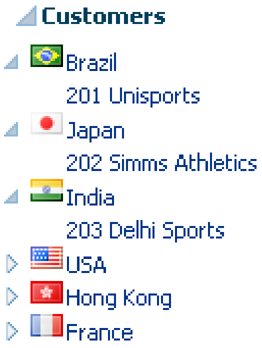
A master object can have many detail objects, and each detail object can in turn have its own detail objects, down to many levels of depth. If one of the detail objects in this hierarchy is dropped from the Applications window as a master-detail form on a page, only its immediate parent master object displays on the page. The hierarchy will not display all the way up to the topmost parent object.

If you display the detail object as a tree or tree table object, it is possible to display the entire hierarchy with multiple levels of depth, starting with the topmost master object, and traversing detail children objects at each node.

Figure 34-1 shows an example of a master form with detail table. When the user navigates through the customers using the command buttons, the table lists the orders that the customer has made.

Figure 34-2 shows a tree that has two levels of nodes. You can use trees to display hierarchical information. In this example, a tree is used to display customers grouped by countries. The user can expand nodes to traverse down a branch of the tree to access the leaf items.

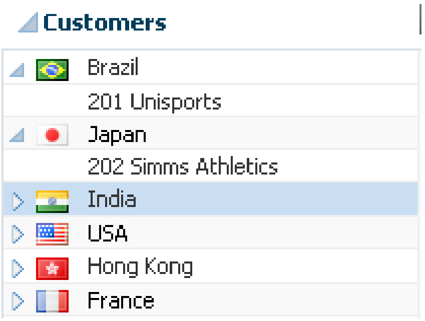
Figure 34-2 Multi-level Tree



Customers
- Brazil
  - 201 Unisports
- Japan
  - 202 Simms Athletics
- India
  - 203 Delhi Sports
- USA
- Hong Kong
- France

Figure 34-3 shows the same grouping of customers by country, but using a tree table instead of a tree. In addition to the hierarchical node levels of a tree, a tree table can also display column information for each node. You can also focus on subtrees as well as collapsing and expanding the elements.

Figure 34-3 Tree Table



Customers
Brazil
201 Unisports
Japan
202 Simms Athletics
India
USA
Hong Kong
France

Additional Functionality for Master-Detail Tables, Forms, and Trees

You may find it helpful to understand other **Oracle ADF** features before you configure or use the ADF Model layer. Additionally, you may want to read about what you can do with your model layer configurations. Following are links to other functionality that may be of interest.

- **ADF view objects and view links**: Much of how the components display and function in the form is controlled by the corresponding view objects. For more information about creating view objects and establishing a master-detail hierarchy in your data model with view links, see Working with Multiple Tables in a Master-Detail Hierarchy.

- **ADF application modules**: The Data Controls panel, from which you drag databound components to your pages, is populated with representations of view objects that you have added to application modules. See Implementing Business Services with Application Modules.
- **Adapter-based data controls**: If you are using other types of business services, such as EJB components or web services, you can create data control for those business services as described in Creating and Configuring EJB Data Controls of the *Developing Applications with Oracle ADF Data Controls*.
- **ADF Model and data binding**: When you create forms in an ADF web application, you use ADF Model and data binding. See Using ADF Model in a Fusion Web Application.
- **ADF Faces**: You also use ADF Faces UI components. For detailed information about developing with ADF Faces, see Introduction to ADF Faces in *Developing Web User Interfaces with Oracle ADF Faces*.
- **ADF forms and tables**: If you want to modify the default forms or tables, see Creating Basic Forms Using Data Control Collections or Creating a Basic Table.
- **Task flows**: If your form takes part in a transaction, then you may need to use an ADF task flow to invoke certain operations before or after the form is rendered. See Creating ADF Task Flows.
- **Event handling**: You may need to write listener methods to process client-side events. See Handling Events in *Developing Web User Interfaces with Oracle ADF Faces*.
- **Managed beans**: You may need to create managed beans for event handling and other front-end processing. For information about creating and using managed beans, see Using a Managed Bean in a Fusion Web Application.

Prerequisites for Master-Detail Tables, Forms, and Trees

A master-detail relationship relates two view object instances by a view link. A view link represents the relationship between two view objects, which is usually, but not necessarily, based on a foreign-key relationship between the underlying data tables, forms, and trees.

Master-detail tables, forms, trees, and tree tables require a master-detail relationship to be established in the data model. That master-detail relationship is then reflected in the Data Controls panel, from where you can drag and drop objects to declaratively create pages that display master-detail data.

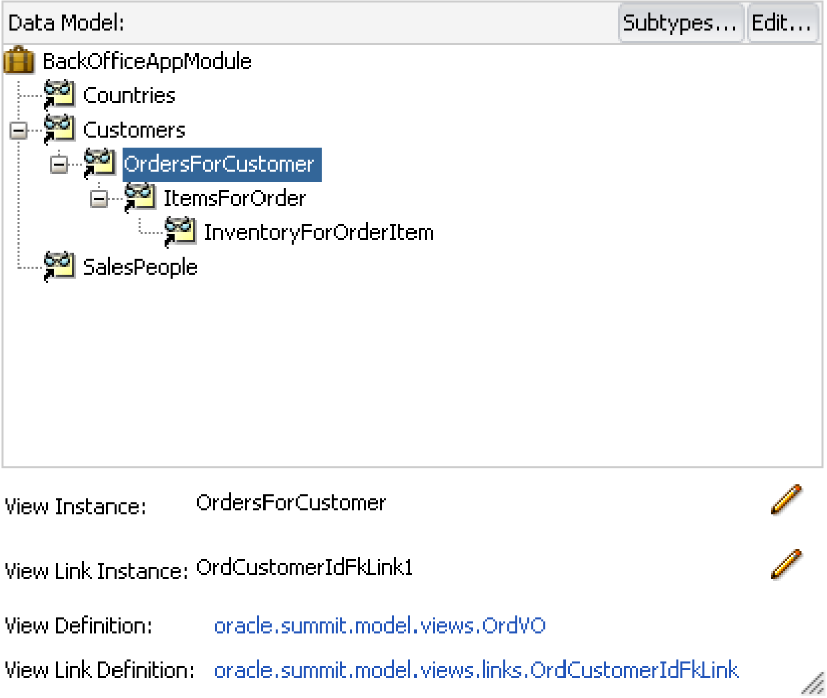
In an ADF Business Components application, you indicate a master-detail relationship by creating a view link between two view objects. Then you add the master view object and the detail view object instances to the application module data model. For more information, see How to Enable Active Master-Detail Coordination in the Data Model.

Note:

For data controls based on other business services, such as web services or EJB session beans, the master-detail relationship is typically inferred from standard mechanisms within those services, such as definition files for web services or `JoinColumn` annotations in JPA-based beans.

For example, in the Summit sample application for Oracle ADF, there is a view link from the `CustomerVO` view object to the `OrdVO` view object based on the `CustomerId` attribute, both contained in the application module data model, as shown in Figure 34-4. A change in the current row of the master view object instance causes the row set of the detail view object instance to refresh to include the details for the current master.

Figure 34-4 View Link between CustomerVO and OrdVO View Objects



View Instance: OrdersForCustomer

View Link Instance: OrdCustomerIdFkLink1

View Definition: oracle.summit.model.views.OrdVO

View Link Definition: oracle.summit.model.views.links.OrdCustomerIdFkLink

Note:

In the Summit ADF sample application, the view link between the `CustomerVO` view object and `OrdVO` view object is a one-way relationship. If the view link were bidirectional and both sets of master and detail view objects were added to the application module data model, then the Data Controls panel would also display the `OrdersForCustomer` collection at the same node level as the `Customers` collection, and the detail instance of the `OrdersForCustomer` collection as a child of the `Customers` collection.

How to Identify Master-Detail Objects on the Data Controls Panel

You can identify master-detail objects by looking at the hierarchy in the Data Controls panel.

Before you begin:

It may help to understand the basics of master-detail relationships. For more information, see Prerequisites for Master-Detail Tables, Forms, and Trees.

You may also find it useful to understand additional functionality that can be used with master-detail tables and trees. For more information, see Additional Functionality for Master-Detail Tables, Forms, and Trees.

You will need to complete this task:

- Create an **application module** that contains instances of the view objects and view links that establish the desired master-detail hierarchy, as described in Creating and Modifying an Application Module. Alternatively, if you are using another type of business service, create a data control for that business service as described in Exposing Business Services with Data Controls in *Developing Applications with Oracle ADF Data Controls*.

To identify master-detail objects:

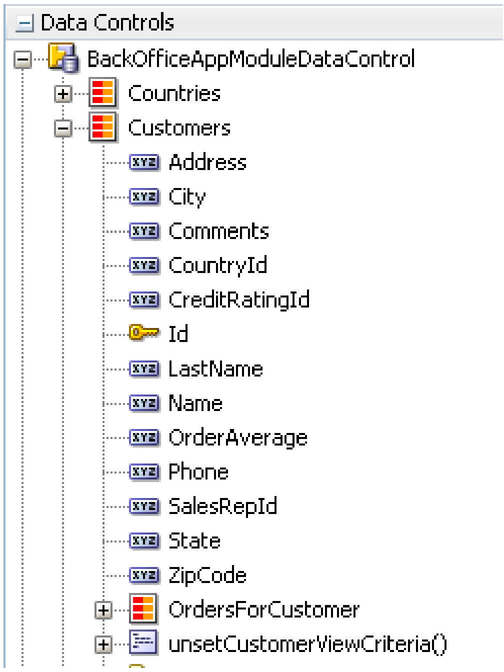
- In the Applications window, open the Data Controls panel, expand any collection nodes, and look for collection child nodes of those collections.

Any child collection node of a collection node represents the detail part of a master-detail relationship with the parent node. A parent collection can be part of a master-detail relationship with multiple child collections. And collections can be nested in multiple layers, meaning that a collection can be the detail part of one master-detail relationship and the master part of another master-detail relationship.

Figure 34-5 shows two master-detail related collections in the Data Controls panel of the Summit ADF sample application. The `Customers` collection is an instance of the `CustomerVO` view object, and the `OrdersForCustomer` collection, which appears as a child of the `Customers` collection, is an instance of the `OrdVO` view object.

> **Note:**
>
> The master-detail hierarchy displayed in the Data Controls panel does not reflect the cardinality of the relationship (that is, one-to-many, one-to-one, many-to-many). The hierarchy simply shows which collection (the master) is being use to retrieve one or more objects from another collection (the detail).

Figure 34-5 Master-Detail Objects in the Data Controls Panel



```
Data Controls
BackOfficeAppModuleDataControl
  Countries
  Customers
    Address
    City
    Comments
    CountryId
    CreditRatingId
    Id
    LastName
    Name
    OrderAverage
    Phone
    SalesRepId
    State
    ZipCode
    OrdersForCustomer
    unsetCustomerViewCriteria()
```

The master-detail hierarchy on the Data Controls panel reflects the hierarchy defined in the application module data model, as shown in Figure 34-4. For more information about the icons displayed on the Data Controls panel, see How to Use the Data Controls Panel.

Using Tables and Forms to Display Master-Detail Objects

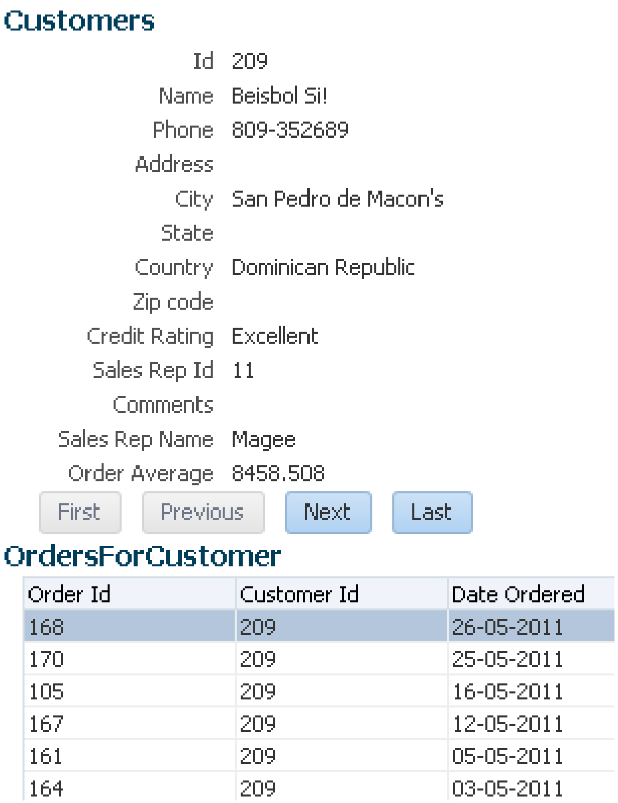
The Data Controls panel enables you to create both the master and detail widgets on one page with a single declarative action using prebuilt master-detail forms and tables. If you do not want to use the prebuilt master-detail widgets, you can drag and drop the master and detail objects individually from the Data Controls panel as tables and forms.

You can create master-detail functionality on a page with form and table components. You can do so either by creating individual form and table components or by using pre-built widgets that encompass both the master and detail components.

The pre-built widgets enable you to create a read-only master-detail browse page in a single declarative action using the Data Controls panel. All you have to do is drop the detail collection on the page and choose the type of widget you want to use. The generated components include range navigation that enables the end user to scroll through the data objects in collections. You can delete unwanted attributes by removing the text field or column from the page.

Figure 34-6 shows an example of prebuilt master-detail widget, which displays general order information in a form at the top of the page and order details in a table at the bottom of the page. When the user clicks the **Next** button to scroll through the records in the master data at the top of the page, the page automatically updates to display the detail data for the currently displayed master record.

Figure 34-6 Prebuilt Data Controls Panel Master-Detail Widget



Customers

Id	209
Name	Beisbol Si!
Phone	809-352689
Address	
City	San Pedro de Macon's
State	
Country	Dominican Republic
Zip code	
Credit Rating	Excellent
Sales Rep Id	11
Comments	
Sales Rep Name	Magee
Order Average	8458.508

OrdersForCustomer

Order Id	Customer Id	Date Ordered
168	209	26-05-2011
170	209	25-05-2011
105	209	16-05-2011
167	209	12-05-2011
161	209	05-05-2011
164	209	03-05-2011

There are some cases when the pre-built master-detail UI components that JDeveloper provides cannot provide the functionality you require. For example, you may need to make a component editable or bind components programatically instead of using the master-detail UI components. In these cases, you can create master-detail pages by creating forms and tables separately. When components are created from a collection and nested collection, the nested collection's iterator binding manages coordination between the master and detail component automatically. See How to Create Master-Detail Forms from Separate Components and How to Display Master-Detail Components on Separate Pages.

> **Note:**
>
> You can also use the `detailStamp` facet of a master table to display the detail for selected rows. See How to Create a Dynamic Table with a detailStamp Facet for information on doing so with a dynamic component.

How to Display Master-Detail Objects in Tables and Forms Using Master-Detail Widgets

The Data Controls panel enables you to create both the master and detail widgets on one page with a single declarative action using prebuilt master-detail forms and tables. These prebuilt master-detail widgets are read-only. For information about displaying master and detail data on separate pages, see How to Create Master-Detail Forms from Separate Components. For information about displaying master and detail data on separate pages, see How to Display Master-Detail Components on Separate Pages.

Before you begin:

It may help to understand the options that are available to you when you create a master-detail table and form. For more information, see Using Tables and Forms to Display Master-Detail Objects.

You may also find it useful to understand additional functionality that can be used with master-detail tables and trees. For more information, see Additional Functionality for Master-Detail Tables, Forms, and Trees.

You will need to complete these tasks:

- Create an application module that contains instances of the view objects and view links that establish the desired master-detail hierarchy, as described in Creating and Modifying an Application Module. Alternatively, if you are using another type of business service, create a data control for that business service as described in Exposing Business Services with Data Controls in *Developing Applications with Oracle ADF Data Controls*.
- Create a JSF page as described in Creating a Web Page.

To create a master-detail page using the prebuilt ADF master-detail forms and tables:

1. From the Data Controls panel, locate the detail object, as described in How to Identify Master-Detail Objects on the Data Controls Panel.
2. Drag and drop the detail object onto the JSF page.
3. From the **Master-Detail** submenu of the context menu, choose one of the following components:
   - **ADF Master Table, Detail Form**: Displays the master objects in a read-only table and the detail objects in a read-only form under the table.

     When a specific data object is selected in the master table, the first related detail data object is displayed in the form below it. The user must use the form navigation to scroll through each subsequent detail data object.
   - **ADF Master Form, Detail Table**: Displays the master objects in a read-only form and the detail objects in a read-only table under the form.

     When a specific master data object is displayed in the form, the related detail data objects are displayed in a table below it.
   - **ADF Master Form, Detail Form**: Displays the master and detail objects in separate read-only forms.

     When a specific master data object is displayed in the top form, the first related detail data object is displayed in the form below it. The user must use the form navigation to scroll through each subsequent detail data object.
   - **ADF Master Table, Detail Table**: Displays the master and detail objects in separate read-only tables.

     When a specific master data object is selected in the top table, the first set of related detail data objects is displayed in the table below it.

If you want to modify the default forms or tables, see Modifying the UI Components and Bindings on a Form and Modifying the Attributes Displayed in the Table.

How to Create Master-Detail Forms from Separate Components

Instead of using the pre-built master-detail widgets to create master-detail forms, you can create table and form components separately from data control objects that have a master-detail relationship and take advantage of the detail object's iterator bindings to provide the master-detail coordination. Creating master-detail forms from separate components enables you more flexibility in how the components and their bindings are generated.

Before you begin:

It may help to understand the options that are available to you when you create a master-detail table and form. For more information, see Using Tables and Forms to Display Master-Detail Objects.

You may also find it useful to understand additional functionality that can be used with master-detail tables and trees. For more information, see Additional Functionality for Master-Detail Tables, Forms, and Trees.

You will need to complete these tasks:

- Create an application module that contains instances of the view objects and view links that establish the desired master-detail hierarchy, as described in How to Create a Master-Detail Hierarchy Based on View Objects Alone. Alternatively, if you are using another type of business service, create a data control for that business service as described in Exposing Business Services with Data Controls in *Developing Applications with Oracle ADF Data Controls*.
- Create a JSF page as described in Creating a Web Page.

To create a master-detail form with individual components:

1. Create the detail widget by dragging a child collection from the Data Controls panel and dropping it as a table or a form on the JSF page or fragment that corresponds to the detail view activity. For more information, see How to Identify Master-Detail Objects on the Data Controls Panel.

   For more information on creation of the individual components, see Creating Basic Forms Using Data Control Collections and Creating a Basic Table.
2. Create the master widget by dragging the collection that is the parent of the collection in step 1 and dropping it as a table or a form on the JSF page or fragment that corresponds to the master view activity.

What Happens When You Create Master-Detail Tables and Forms

When you drag and drop a collection from the Data Controls panel, JDeveloper does many things for you, including adding code to the JSF page and the corresponding entries in the page definition file. For a general description of what happens and what is created when you use the Data Controls panel, see What Happens When You Use the Data Controls Panel.

Code Generated in a Master-Detail JSF Page

The JSF code generated for a prebuilt master-detail widget is similar to the JSF code generated when you use the Data Controls panel to create a read-only form or table. If you are building your own master-detail widgets, you might want to consider including similar components that are automatically included in the prebuilt master-detail tables and forms.

The tables and forms in the prebuilt master-detail widgets include a `panelHeader` tag that contains the name of the data control populating the master form or table. You can change this label as needed using a string or an EL expression that binds to a resource bundle.

If there is more than one object in a collection, a form in a prebuilt master-detail widget includes four `button` tags for range navigation: `First`, `Previous`, `Next`, and `Last`. These range navigation buttons enable the user to scroll through the objects in the collection. The `actionListener` attribute of each button is bound to a data control operation, which performs the navigation. The `execute` property used in the `actionListener` binding, invokes the operation when the button is clicked. (If the form displays a single object, JDeveloper automatically omits the range navigation components.) For more information about range navigation, see Incorporating Range Navigation into Forms.

> **Tip:**
>
> If you drop an **ADF Master Table, Detail Form** or **ADF Master Table, Detail Table** widget on the page, the parent tag of the detail component (for example, `panelHeader` tag or `table` tag) automatically has the `partialTriggers` attribute set to the `id` of the master component (see How to Create an Input Table for more information about partial triggers). At runtime, the `partialTriggers` attribute causes only the detail component to be rerendered when the user makes a selection in the master component, which is called **partial page rendering**. Partial page rendering is used because only the detail component needs to be rerendered to display the new data. The table does not need to be rerendered when the user simply makes a selection in the facet.

Binding Objects Defined in a Master-Detail Page Definition File

The following example shows the page definition file created for a master-detail page that was created by dropping the `OrdersFromCustomer` collection, which is a detail object under the `Customers` object, on the page as an **ADF Master Form, Detail Table**.

The `executables` element defines two iterators: one for the master object and one for the detail object. At runtime, the data model and the **row set iterator** for the detail view object instance keep the row set of the detail view object refreshed to the correct set of rows for the current master row as that current row changes. For more information, see What Happens at Runtime: ADF Iterator for Master-Detail Tables and Forms.

The bindings element defines the value bindings for the form and the table. The attribute bindings that populate the text fields in the form are defined in the attributeValues elements. The id attribute of the attributeValues element contains the name of each data attribute, and the IterBinding attribute references an iterator binding to display data from the master object in the text fields.

The range navigation buttons in the form are bound to the action bindings defined in the action elements. As in the attribute bindings, the IterBinding attribute of the action binding references the iterator binding for the master object.

The table, which displays the detail data, is bound to the table binding object defined in the table element. The IterBinding attribute references the iterator binding for the detail object.

For more information about the elements and attributes of the page definition file, see pageNamePageDef.xml.

```
<executables>
  <variableIterator id="variables"/>
  <iterator Binds="Customers" RangeSize="25"
DataControl="BackOfficeAppModuleDataControl" id="CustomersIterator"
            ChangeEventPolicy="ppr"/>
  <iterator Binds="OrdersForCustomer" RangeSize="25"
DataControl="BackOfficeAppModuleDataControl"
            id="OrdersForCustomerIterator" ChangeEventPolicy="ppr"/>
</executables>
<bindings>
  <action IterBinding="CustomersIterator" id="First" RequiresUpdateModel="true"
Action="first"/>
  <action IterBinding="CustomersIterator" id="Previous"
RequiresUpdateModel="true" Action="previous"/>
  <action IterBinding="CustomersIterator" id="Next" RequiresUpdateModel="true"
Action="next"/>
  <action IterBinding="CustomersIterator" id="Last" RequiresUpdateModel="true"
Action="last"/>
  <attributeValues IterBinding="CustomersIterator" id="Id">
    <AttrNames>
      <Item Value="Id"/>
    </AttrNames>
  </attributeValues>
  <attributeValues IterBinding="CustomersIterator" id="Name">
    <AttrNames>
      <Item Value="Name"/>
    </AttrNames>
  </attributeValues>
  <attributeValues IterBinding="CustomersIterator" id="Phone">
    <AttrNames>
      <Item Value="Phone"/>
    </AttrNames>
  </attributeValues>
. . .
  <tree IterBinding="OrdersForCustomerIterator" id="OrdersForCustomer">
    <nodeDefinition DefName="oracle.summit.model.views.OrdVO"
                    Name="OrdersForCustomer0">
      <AttrNames>
        <Item Value="Id"/>
        <Item Value="CustomerId"/>
        <Item Value="DateOrdered"/>
...
      </AttrNames>
    </nodeDefinition>
  </tree>
</bindings>
```

> **Note:**
>
> For data controls that are based on other types of business services, such as EJB session beans, `accessorIterator` elements are used instead of `iterator` elements to handle the iterator binding.

What Happens at Runtime: ADF Iterator for Master-Detail Tables and Forms

At runtime, an ADF iterator determines which row from the master table object to display in the master-detail form. When the form first displays, the first master table object row appears highlighted in the master section of the form. Detail table rows that are associated with the master row display in the detail section of the form.

As described in Binding Objects Defined in a Master-Detail Page Definition File, ADF iterators are associated with underlying `rowsetIterator` objects. These iterators manage which data objects, or *rows*, currently display on a page. At runtime, the row set iterators manage the data displayed in the master and detail components.

Both the master and detail row set iterators listen to row set navigation events, such as the user clicking the range navigation buttons, and display the appropriate row in the UI. In the case of the default master-detail components, the row set navigation events are the command buttons on a form (**First**, **Previous**, **Next**, **Last**).

The row set iterator for the detail collection manages the synchronization of the detail data with the master data. Because of the underlying view link from the master view object to the detail view object, the detail row set iterator listens for row navigation events in both the master and detail collections. If a row set navigation event occurs in the master collection, the detail row set iterator automatically executes and returns the detail rows related to the current master row.

How to Display Master-Detail Components on Separate Pages

The default master-detail components display the master-detail data on a single page. However, using the master and detail objects on the Data Controls panel, you can also display the collections on separate pages (or on separate fragments on the same page), and still have the binding iterators manage the synchronization of the master and detail objects.

Before you begin:

It may help to understand the options that are available to you when you create a master-detail table and form. For more information, see Using Tables and Forms to Display Master-Detail Objects.

You may also find it useful to understand additional functionality that can be used with master-detail tables and trees. For more information, see Additional Functionality for Master-Detail Tables, Forms, and Trees.

You will need to complete this task:

- Create an application module that contains instances of the view objects and view links that establish the desired master-detail hierarchy, as described in How to Create a Master-Detail Hierarchy Based on View Objects Alone. Alternatively, if you are using another type of business service, create a data control for that business service as described in Exposing Business Services with Data Controls in *Developing Applications with Oracle ADF Data Controls*.

To create master and detail objects on separate pages:

1. Create an ADF task flow as described in How to Create a Task Flow.

   If you plan to use page fragments to hold the master and detail components, select the Create with Page Fragments checkbox in this step.

2. Add two view activities to the task flow, one for the master view and one for the detail view, as described in How to Add an Activity to a Task Flow.

3. In the task flow definition file, add a control flow case from the master view to the detail view, and a control flow case to return from the detail view to the master view. For more information, see How to Add a Control Flow Rule to a Task Flow.

4. Create two JSF pages or two JSF fragments to hold the master and detail components.

   You can create a page or fragment for a view activity by clicking on the view activity in the overview editor for task flow.

   If you already have pages or fragments that you want to map to the view activities, you can do so by selecting the view activity in the overview editor for the task flow, navigating to the Properties window, and specifying the JSF page or fragment in the `Page` property.

5. Create the detail component by dragging a child collection from the Data Controls panel and dropping it as a table or a form on the JSF page or fragment that corresponds to the detail view activity. For more information, see How to Identify Master-Detail Objects on the Data Controls Panel.

   For more information on creating forms and tables, see Creating Basic Forms Using Data Control Collections and Creating a Basic Table.

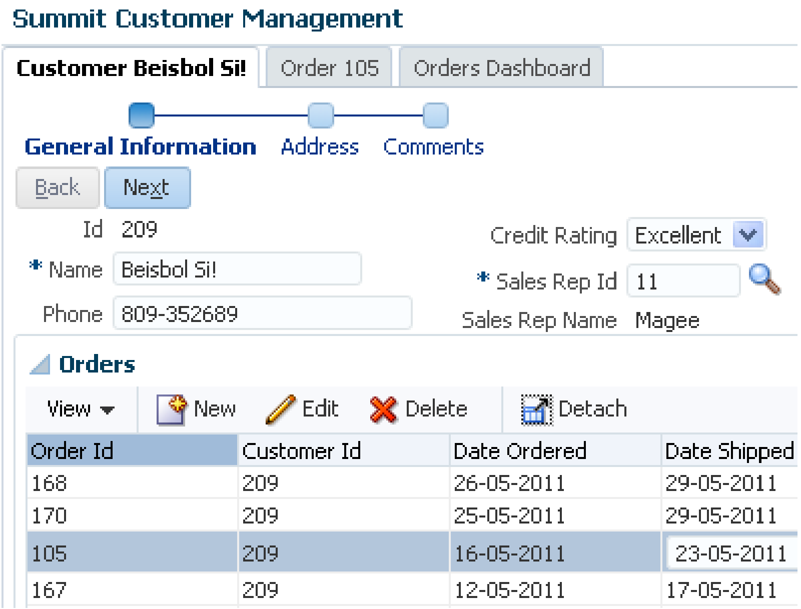
6. Create the master component by dragging the collection that is the parent of the collection in step 5 and dropping it as a table or a form on the JSF page or fragment that corresponds to the master view activity.

7. Add command buttons or links to each page.

8. In the `action` attribute of each button, specify the control flow case that is used to navigate to the other page. You can get this value from the `from-outcome` property of the corresponding control flow case.

For example, in the Summit ADF sample application, the `Customers.jsff` fragment contains a table created from the `OrdersFromCustomer` collection. That table serves as both a detail view of customer orders and a master view for the Order Items table that is created from the `ItemsFromOrder` collection and is located in `Orders.jsff` (and appears on the Order tab). Figure 34-7 shows the Orders table as a detail view for a customer, and Figure 34-8 shows the order that is selected in that table and displays that order's detail. Though the `OrdersFromCustomer` collection is used in both the `Customers.jsff` and `Orders.jsff` fragments, it does not have to be used in both places to maintain master-detail coordination. The iterator binding for the `ItemsFromOrder` maintains the master -detail coordination, even across the page fragments.

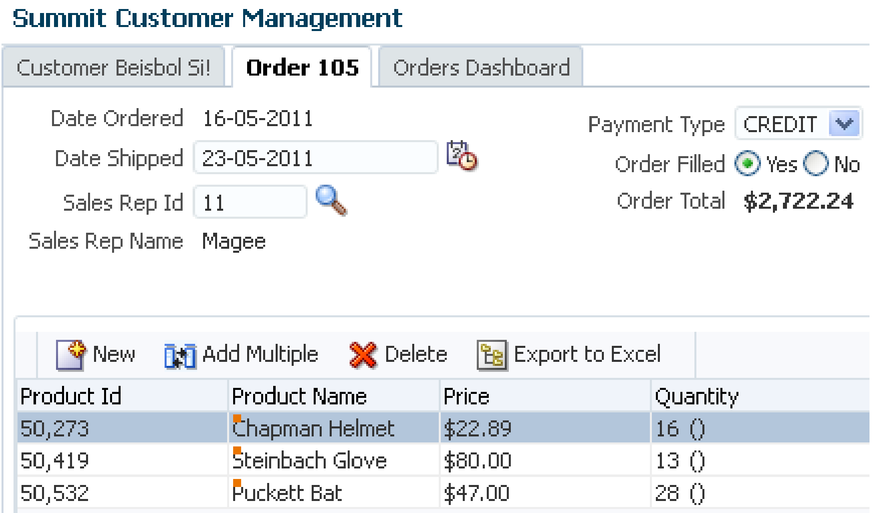
Figure 34-7 Master View on One Page Fragment



Id 209
* Name Beisbol Si!
Phone 809-352689
Credit Rating Excellent
* Sales Rep Id 11
Sales Rep Name Magee

Order Id	Customer Id	Date Ordered	Date Shipped
168	209	26-05-2011	29-05-2011
170	209	25-05-2011	29-05-2011
105	209	16-05-2011	23-05-2011
167	209	12-05-2011	17-05-2011

Figure 34-8 Detail View on a Separate Page Fragment



Date Ordered 16-05-2011
Date Shipped 23-05-2011
Sales Rep Id 11
Sales Rep Name Magee
Payment Type CREDIT
Order Filled ● Yes ○ No
Order Total $2,722.24

Product Id	Product Name	Price	Quantity
50,273	Chapman Helmet	$22.89	16 ()
50,419	Steinbach Glove	$80.00	13 ()
50,532	Puckett Bat	$47.00	28 ()

Using Trees to Display Master-Detail Objects

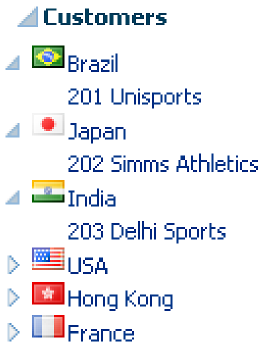
The ADF Faces tree component can be used to display model-driven master-detail data relationships in a hierarchical manner. In this case, the parent node of the tree indicates the master object, while the child nodes of the tree are detail objects.

In addition to tables and forms, you can also display master-detail data in hierarchical trees. The ADF Faces `tree` component is used to display hierarchical data. It can display multiple root nodes that are populated by a binding on a master object. Each root node in the tree may have any number of branches, which are populated by bindings on detail objects. A tree can have multiple levels of nodes, each representing a detail object of the parent node. Each node in the tree is indented to show its level in the hierarchy.

The `tree` component includes mechanisms for expanding and collapsing the tree nodes; however, it does not have focusing capability. If you need to use focusing, consider using the ADF Faces `treeTable` component (for more information, see Using Tree Tables to Display Master-Detail Objects). By default, the icon for each node in the tree is a folder; however, you can use your own icons for each level of nodes in the hierarchy.

Figure 34-9 shows an example of a tree located in the `Customers.jsff` fragment of the Summit ADF sample application. The tree displays two levels of nodes: root and branch. The root nodes display countries. The branch nodes display customers.

Figure 34-9 Databound ADF Faces Tree



How to Display Master-Detail Objects in Trees

A *tree* consists of a hierarchy of nodes, where each subnode is a branch off a higher level node. Each node level in a databound ADF Faces `tree` component is populated by a different data collection. In JDeveloper, you define a databound tree using the Edit Tree Binding dialog, which enables you to define the rules for populating each node level in the tree. There must be one rule for each node level in the hierarchy. Each rule defines the following node-level properties:

- The data collection that populates that node level
- The attributes from the data collection that are displayed at that node level

To enable creation of the Customers tree in the Summit ADF sample application as shown in Figure 34-9, a view object, `CountryVO` was first created to return a list of countries. Another view object, `CustomerVO` was then created to return customers. And a view link was created from `CountryVO` to `CustomerVO` in order to establish a master-detail relationship.

In this case, it is also possible to add a third-level and fourth-level nodes, since `CustomerVO` has a view link to the view object `OrdVO`, and `OrdVO` has a view link to `ItemVO`.

In the case where a branch of the tree is recursive, a single view object accompanied by a self-referential view link must be defined in the data model project. For example, take a collection defined by `EmployeesView`, where the root node of each branch is specified by the `ManagerId` attribute and the child nodes of the same branch are the employees (also known as "direct reports") who are related to the `ManagerId`. The source and destination view object named by the self-referential view link are both defined as `EmployeesView`, but the destination can be renamed to `DirectReports` for clarify. For more information about creating self-referential view links, see How to Create a Recursive Master-Detail Hierarchy for an Entity-Based View Object.

> ✎ **Note:**
>
> When your business services are based on ADF Business Components, you can programmatically access a master view object's **view link accessor** attribute to return a detail object to be displayed as a branch of the current node level. For information about view link accessors, see How to Access the Detail Collection Using the View Link Accessor.

Before you begin:

It may help to understand the options that are available to you when you create a master-detail table and form. For more information, see Using Trees to Display Master-Detail Objects.

You may also find it useful to understand additional functionality that can be used with master-detail tables and trees. For more information, see Additional Functionality for Master-Detail Tables, Forms, and Trees.

You will need to complete these tasks:

- Create the master-detail view objects and view links in the data model project, optimize the performance of row set access for tree components, and add the master view object (and optionally the detail view object) to the application module. For details about setting the optimization flag **Retain Row Set**, see Optimizing View Link Accessor Access to Display Master-Detail Data. For information on creating the master-detail hierarchy and adding it to the application module, see How to Create a Master-Detail Hierarchy Based on View Objects Alone.

  Alternatively, if you are using another type of business service, create a data control for that business service as described in Exposing Business Services with Data Controls in *Developing Applications with Oracle ADF Data Controls*.
- Create a JSF page as described in Creating a Web Page.

To display master-detail objects in a tree:

1. Drag the master object from the Data Controls panel, and drop it onto the page. This should be the master data that will represent the root level of the tree.
2. From the context menu, choose **Tree** > **ADF Tree**.

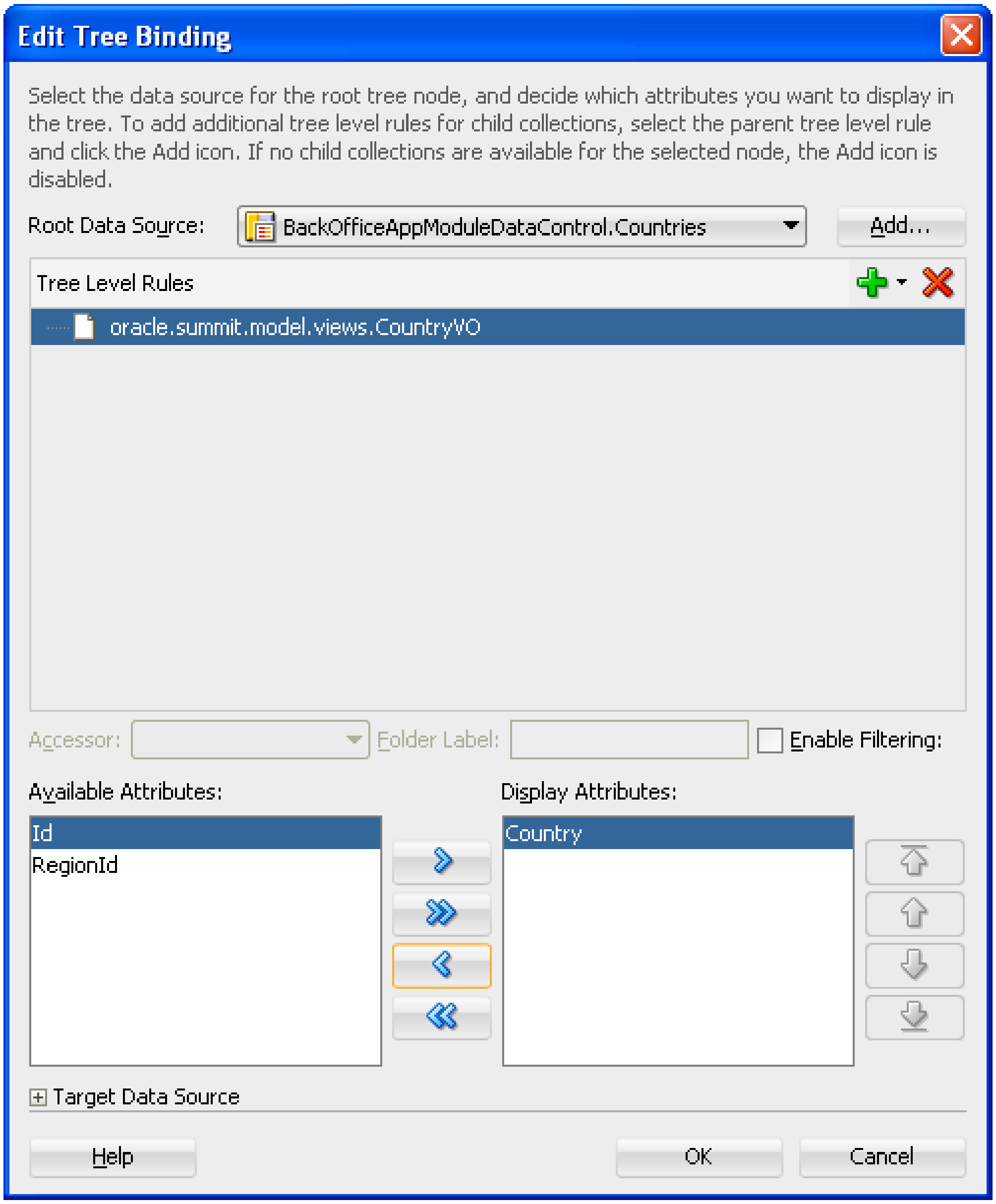
   JDeveloper displays the Edit Tree Binding dialog, as shown in Figure 34-10. You use the binding editor to define a rule for each level that you want to appear in the tree.

   When the dialog first appears, the collection that you dragged is displayed in the **Root Data Source** dropdown list and the corresponding view object is displayed in the **Tree Level Rules** list. This will represent the master data collection.

Tip:

If you don't see the data collection you want in the **Root Data Source** list, click the **Add** button. In the Add Data Source dialog, select a data control and an iterator name to create a new data source. Then click the **Add Rule** icon and select **Add Rule** to add the master rule.

Figure 34-10   Edit Tree Binding Dialog



Edit Tree Binding

Select the data source for the root tree node, and decide which attributes you want to display in the tree. To add additional tree level rules for child collections, select the parent tree level rule and click the Add icon. If no child collections are available for the selected node, the Add icon is disabled.

Root Data Source: BackOfficeAppModuleDataControl.Countries

Tree Level Rules: oracle.summit.model.views.CountryVO

Accessor:

Folder Label:

□ Enable Filtering:

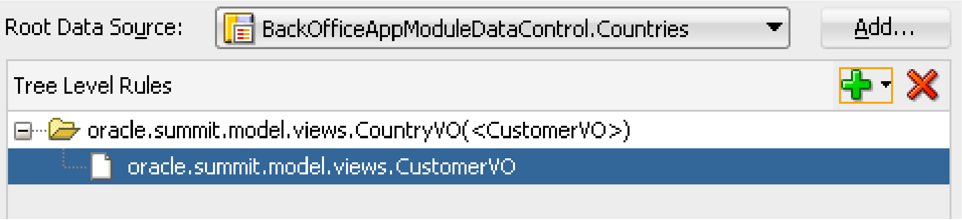
Available Attributes: Id, RegionId

Display Attributes: Country

3. Click the **Add Rule** icon and then select a detail collection to add to the **Tree Level Rules** list.

   A detail data source should appear under the master data source, as shown in Figure 34-11.

Figure 34-11 Master-Detail Tree Level Rules



Root Data Source: BackOfficeAppModuleDataControl.Countries

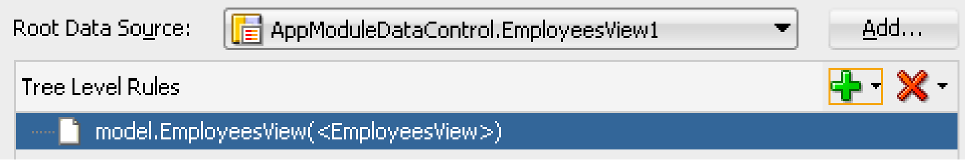
Tree Level Rules

oracle.summit.model.views.CountryVO(<CustomerVO>)
oracle.summit.model.views.CustomerVO

If you are creating a tree with a recursive master-detail hierarchy, then you only need to define a rule that specifies a data source with a self accessor. A recursive tree displays root nodes based on a single collection and displays the child nodes from the attributes of a self accessor that recursively fetches data from that collection. The recursive tree differs from a typical master-detail tree because it requires only a single rule to define the branches of the tree. A recursive data source should display the data source followed by the name of the self accessor in brackets, as shown in Figure 34-12.

Repeat this step for any additional nodes (or further levels of hierarchy) that you want to add to the tree.

Figure 34-12 Recursive Tree Level Rule



Root Data Source: AppModuleDataControl.EmployeesView1

Tree Level Rules

model.EmployeesView(<EmployeesView>)

4. In the **Tree Level Rules** list, select one of the rules.
5. In the bottom part of the dialog, set the attribute or attributes that you want to be used for the node's display by shuttling those attributes to the to the **Display Attributes** list.
6. Repeat steps 4 and 5 for each of the rest of the rules.
7. Click **OK**.

What Happens When You Create an ADF Databound Tree

When you drag and drop from the Data Controls panel, JDeveloper does many things for you. For a general description of what happens and what is created when you use the Data Controls panel, see How to Use the Data Controls Panel.

When you create a databound tree using the Data Controls panel, JDeveloper adds binding objects to the page definition file, and it also adds the `tree` tag to the JSF Page. The resulting UI component is fully functional and does not require any further modification.

Code Generated in the JSF Page

The following example shows the code generated in a JSF page when you use the Data Controls panel to create a tree. This sample tree displays two levels of nodes:

countries and customers. The Countries collection was used to populate the root node.

```
<af:tree value="#{bindings.Countries.treeModel}"
          var="node"
          selectionListener="#{bindings.Countries.treeModel.makeCurrent}"
          rowSelection="single" id="t1"
   <f:facet name="nodeStamp">
      <af:outputText value="#{node}" id="ot1"/>
   </f:facet>
</af:tree>
```

By default, the af:tree tag is created inside a form. The value attribute of the tree tag contains an EL expression that binds the tree component to the Countries tree binding object in the page definition file. The treeModel property in the binding expression refers to an ADF class that defines how the tree hierarchy is displayed, based on the underlying data model. The var attribute provides access to the current node.

In the f:facet tag, the nodeStamp facet is used to display the data for each node. Instead of having a component for each node, the tree repeatedly renders the nodeStamp facet, similar to the way rows are rendered for the ADF Faces table component.

The ADF Faces tree component uses an instance of the oracle.adf.view.faces.model.PathSet class to display expanded nodes. This instance is stored as the treeState attribute on the component. You may use this instance to programmatically control the expanded or collapsed state of an element in the hierarchy. Any element contained by the PathSet instance is deemed expanded. All other elements are collapsed. For more information, see What You May Need to Know About Programmatically Expanding and Collapsing Nodes in *Developing Web User Interfaces with Oracle ADF Faces*.

Binding Objects Defined in the Page Definition File

The following example shows the binding objects defined in the page definition file that pertain to the Countries ADF databound tree in the Summit ADF sample application.

```
<executables>
...
      <iterator Binds="Countries" RangeSize="25"
                DataControl="BackOfficeAppModuleDataControl"
                id="CountriesIterator"/>
</executables>
<bindings>
...
   <tree IterBinding="CountriesIterator" id="Countries">
      <nodeDefinition DefName="oracle.summit.model.views.CountryVO"
                      Name="Countries0">
         <AttrNames>
            <Item Value="Country"/>
         </AttrNames>
         <Accessors>
            <Item Value="CustomerVO"/>
         </Accessors>
      </nodeDefinition>
      <nodeDefinition DefName="oracle.summit.model.views.CustomerVO"
                      Name="Countries1"
                      TargetIterator="${bindings.CustomersIterator}">
         <AttrNames>
            <Item Value="Id"/>
            <Item Value="Name"/>
         </AttrNames>
      </nodeDefinition>
   </tree>
</bindings>
```

The page definition file contains the rule information defined in the Tree Binding Editor. In the `executables` element, notice that although the tree displays two levels of nodes, only one iterator binding object is needed. This iterator iterates over the master collection, which populates the root nodes of the tree. The accessor you specified in the node rules returns the detail data for each branch node.

The `tree` element is the value binding for all the attributes displayed in the tree. The `iterBinding` attribute of the `tree` element references the iterator binding that populates the data in the tree. The `AttrNames` element within the `tree` element defines binding objects for *all* the attributes in the master collection. However, the attributes that you select to appear in the tree are defined in the `AttrNames` elements within the `nodeDefinition` elements.

The `nodeDefinition` elements define the rules for populating the nodes of the tree. There is one `nodeDefinition` element for each node, and each one contains the following attributes and subelements:

- `DefName`: An attribute that contains the fully qualified name of the data collection that will be used to populate the node.
- `id`: An attribute that defines the name of the node.
- `AttrNames`: A subelement that defines the attributes that will be displayed in the node at runtime.
- `Accessors`: A subelement that defines the accessor attribute that returns the next branch of the tree.

The order of the `nodeDefintion` elements within the page definition file defines the order or level of the nodes in the tree, where the first `nodeDefinition` element defines the root node. Each subsequent `nodeDefinition` element defines a subnode of the one before it.

For more information about the elements and attributes of the page definition file, see pageNamePageDef.xml.

What Happens at Runtime: How an ADF Databound Tree Is Displayed

`Tree` components use `org.apache.myfaces.trinidad.model.TreeModel` to access data. This class extends `CollectionModel`, which is used by the ADF Faces `table` component to access data. For more information about the `TreeModel` class, refer to the ADF Faces Javadoc.

When a page with a `tree` is displayed, the iterator binding on the tree populates the root nodes. When a user collapses or expands a node to display or hide its branches, a `DisclosureEvent` event is sent. The `isExpanded` method on this event determines whether the user is expanding or collapsing the node. The `DisclosureEvent` event has an associated listener.

The DisclosureListener attribute on the tree is bound to the accessor attribute specified in the node rule defined in the page definition file. This accessor attribute is invoked in response to the DisclosureEvent event; in other words, whenever a user expands the node the accessor attribute populates the branch nodes.

How to Synchronize Other Parts of a Page With a Tree Selection

In addition to synchronizing a detail view with a master view, you can synchronize other parts of a page with a tree selection. For each node definition (rule), you can specify an optional TargetIterator property and set it to an EL expression for an iterator binding in the current binding container. The expression is evaluated at runtime when the user selects a row in the tree. The iterator binding's view row key attributes match (in order, number, and data type) the view row key of the iterator from which the nodeDefinition type's rows are retrieved for the tree. At runtime, when the tree control receives a selectionChanged event, it passes in the list of keys for each level of the tree. These keys uniquely identify the selected node.

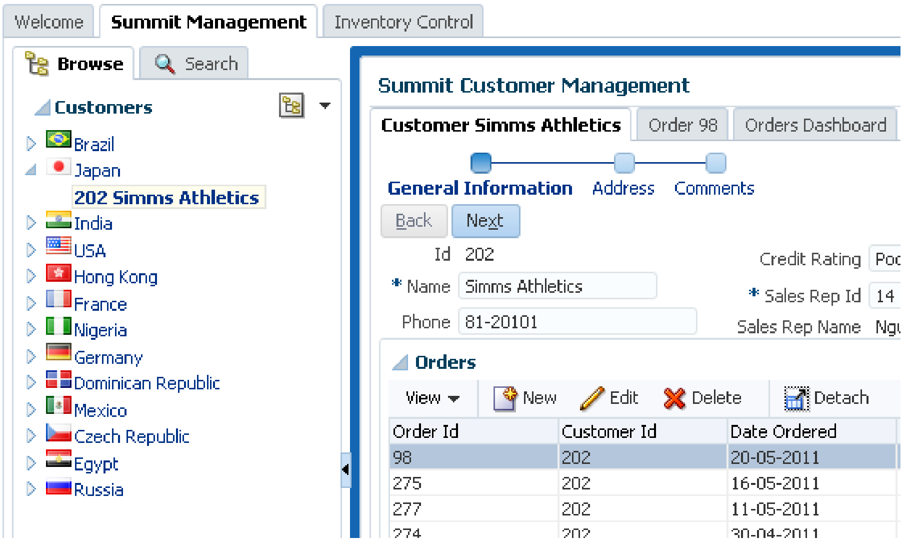
Note:

You can view a page's nodeDefinition elements by expanding a node binding in the page definition editor. These are the same tree binding rules that you can configure in the tree binding dialog.

The tree binding starts at the top of the tree. For each tree level whose key is present in the Currently Selected Tree Node Keys list, if there is a TargetIterator property configured for that nodeDefinition, the tree binding performs a setCurrentRowWithKey() operation on the selected target iterator. It uses the key from the appropriate level of the Currently Selected Tree Node Keys list.

For example, the Summit ADF sample application contains a tree that displays countries at the top node and customers at the second level. When a user selects a customer in the tree, a form displaying the customer information is displayed in the Summit Customer Management panel of the page as shown in Figure 34-13. This functionality consists of a tree that was created from the Countries data control object, a form created from the Customers object, and a detail table created from the OrdersFromCustomer object (which is nested under Customers in the Data Controls panel).

Figure 34-13 Form Synchronized with Tree Selection



Id 202
* Name Simms Athletics
Phone 81-20101
Credit Rating Poo
* Sales Rep Id 14
Sales Rep Name Ngu

Order Id	Customer Id	Date Ordered
98	202	20-05-2011
275	202	16-05-2011
277	202	11-05-2011
274	202	30-04-2011

Before you begin:

It may help to understand the options that are available to you when you create a tree. For more information, see Using Trees to Display Master-Detail Objects.

You may also find it useful to understand additional functionality that can be used with master-detail tables and trees. For more information, see Additional Functionality for Master-Detail Tables, Forms, and Trees.

You will need to complete these tasks:

- Create the master-detail view objects and view links in the data model project, optimize the performance of row set access for tree components, and add the master view object (and optionally the detail view object) to the application module. For details about setting the optimization flag **Retain Row Set**, see Optimizing View Link Accessor Access to Display Master-Detail Data. For information on creating the master-detail hierarchy and adding it to the application module, see How to Create a Master-Detail Hierarchy Based on View Objects Alone.

  Alternatively, if you are using another type of business service, create a data control for that business service as described in Exposing Business Services with Data Controls in *Developing Applications with Oracle ADF Data Controls*.
- Create a JSF page as described in Creating a Web Page.

To synchronize another part of the page with the tree selection:

1. From the Data Controls panel, drag the detail collection from which a record will be displayed based on the user selection in a tree, and drop it as a form or another databound component.
2. From the Data Controls panel, drag the parent collection of the detail collection that you just dropped and choose **Create** > **Tree** > **ADF Tree** (or **ADF Tree Table**).
3. In the Edit Tree Binding dialog, configure tree binding rules for all of the levels of the tree hierarchy as described in How to Display Master-Detail Objects in Trees, and leave the dialog open.

4. Select the detail tree-level rule that corresponds to the same collection that is used in step 1.
5. Expand the Target Data Source node and click the EL Picker button.
6. In the Variables dialog, expand **ADF Bindings**, expand **bindings**, and select the iterator binding for the component that you created in step 1.

   The expression will display in the **Expression** text area. The expression should look something like `${bindings.CustomersIterator}`. This expression will then be used to set the target iterator when a user clicks a detail node.
7. Click **OK** to exit the Variables dialog, and click **OK** to exit the Edit Tree Binding dialog.

Using Tree Tables to Display Master-Detail Objects

The ADF Faces treeTable component is used to display master-detail data relationships in a hierarchical format. This component displays a hierarchy in a UI similar to a table, and is more elaborate than the tree component.

Use the ADF Faces `treeTable` component to display a hierarchy of master-detail collections in a table. The advantage of using a `treeTable` component rather than a `tree` component is that the `treeTable` component provides a mechanism that enables users to focus the view on a particular node in the tree.

For example, you can create a tree table that displays levels of nodes for countries and customers where each root node represents an individual country. The branches off the root nodes display the customers for that country. As with trees, to create such a tree table, it is necessary to have master-detail relationships between the collections in the hierarchy.

A databound ADF Faces `treeTable` displays one root node at a time, but provides navigation for scrolling through the different root nodes. Each root node can display any number of branch nodes. Every node is displayed in a separate row of the table, and each row provides a focusing mechanism in the leftmost column.

You can edit the following `treeTable` component properties in the Properties window:

- Range navigation: The user can click the **Previous** and **Next** navigation buttons to scroll through the root nodes.
- List navigation: The list navigation, which is located between the **Previous** and **Next** buttons, enables the user to navigate to a specific root node in the data collection using a selection list.
- Node expanding and collapsing mechanism: The user can open or close each node individually or use the **Expand All** or **Collapse All** command links. By default, the icon for opening and closing the individual nodes is an arrowhead with a plus or minus sign. You can also use a custom icon of your choosing.
- Focusing mechanism: When the user clicks on the focusing icon (which is displayed in the leftmost column) next to a node, the page is redisplayed showing only that node and its branches. A navigation link is provided to enable the user to return to the parent node.

How to Display Master-Detail Objects in Tree Tables

The steps for creating an ADF Faces databound tree table are exactly the same as those for creating an ADF Faces databound tree, except that you drop the data collection as an **ADF Tree Table** instead of an **ADF Tree**. For more information, see How to Display Master-Detail Objects in Trees.

What Happens When You Create a Databound Tree Table

When you drag and drop from the Data Controls panel, JDeveloper does many things for you. For a full description of what happens and what is created when you use the Data Controls panel, see How to Use the Data Controls Panel.

When you create a databound tree table using the Data Controls panel, JDeveloper adds binding objects to the page definition file, and it also adds the `treeTable` tag to the JSF Page. The resulting UI component is fully functional and does not require any further modification.

Code Generated in the JSF Page

The following example shows the code generated in a JSF page when you use the Data Controls panel to create a tree table. This sample tree table displays two levels of nodes: countries and customers.

By default, the `treeTable` tag is created inside a form. The `value` attribute of the `treeTable` tag contains an EL expression that binds the `tree` component to the binding object that will populate it with data. The `treeModel` property refers to an ADF class that defines how the tree hierarchy is displayed, based on the underlying data model. The `var` attribute provides access to the current node.

```
<af:treeTable value="#{bindings.Countries.treeModel}" var="node"
              selectionListener="#{bindings.Countries.treeModel.makeCurrent}"
              rowSelection="single"
              id="tt1">
    <f:facet name="nodeStamp">
        <af:column id="c1">
            <af:outputText value="#{node}" id="ot1"/>
        </af:column>
    </f:facet>
    <f:facet name="pathStamp">
        <af:outputText value="#{node}" id="ot2"/>
    </f:facet>
</af:treeTable>
```

The `nodeStamp` facet is used to display the data for each node. Instead of having a component for each node, the tree repeatedly renders the `nodeStamp` facet, similar to the way rows are rendered for the ADF Faces `table` component. The `pathStamp` facet renders the column and the path links above the table that enable the user to return to the parent node after focusing on a detail node.

Binding Objects Defined in the Page Definition File

The binding objects created in the page definition file for a tree table are exactly the same as those created for a tree. For more information, see Binding Objects Defined in the Page Definition File.

What Happens at Runtime: Events for Tree Tables

Tree components use oracle.adf.view.faces.model.TreeModel to access data. This class extends CollectionModel, which is used by the ADF Faces table component to access data. For more information about the TreeModel class, refer to the ADF Faces Javadoc.

When a page with a tree table is displayed, the iterator binding on the treeTable component populates the root node and listens for a row navigation event (such as the user clicking the **Next** or **Previous** buttons or selecting a row from the range navigator). When the user initiates a row navigation event, the iterator displays the appropriate row.

If the user changes the view focus (by clicking on the component's focus icon), the treeTable component generates a focus event (FocusEvent). The node to which the user wants to change focus is made the current node before the event is delivered. The treeTable component then modifies the focusPath property accordingly. You can bind the FocusListener attribute on the tree to a method on a managed bean. This method will then be invoked in response to the focus event.

When a user collapses or expands a node, a disclosure event (DisclosureEvent) is sent. The isExpanded method on the disclosure event determines whether the user is expanding or collapsing the node. The disclosure event has an associated listener, DisclosureListener. The DisclosureListener attribute on the tree table is bound to the accessor attribute specified in the node rule defined in the page definition file. This accessor attribute is invoked in response to a disclosure event (for example, the user expands a node) and returns the collection that populates that node.

The treeTable component includes **Expand All** and **Collapse All** links. When a user clicks one of these links, the treeTable sends a DisclosureAllEvent event. The isExpandAll method on this event determines whether the user is expanding or collapsing all the nodes. The table then expands or collapses the nodes that are children of the root node currently in focus. In large trees, the expand all command will not expand nodes beyond the immediate children. The ADF Faces treeTable component uses an instance of the oracle.adf.view.faces.model.PathSet class to determine expanded nodes. This instance is stored as the treeState attribute on the component. You can use this instance to programmatically control the expanded or collapsed state of a node in the hierarchy. Any node contained by the PathSet instance is deemed expanded. All other nodes are collapsed. This class also supports operations like addAll() and removeAll().

For more information about the ADF Faces treeTable component, refer to the oracle.adf.view.faces.component.core.data.CoreTreeTable class in the ADF Faces Javadoc.

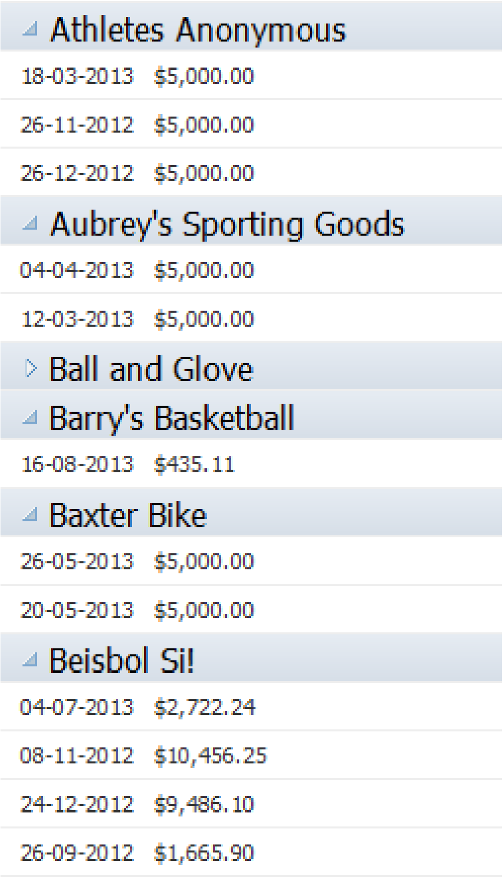
Using List Views to Display Master-Detail Objects

The ADF Faces listView component is used to display a list of items based on rows of a master-detail collection model. The listView component uses a model to access the data in the underlying list.

Use the ADF Faces listView component to display a hierarchy of master-detail collections in a simple tree table. The attributes of each object are displayed in a group or grid within a single row and single column.

For example, as shown in Figure 34-14, you can create a `listView` that displays orders and groups them by customer.

Figure 34-14 Master-Detail with the listView Component



◢ Athletes Anonymous

18-03-2013 $5,000.00

26-11-2012 $5,000.00

26-12-2012 $5,000.00

◢ Aubrey's Sporting Goods

04-04-2013 $5,000.00

12-03-2013 $5,000.00

▷ Ball and Glove

◢ Barry's Basketball

16-08-2013 $435.11

◢ Baxter Bike

26-05-2013 $5,000.00

20-05-2013 $5,000.00

◢ Beisbol Si!

04-07-2013 $2,722.24

08-11-2012 $10,456.25

24-12-2012 $9,486.10

26-09-2012 $1,665.90

For more information on using databound list views, see Creating a List View of a Collection.

How to Display Master-Detail Objects in List Views

You use the Create List View wizard to create your list. This wizard allows you to choose between a `panelGroupLayout` or a `panelGridLayout` component to arrange the components for your list item.

Before you begin:

It may help to understand the options that are available to you when you create a master-detail list view. For more information, see Using List Views to Display Master-Detail Objects.

You may also find it useful to understand additional functionality that can be used with master-detail tables and trees. For more information, see Additional Functionality for Master-Detail Tables, Forms, and Trees.

You will need to complete these tasks:

- Create an application module that contains instances of the view objects and view links that establish the desired master-detail hierarchy, as described in Creating and Modifying an Application Module. Alternatively, if you are using another type of business service, create a data control for that business service as described in Exposing Business Services with Data Controls in *Developing Applications with Oracle ADF Data Controls*.
- Create a JSF page as described in Creating a Web Page.

To create a list view:

1. From the Data Controls panel, locate the detail object, as described in How to Identify Master-Detail Objects on the Data Controls Panel.
2. From the Data Controls panel, drag the detail collection onto the page and choose **Table/List View > ADF List View.**
3. On the first page of the create List View wizard, select a layout component and select **Include header using parent collection**.
4. If you have selected the **Panel Grid Layout** option on the first page the wizard, use the next two pages of the wizard to organize the size and arrangement of the grid cells.

   For more information about using these wizard pages, press the F1 key or click **Help**.
5. On the List Item Data page, determine which fields to display for the detail data by setting the bindings and setting the corresponding components to use for each.
6. In the same page, click the **Group Header Data** tab, and set the bindings and components for the master data.
7. Click **Finish**.
8. Use the Properties window to further configure the layout components.

   For complete information about using the `panelGridLayout` component, see the Arranging Content in a Grid section of the *Developing Web User Interfaces with Oracle ADF Faces*. For more information about the `panelGroupLayout` component, see the Grouping Related Items section of the *Developing Web User Interfaces with Oracle ADF Faces*.

What Happens When You Create a Master-Detail List View

When you use the wizard to create a `listView` component with group headers, JDeveloper creates and configures the `listView`, `listItem`, and layout components. The `listView` contains a `listItem` for the detail items and a `facet` for the group header stamp, which in turn contains a `listItem` for the master items.

The bindings are the same as for a table—the `listView` component is bound to the parent of the collection dropped from the Data Controls panel. The value of the `var` attribute on this component is used by the `outputFormatted` components to access the data, in the same way as a table. For more information about the page definition code and the JSP code for collections, see What Happens When You Create a Table.

The following example shows the page code for the simple table shown in Figure 34-14.

```
<af:form id="f1">
  <af:panelGridLayout id="pgl1">
```

```
    <af:gridRow height="100%" id="gr1">
      <af:gridCell width="100%" halign="stretch" valign="stretch" id="gc1">
        <!-- Content -->
        <af:listView value="#{bindings.Customers.treeModel}" var="item"
                     emptyText="#{bindings.Customers.viewable ? 'No data to
                                  display.' : 'Access Denied.'}"
                     fetchSize="#{bindings.Customers.rangeSize}" id="lv1">
          <af:listItem id="li1">
            <af:panelGroupLayout layout="horizontal" id="pgl2">
              <f:facet name="separator">
                <af:spacer width="10" id="s1"/>
              </f:facet>
              <af:outputFormatted
                    value="#{item.bindings.DateOrdered.inputValue}" id="of1">
                <af:convertDateTime
                    pattern="#{bindings.Customers.hints.DateOrdered.format}"/>
              </af:outputFormatted>
              <af:outputFormatted value="#{item.bindings.Total.inputValue}"
                                  id="of2">
                <af:convertNumber groupingUsed="false"
                           pattern="#{bindings.Customers.hints.Total.format}"/>
              </af:outputFormatted>
            </af:panelGroupLayout>
          </af:listItem>
          <f:facet name="groupHeaderStamp">
            <af:listItem id="li2" inlineStyle="font-size:medium;">
              <af:panelGroupLayout layout="horizontal" id="pgl3">
                <f:facet name="separator">
                  <af:spacer width="10" id="s2"/>
                </f:facet>
                <af:outputFormatted value="#{item.bindings.Name.inputValue}"
                                    id="of3"/>
              </af:panelGroupLayout>
            </af:listItem>
          </f:facet>
        </af:listView>
      </af:gridCell>
    </af:gridRow>
  </af:panelGridLayout>
</af:form>
```

About Selection Events in Trees and Tree Tables

Selection Listener is fired when the server receives a selectionEvent from the client which indicates that whenever you select a node in a tree or row in a table, the server will process the event and executes the Selection Listener defined in a managed bean.

There may be cases when you need to determine which node in a tree or tree table has been selected in order to handle some processing in your application. For example, an application might require processing to determine the products that match a product category node that is selected by the user.

Whenever a user selects a node in a tree (or a row in a table), the component triggers selection events. A `selectionEvent` event reports which rows were just deselected and which rows were just selected. The current selection, that is, the selected row or rows, is managed by the `RowKeySet` object, which keeps track of all currently selected nodes by adding and deleting the associated key for the row into or out of the key set. When a user selects a new node, and the tree or table is configured for single selection, then the previously selected key is discarded and the newly selected key is added. If the tree or table is configured for multiple selection, then the newly selected keys are added to the set, and the previously selected keys may or may not be discarded, based on how the nodes were selected. For example, if the user pressed the `CTRL` key, then the newly selected nodes would be added to the current set.

For more information on selection events, see What You May Need to Know About Performing an Action on Selected Rows in Tables in *Developing Web User Interfaces with Oracle ADF Faces*.

> **Note:**
>
> In cases where you merely need to synchronize the row currency between the selected node and other UI components on the page, you can do so declaratively using the `TargetIterator` property for each node definition. For more information, see How to Synchronize Other Parts of a Page With a Tree Selection.

Selection changes in a tree or table will also trigger contextual events for communicating between regions within a page. See Using Contextual Events.

35

Creating Databound Selection Lists and Shuttles

This chapter describes how to create lists and shuttle components from data modeled with ADF Business Components, using ADF data binding and ADF Faces components. It describes how to create the List of Value (LOV) components that utilize a query to populate the selection list. It includes instructions for creating standard selection components that use a model-driven, fixed-value, or dynamically generated list. It describes how to add navigation list bindings to let users navigate through a list of objects in a collection. It also describes how to use the shuttle component to allow the user to quickly move items between two lists.
This chapter includes the following sections:

About Selection Lists and Shuttles

ADF Faces provides selection list and shuttle components that can display a list of objects from which a user can select a value or select multiple values. Use the List of Values (LOV) component, navigation list bindings, or the `selectManyShuttle` component to create the list with multiple items.

Selection list components present the user with a list of choices as input values. The selection list may be model-driven, obtained from a fixed list, or dynamically created at runtime. Selection list components can be rendered as lists, radio buttons, checkboxes, and list boxes. Some selection types can also be singular (select one) or multiple (select many).

Selection lists and shuttles work the same way as do standard JSF list components. ADF Faces list components, however, provide extra functionality such as support for label and message display, automatic form submission, and partial page rendering. When you present users with a list, they can readily see the available choices and make a selection. Picking from a list also eliminates typing errors that may occur.

List of values (LOV) components are UI components that allow the user to enter values by picking from a list that is generated by a query. The LOV displays inside a modal popup dialog that typically includes search capabilities. The `af:inputListOfValues` and `af:inputComboboxListOfValues` and `af:inputSearch` components, for example, offer additional features that are not available in selection lists, such as search fields inside the LOV modal dialog and queries based on multiple table columns. See How to Create a Popup List of Values and *Developing Web User Interfaces with Oracle ADF Faces*.

List of values components offer more complex search and input capabilities by using the query component to create a popup search panel with a results table. The query component also has features such as auto-suggestion, smart list filtering, and custom facets for additional functionality.

When the user selects an item from a navigation list, a corresponding component bound to the list also changes its value in response to the selection. For example, when the user selects a product from a shopping list, the table that is bound to the products list updates to display the details of the selected product.

A shuttle allows the user to easily see the available items on an available list and the selected items in the selected list and to quickly move those items back and forth between the lists.

Shuttles provide a visual way to select items from an available list and at the same time see those selected items. ADF Faces provides the `selectManyShuttle` component and the `selectOrderShuttle` component, which allow reordering of the selected list. Both components can be implemented by adding code in a managed bean to populate the lists.

Selection Lists and Shuttles Use Cases and Examples

You can use a selection list component such as `selectOneChoice` for a single value input selection situation in which the list is relatively small. For example, you can use a `selectOneChoice` to select from a list of product colors or the type of credit card being used.

For larger lists and with more complex filtering, use an `af:inputListOfValues` or `af:inputComboboxListOfValues` component. These components use the query component to perform a transactional search to populate the list. For instance, you can use the `af:inputComboboxListOfValues` to select from a list of countries in an address input page.

Use the selectShuttle components when you want the user to be able to assemble a list of items and be able to select and unselect them. You should use the shuttle components when the number of items is large but not overwhelming to display. The user can review the selection and be able to iteratively select and unselect the items until the user is satisfied with the final selection. For instance, the available list could be the options available for a particular automobile and the user can shuttle the options the user likes to the selected list.

Additional Functionality for Selection Lists and Shuttles

You may find it helpful to understand other ADF features before you configure or use the ADF Model layer. Additionally, you may want to read about what you can do with your model layer configurations. Following are links to other functionality that may be of interest.

- For more information about using LOV components with search forms and the `af:query` component, see List of Values (LOV) Input Fields.
- You can use the LOV in a task flow that uses an isolated data control. For more information about shared and isolated data controls, see Sharing Data Controls Between Task Flows.
- LOVs are associated with a named **view criteria**. For more information about setting view criteria options, see How to Create Named View Criteria Declaratively.

- LOV components uses the af:query component to populate the selection list. For more information about the af:query component, see Creating ADF Databound Search Forms.

Creating List of Values (LOV) Components

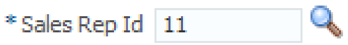
ADF Faces provides List of Values (LOV) components that can display a model-driven list of objects from which a user can select a value or optionally search for the needed item. Use the af:inputListOfValues, af:inputSearchor the af:inputComboboxListOfValues component to create an LOV.

LOV components are input components that allow the user to enter values by picking from a list that is generated by a query. ADF Faces provides the af:inputListOfValues, af:inputSearch and af:inputComboboxListOfValues components. If you are creating a fully featured LOV component, you must define the LOV in the **view object** as described in Working with List of Values (LOV) in View Object Attributes.

If you are using dependent LOVs as part of your search form, you must use them with the af:query component. For more information about using LOV components with search forms, see List of Values (LOV) Input Fields.

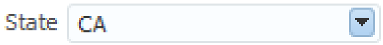
The af:inputListOfValues component has a search icon next to the search criteria field, as shown in Figure 35-1.

Figure 35-1 Search Criteria Input Field Defined as an inputListOfValues



The af:inputComboboxListOfValues component has a dropdown icon next to the field, as shown in Figure 35-2 for the State attribute.

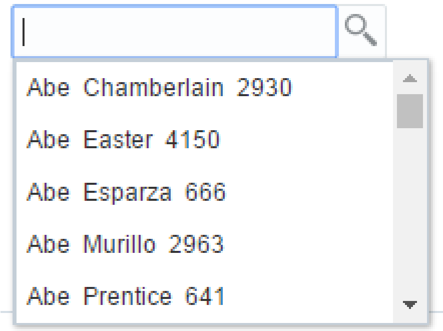
Figure 35-2 Search Criteria Input Field Defined as an inputComboboxListOfValues



The af:inputSearch component filters and highlights matching suggestions. You use this LOV component with an ADF REST resource. Filtering of suggestions is case insensitive. The difference between af:inputSearch and af:inputComboboxListOfValues is that the latter component fetches the list from the server every time you launch the dropdown. However, af:inputComboboxListOfValues has an attribute named contentDelivery which if set to immediate will not fetch the list while launching the dropdown. The list would have to be pre-fetched while loading or rendering the component. In the case ofaf:inputSearch , this component fetches the list from an ADF REST resource based on a RESTful web service. If caching is enabled by way of HTTP headers on the ADF REST service, the resource can also be cached on the client. Caching is controlled by the ADF REST service implementor. In such a scenario, subsequent LOV dropdown launches retrieve the list from the cache. If caching is enabled by way of HTTP headers, it facilitates in-browser filtering based on user input, and also managing MRU (Most Recently Used) suggestions within the browser.

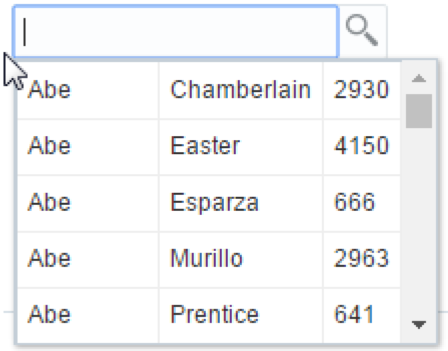
The default display of `af:inputSearch` component's suggestion list is in the list and table modes as shown in Figure 35-3 and Figure 35-4.

Figure 35-3 Search Criteria Input Search Field with List Display Mode



Abe Chamberlain 2930
Abe Easter 4150
Abe Esparza 666
Abe Murillo 2963
Abe Prentice 641

Figure 35-4 Search Criteria Input Search Field with Table Display Mode

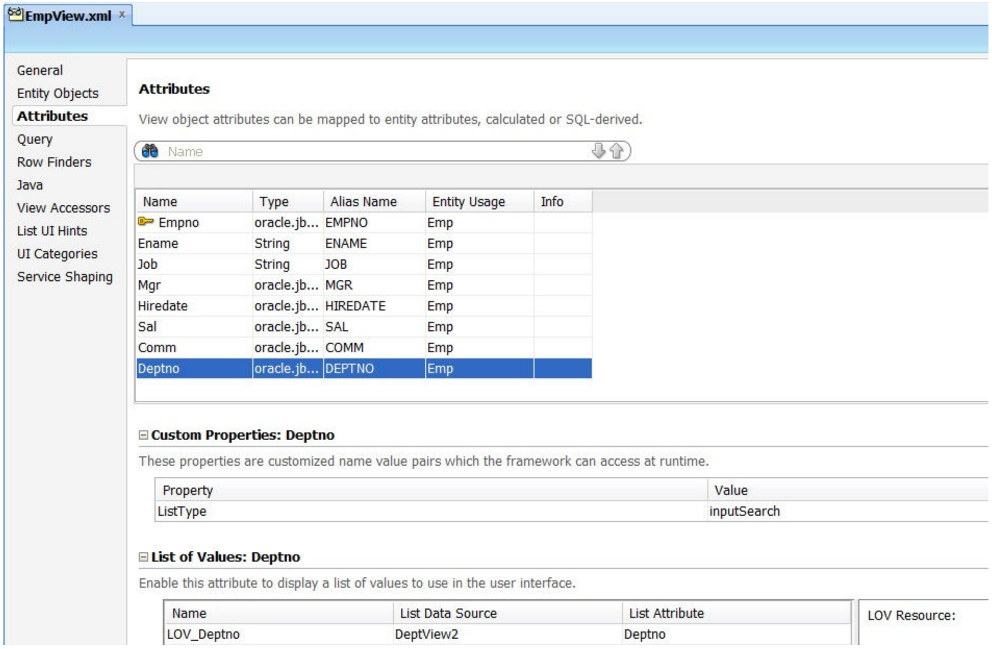


Abe	Chamberlain	2930
Abe	Easter	4150
Abe	Esparza	666
Abe	Murillo	2963
Abe	Prentice	641

You can customize the display of this component's suggestion list using the `contentStamp` and `contentHeader` facets. The code populates within the `af:inputSearch` tag. The code has the `ValueAttribute` attribute, which is the LOV attribute for which the `af:inputSearch` is configured. In this example, the LOV attribute is `Deptno` and hence the `ValueAttribute` attribute is assigned the value `deptnoid`. The `contentMode` attribute is assigned the value of table, which indicates that the suggestion list rendered by this component for this LOV attribute will display in tabular format. Further, `f:facet` tags encloses the code for the `contentStamp` facet (as indicated by `name = contentStamp`). In this example, the LOV attribute is `Deptno`, and this attribute retrieves data from the Dept (Department) table. As indicated in the `f:facet` tag, Dname (Department Name) and Deptno (Department Number) are the columns configured to appear in the suggestion list when the user enters input in the LOV component.

You can configure the columns that appear in the suggestion list. To illustrate this with an example, see Figure 35-5 of `EmpView.xml`; `Deptno` attribute selected in the Attributes tab, and LOV_Deptno entry is selected in the **List of Values: Deptno** section. **Deptno** is already configured as an LOV attribute of type `af:inputSearch`. Double-click on LOV_Deptno entry in the **List of Values: Deptno** section to open the **Edit List of Values** dialog. Figure 35-6 displays this dialog – Edit List of Values. Select **UI Hints** tab and move the columns from the **Available** section to **Selected** section. The entries in the **Selected** section appear in the suggestion list when a user enters input into the LOV component.

Figure 35-5 LOV attribute configured as `af:inputSearch`



Attributes

View object attributes can be mapped to entity attributes, calculated or SQL-derived.

Name	Type	Alias Name	Entity Usage	Info
Empno	oracle.jb...	EMPNO	Emp	
Ename	String	ENAME	Emp	
Job	String	JOB	Emp	
Mgr	oracle.jb...	MGR	Emp	
Hiredate	oracle.jb...	HIREDATE	Emp	
Sal	oracle.jb...	SAL	Emp	
Comm	oracle.jb...	COMM	Emp	
Deptno	oracle.jb...	DEPTNO	Emp	

Custom Properties: Deptno

These properties are customized name value pairs which the framework can access at runtime.

Property	Value
ListType	inputSearch

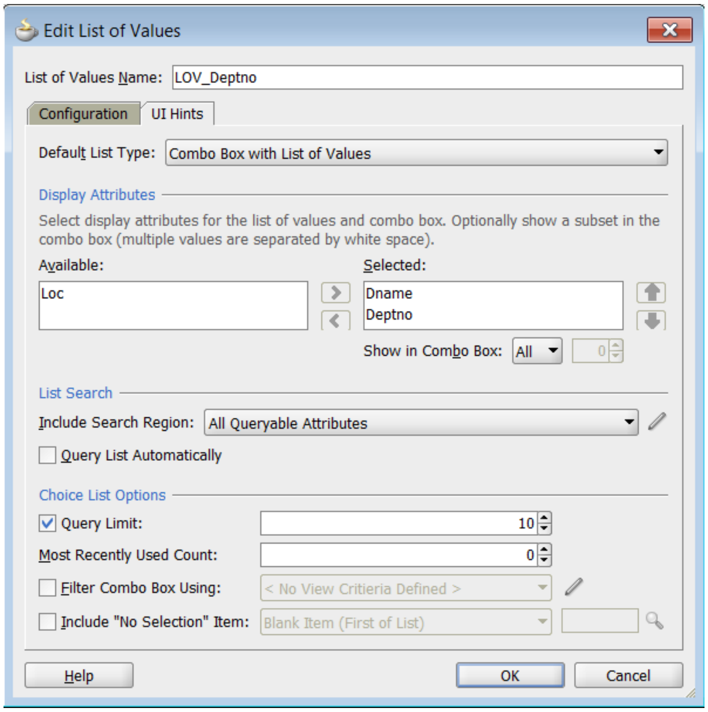
List of Values: Deptno

Enable this attribute to display a list of values to use in the user interface.

Name	List Data Source	List Attribute
LOV_Deptno	DeptView2	Deptno

LOV Resource:

Figure 35-6 Edit List of Value Dialog



Edit List of Values

List of Values Name: LOV_Deptno

Default List Type: Combo Box with List of Values

Display Attributes

Select display attributes for the list of values and combo box. Optionally show a subset in the combo box (multiple values are separated by white space).

Available: Loc

Selected: Dname, Deptno

Show in Combo Box: All 0

List Search

Include Search Region: All Queryable Attributes

□ Query List Automatically

Choice List Options

☑ Query Limit: 10

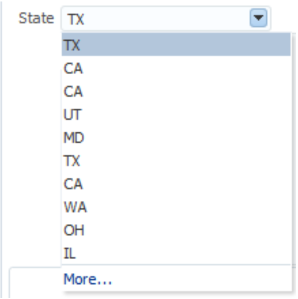
Most Recently Used Count: 0

□ Filter Combo Box Using: < No View Critieria Defined >

□ Include "No Selection" Item: Blank Item (First of List)

For `af:inputComboboxListOfValues`, clicking the dropdown icon displays the LOV dropdown list and either a **More** or a **Search** link, as shown in Figure 35-7. A **Search** link appears when the LOV Search and Select popup dialog contains a search panel to refine the search. A **More** link appears when the popup dialog contains only the list-of-values table or when a **smart list** is used.

Figure 35-7 LOV Dropdown List with Search Link for inputComboboxListOfValues



The user can select any of the items in the dropdown list to populate the input field. The dropdown list includes a filtered list of values and an optional list of most recently used items. The width of each attribute in the dropdown list can be configured using the **DisplayWidth** UI hint for that attribute in the view object definition. The width of the dropdown list will be the sum of the widths of all the attributes in the dropdown list.

The user can also enter the value into the field. If the user enters data that is of a different data type than what is defined for the `inputComboboxListOfValues`, a conversion error will be thrown. For example, if the `inputComboboxListOfValues` is of `NUMBER` datatype and the user enters an alphanumeric string, a conversion error will result.

You can create custom content to be rendered in the Search and Select dialog using the `searchContent` facet. You can add components such as input text, `tables`, and `trees` to display the custom content. You will need to define the `returnPopupDataValue` attribute and implement a `returnPopupListener`. When you want to add the auto-complete feature with your custom popup, you can disable the popup by adding `launchPopupEvent.setLaunchPopup(false)` to your `LaunchPopupListener()` code. However, clicking on the **Search** link will still launch the Search and Select dialog. For information about adding custom content, see Using List-of-Values Components.

The `resultsTable` facet is used to display a custom message when the results table of the Search and Select dialog is empty.

If the component's `readOnly` attribute is set to true, then the input field will not be rendered and the value cannot be changed. By default, the `readOnly` attribute is set to `false`, which also enables the `editMode` attribute to determine whether the user is permitted only to select a value from the list (`editMode` set to `select`) or whether the user can also enter a value into the input field (`editMode` set to `input`).

You can also set up the LOV component to display a list of selectable suggested items when the user types in a partial value. For example, when the user types in `CA`, then a suggested list which partially matches `CA` is displayed as a suggested items list. The user can select an item from the list to be entered into the input field. If there are no matches, a "No results found" message will be displayed. You add the `af:autoSuggestBehavior` tag from the Components window into the LOV and set the `selectedItem` attribute to the `suggestedItems` method implemented in ADF Model.

Note that auto suggest behavior occurs in the Apply Request Values phase. If the `immediate` property is set to `true`, validation will be performed in the same phase. If there is a validation error, the suggestion list will not be displayed. If you want to add auto suggest behavior, do not set `immediate` to `true`.

If the LOV attribute is not unique and has multiple entries with the same value, auto-suggest will only display one value in the list. To display all duplicate values in the auto-suggest selection list, you need to select an attribute that has a unique foreign key to be the LOV attribute. You can select a more readable attribute to be the display attribute in the LOV definition by adding the `f:converter binding` tag inside the `inputListOfValues` or `inputComboboxListOfValues` component. The display attribute values will be shown in the input field and in the auto-suggest list.

```
f:converter binding = "#{bindings.LOV_ATTR.converter}"
```

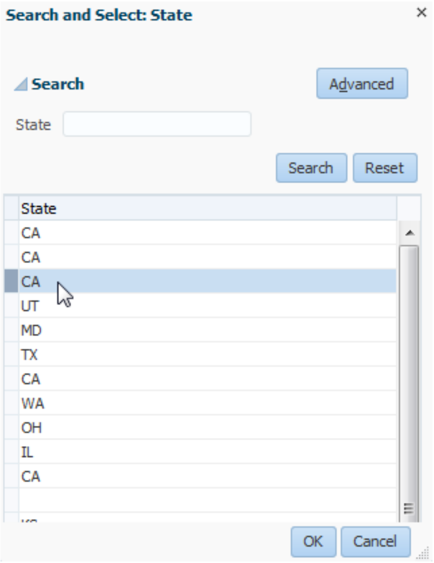
For instance, if the Research department has multiple entries with different department numbers and you defined the `DeptName` attribute as a LOV, the auto-suggest list will only display one department entry for Research. Instead, you can define `Deptno` as the LOV attribute because `Deptno` is a unique foreign key. Auto-suggest will then list all entries for the Research department. However, to make the LOV display more readable, define `DeptName` as the display attribute and use the `f:converter binding` tag to display `DeptName` instead of `Deptno` in the rendered selection field, as shown in the following example.

```
<af:inputComboboxListOfValues id="deptnoId"
                              popupTitle="Search and Select:
                              #{bindings.Deptno.hints.label}"
                              value="#{bindings.Deptno.inputValue}"
                              label="#{bindings.Deptno.hints.label}"
                              model="#{bindings.Deptno.listOfValuesModel}"
                              required="#{bindings.Deptno.hints.mandatory}"
                              columns="#{bindings.Deptno.hints.displayWidth}"
                              shortDesc="#{bindings.Deptno.hints.tooltip}">
  <f:validator binding="#{bindings.Deptno.validator}"/>
  <f:converter binding="#{bindings.Deptno.converter}"/>
  <af:autoSuggestBehavior suggestItems="#{bindings.Deptno.suggestItems}"/>
</af:inputComboboxListOfValues>
```

The LOV will display the department name to the user, but will use the unique department number when interacting with the model.

When the user clicks the **Search** or **More** link (or for `af:inputListOfValues`, the search icon), the LOV Search and Select dialog appears with the full list of values in a table format. The LOV Search and Select dialog launched from a **More** link is shown in Figure 35-8.

Figure 35-8   LOV Search and Select Dialog



State
CA
CA
CA
UT
MD
TX
CA
WA
OH
IL
CA

If a user enters a partial string in the LOV and click **Search** or tab out, a Search and Select dialog will appear with the partial string displayed in the search field and the resulting list of matching values in a table. The user can then select an entry from the list to be the LOV value.

The Search and Select popup dialog also presents a create function that allows the user to add a new row. Be aware that if you are creating a new record using the LOV Search and Select dialog, the new row will appear as an empty row in the table if the LOV has the **Query Automatically** control hint set to `false`. The content of the row will appear when you perform a query using the **Search** button. The following example shows a code sample that uses a link to invoke the create function.

```
<f:facet name="customActions">
      <af:link id="createLink" text="Create..."
         partialSubmit="true">
                 <af:showPopupBehavior popupId="createSLPopup"
                 alignId="createLink"/>
      </af:link>
</f:facet>
```

If the LOV is part of a task flow that is using an isolated data control, and you use the create function to add a new record, the newly added record will not be displayed in the parent page. This is because the search region is also using an isolated data control scope, so the underlying view object updates are not displayed. For information about shared and isolated data controls, see Sharing Data Controls Between Task Flows.

To programmatically refresh the LOV and display the view object updates, add a `returnListener` to the **Create** link with code similar to that shown in the following example.

```
public void refreshLOV() {
     BindingContainer bindings = this.getBindings();
     oracle.jbo.uicli.binding.JUCtrlListBinding lovBinding =
         (oracle.jbo.uicli.binding.JUCtrlListBinding)
```

```
          bindings.get("Description1");
     JUIteratorBinding lovIter = lovBinding.getIteratorBinding();
     RowSet rs = lovIter.getRowSetIterator().getRowSet();
     rs.executeQuery();

    //Add LOV as the partialTrigger


AdfFacesContext.getCurrentInstance().addPartialTarget(this.getPlatformDesc());
```

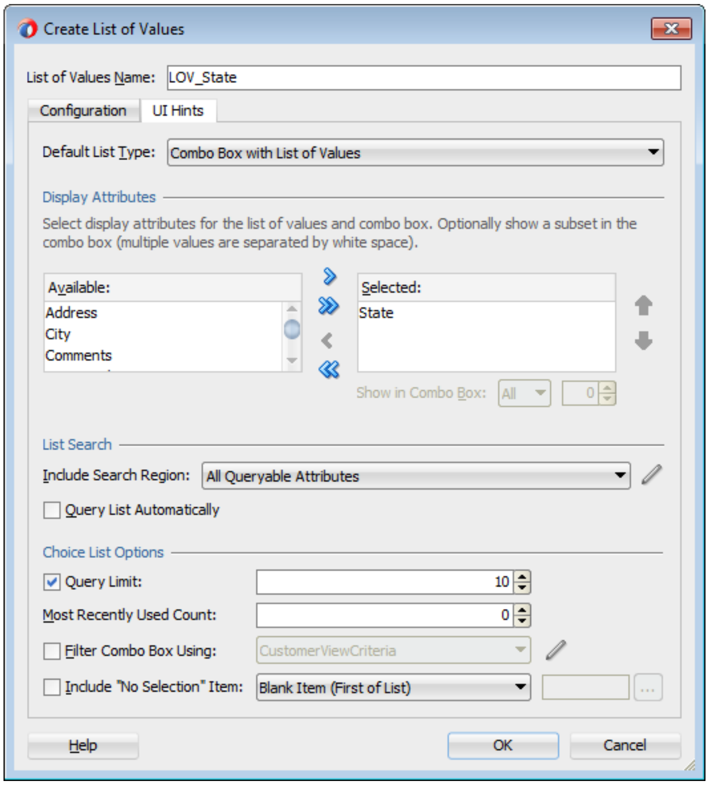
An LOV is associated with a data source via the view accessor. You can apply one or more view criteria to the view accessor associated with the LOV. The **view accessor** provides permanent filtering to the LOV data source. In addition to this permanent filtering, you may be able to apply other filters.

The LOV dialog may include a query panel to allow the user to enter criteria to search for the list of values, or it may contain only a table of results. When you define the LOV in the view object, you can use the UI hints to specify whether a search region should be displayed, which view criteria should be used as the search to populate the list of values, and which LOV component to use. Figure 35-9 shows the Create List of Values dialog and some of its options. In this example, the search region will be made available with the `All Queryable Attributes` named view criteria used for the query, and `af:inputComboboxListOfValues` as the component. Another useful option for the `af:inputComboboxListOfValues` is the **Show in Combo Box** option, which allows you to select the number of attributes to display in the dropdown list and in the Search and Select dialog. For information on LOV UI hints, see How to Set User Interface Hints on a View Object LOV-Enabled Attribute.

For both `af:inputListOfValues` and `af:inputComboboxListOfValues`, if the user enters a partial match in the input search field and presses the Tab or Enter key, the LOV automatically launches the LOV Search and Select dialog and executes the query after applying an auto-generated view criteria with a single item representing the partial match value entered by the user. If there are matches, the Search and Select dialog displays all the entries matching the partially entered criteria. If there are no entries that match the entered partial match, then the dialog displays all the entries.

By default, in the `af:inputComboboxListOfValues` component **auto-complete** feature, the search is case-sensitive. If you want the search to be case-insensitive, create a view criteria to be associated with that LOV attribute. In the Create View Criteria dialog, deselect the **Ignore Case** checkbox for that view criteria item before you apply that view criteria to the LOV definition. For information about setting view criteria options, see How to Create Named View Criteria Declaratively.

Figure 35-9   List of Values Dialog UI Hints Tab



Create List of Values

List of Values Name: LOV_State

Configuration | UI Hints

Default List Type: Combo Box with List of Values

Display Attributes

Select display attributes for the list of values and combo box. Optionally show a subset in the combo box (multiple values are separated by white space).

Available: Address, City, Comments

Selected: State

Show in Combo Box: All 0

List Search

Include Search Region: All Queryable Attributes

☐ Query List Automatically

Choice List Options

☑ Query Limit: 10

Most Recently Used Count: 0

☐ Filter Combo Box Using: CustomerViewCriteria

☐ Include "No Selection" Item: Blank Item (First of List)

You can also set up the LOV component to display a list of selectable suggested items when the user types in a partial value. For example, when the user types in `Ca`, then a suggested list which partially matches `CA` is displayed as a suggested items list. The user can select an item from the list to be entered into the input field. You add this **auto-suggest behavior** feature by including an `af:autoSuggestBehavior` tag from the Components window in the LOV and set the `selectedItem` attribute to the `suggestedItems` method implemented in ADF Model.

If the LOV is an `af:inputComboboxListOfValues`, you can apply an additional view criteria to further filter the values in the dropdown list to create a smart list. If the **Filter Combo Box Using** UI hint is not applied, the dropdown list is filtered by the view criteria applied to the view accessor. Note that this smart list filter applies only to the dropdown list. The full list is still used for validation and for searching from the LOV Search and Select popup dialog. When smart list is enabled, the LOV's Search and Select dialog will display a **More** link instead of the **Search** link.

If both the auto-suggest behavior and smart list filter are enabled for an LOV, auto-suggest will search from the smart list first. If the user waits for two seconds without clicking, then auto-suggest will also search from the full list and append the results. You can also specify the number of suggested items returned by setting the `maxSuggestedItems` attribute (-1 indicates a complete list). If `maxSuggestedItems > 0`, a **More** link is rendered for the user to click on to launch the LOV's Search and Select dialog. The following example shows the code from an LOV with both auto-suggest behavior and a smart list.

```
af:autoSuggestBehavior
    suggestItems="#{bindings.CountryName.suggestItems}"
```

```
smartList="#{bindings.CountryName.smartList}"/>
maxSuggestedItems="5"
```

You can define an attribute in the view object to be one or more LOVs. If multiple LOVs are defined on an attribute, each LOV has its own name, view accessor, data mappings, validator, and UI hints (except for the **Default List Type** hint, which is defined once for the attribute). To switch between LOVs to be used at runtime, an LOV switcher is used. The LOV switcher can be based on an existing attribute of type `String`, or created as a new `String` attribute solely for switching between LOVs. The LOV switcher returns the name of the LOV to use at runtime. For instance, you can create three LOVs for the price attribute, and designate the `CountryName` attribute as the LOV switcher. At runtime, the value of `CountryName` will switch the price attribute to use the LOV that reflects the country's price and currency.

If an attribute is defined as an LOV, you can set the **Support Multiple Value Selection** control hint in its view criteria to enable users to make multiple selections in the search criteria field. If multiple selection is enabled on an LOV attribute, and the `Equal to` or `Not equal to` operator is chosen, a `selectManyChoice` component will render in the query panel. The user can select multiple items as the search criteria.

If the value of an LOV depends on the value of another LOV, then the two LOVs are called **cascading** LOVs. For instance, the list of values for the City LOV depends on the value of the CountryName LOV that was selected. If the LOV is defined as a named bind variable, or if validation is enabled, the LOV query may behave differently depending on such conditions as whether null values for the named bind variable are acceptable. If the bind variable is null, you may want the LOV query to return an empty list. The view criteria's **Ignore Null Values** and **Validation** options can be used to define LOV query behavior.

If the view object has an LOV with a bind variable, you should set that view object bind variable's control hint to **Hide**. Otherwise, the bind variable will be displayed in the LOV Search and Select popup dialog. In the view object overview editor's **Query** tab, double-click the bind variable, click the **Control Hints** tab and select **Hide** from the **Display Hint** dropdown list.

> **Note:**
>
> List of values are designed to return valid values only. As a result, validation rules defined on data source view object attributes will be suppressed. You should ensure that the list of values returned only contains valid values. See What You May Need to Know About List of Values and Attribute Validation Rules.

> **Best Practice:**
>
> An attribute that has been enabled as an LOV should not be used as search criteria in a manually created search form using the "Find Mode" iterator. Creating a search form in this way could result in SQL exception: `java.sql.SQLException: Attempt to set a parameter name that does not occur in the SQL`. It is best practice to use the query and quick query components for creating search forms. For more information, see Creating ADF Databound Search Forms.

Displaying an LOV list for a name attribute when the view object's key attribute is a numerical value requires mapping supplemental attributes on the LOV-enabled attribute. For example, to display a list of credit cards by name instead of by their payment option ID value, you can create an LOV on the name reference attribute and define the primary key attribute as a supplemental attribute to the LOV. For details about creating an LOV-enabled reference attribute, see How to Define an LOV to Display a Reference Attribute.

> **Note:**
>
> `af:inputListOfValues` and `af:inputComboboxListOfValues` assume the list of values returned from the Search and Select dialog to be valid. Any ADF Business Components validation errors will not be displayed.

For information about different usage scenarios and other view criteria options, see How to Create Named View Criteria Declaratively, and What You May Need to Know About Bind Variables in View Criteria.

How to Create an LOV

You can create the LOV using either the `af:inputListOfValues` or the `af:inputComboboxListOfValues` component. You can add auto-suggest behavior to the component to display a list of possible matches from user input.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create an LOV. For more information, see Creating List of Values (LOV) Components.

You may also find it helpful to understand functionality that can be used with LOVs. For more information, see Additional Functionality for Selection Lists and Shuttles.

You will need to complete this task:

> Define the attribute to be an LOV in the view object by following the procedure described in Working with List of Values (LOV) in View Object Attributes.

To create an LOV:

1. In the Application window, double-click the web page that will display the component.

2. From the Data Controls panel, drag and drop the attribute onto the JSF page and choose **Create > List of Values > ADF LOV Input** or **Create > List of Values > ADF LOV Choice List**.
3. In the ADF Faces page of the Components window, from the Operations panel drag and drop **Auto Suggest Behavior** as a child to the LOV component.
4. In the Property Inspector, enter values for **Auto Suggest Behavior**:
   - **SuggestItems**: EL expression that resolves to the `suggestItems` method.
   - **SmartList**: EL expression that resolves to the `smartList` method.
   - **MaxSuggestedItems**: Number of items to be displayed in the auto-suggest list. Enter -1 to display the complete list.
5. If you want to set a different display attribute for the LOV, add the `f:converter binding` tag to the component. For instance, you want the LOV to display the sales rep name instead of the sales rep id, you can add this tag to the component: `f:converter binding="#{bindings.SalesRepId.converter}"` and define `SalesRepName` as the display attribute.
6. If you added the auto-suggest behavior, you must not set the component's `immediate` property to `true`.

   The following example shows an `inputListOfValues` component with the auto-suggest behavior and smart list features.

```
<af:inputListOfValues id="salesRepIdId"
                 popupTitle="Search and Select:
#{bindings.SalesRepId.hints.label}"
                 value="#{bindings.SalesRepId.inputValue}"
                 label="#{bindings.SalesRepId.hints.label}"
                 model="#{bindings.SalesRepId.listOfValuesModel}"
                 required="#{bindings.SalesRepId.hints.mandatory}"
                 columns="#{bindings.SalesRepId.hints.displayWidth}"
                 shortDesc="#{bindings.SalesRepId.hints.tooltip}">
<f:validator binding="#{bindings.SalesRepId.validator}">
<f:converter binding="#{bindings.SalesRepId.converter}">
<af:autoSuggestBehavior suggestItems="#{bindings.SalesRepId.suggestedItems}"
         smartList="#{bindings.SalesRepId.smartList}"
         maxSuggestedItems="5"/>
</af:inputListOfValues>
```

How to Configure an LOV Attribute as inputSearch

You create an `af:inputSearch` LOV component by defining an LOV attribute initially and then configuring the LOV attribute as `inputSearch`.

Before you begin

See How to Define a Single LOV-Enabled View Object Attribute to understand how to define an attribute as LOV attribute. After configuring the LOV attribute as `inputSearch`, you proceed to create an `af:inputSearch` LOV component.

See How to Create an LOV to understand how to create an LOV component. After building the `af:inputSearch` component, you specify the URL of the ADF REST resource so that the `af:inputSearch` LOV component fetches the LOV from this ADF REST resource.

See Creating ADF REST Resources Using the Application Module to understand how to create a RESTful web service with application modules.

See How to Provide af:inputSearch LOV Component the REST URL to understand how to provide the REST URL to the af:inputSearch LOV component.

To configure an LOV attribute as `inputSearch`:

1. In the Applications Navigator, double-click the view object that contains the LOV attribute that you wish to configure as `inputSearch`.
2. In the **Attributes** page, select the LOV attribute and in **Custom Properties** section, click the plus icon and select **Non-translatable Property** to add a row with columns **Property** and **Value**.
3. Double-click the row and enter `ListType` in the **Property** column and `inputSearch` in the **Value** column of this row.
4. Click **Save** to save the project.

What Happens When You Create an inputSearch Component

When you drag and drop an attribute defined as an LOV and also defined as an `inputSearch` from the Data Controls panel onto a `.jspx` page as an `inputSearch` component, JDeveloper adds code to the page similar to that shown in the following example. JDeveloper also does many other things - see What Happens When You Create an LOV to know more.

```
<af:inputSearch id="deptnoID" contentMode="table"
        valueAttribute="#{bindings.Deptno.valueAttribute}"
        displayAttributes="#{bindings.Deptno.displayAttributes}"
        value="#{bindings.Deptno.inputValue}"
        selectionConverter="#{bindings.Deptno.selectionConverter}"
        label="#{bindings.Deptno.hints.label}"
        required="#{bindings.Deptno.hints.mandatory}"
        columns="#{bindings.Deptno.hints.displayWidth}"
        shortDesc="#{bindings.Deptno.hints.tooltip}">
<f:facet name ="contentStamp">
<af:sanitized>
<td>{{Dname}}</td><td>{{Deptno}}</td>
</af:sanitized>
</f:facet>
</af:inputSearch>
```

In the page definition file, JDeveloper adds code as shown in the following example. The code sample shows the `Deptno` attribute designated as `inputSearch` list type.

```
<listOfValues StaticList="false" IterBinding="EmpView1Iterator"
                                Uses="LOV_Deptno" id="Deptno"
ListType="inputSearch"/>
</bindings>
```

How to Provide af:inputSearch LOV Component the REST URL

After configuring an attribute as an LOV attribute, and defining it as of type `inputSearch`, the next step is to create the `af:inputSearch` LOV component and provide this component the REST URL.

You do this so that this component uses the REST URL to fetch the LOV list from the ADF REST resource. See How to Configure an LOV Attribute as inputSearch to understand how to create an LOV attribute and configure it as an LOV attribute that works with `af:inputSearch`. To provide a REST URL to an `inputSearch` LOV component:

1. Ensure that the RESTful web service is running when running the `.jspx` page that contains the `af:inputSearch` component.
2. In the `.jspx` page that contains the `af:inputSearch` LOV component, click the Source tab to view the source code.
3. Locate the `af:searchSection` tag and enter the REST URL as a value to the `dataURL` attribute of this tag. An example REST URL is `dataUrl="//Application3-RESTWebService-context-root/rest/v1/Dept"`. In this REST URL, `Application3-RESTWebService-context-root` is the context root of the RESTful web service. `rest` is the URL pattern of the `RESTServlet` in the `web.xml` file of the RESTful web service. `v1` is the version of the ADF REST resource. `Dept` is the name of the REST resource.
4. Build the `.jspx` page to view LOV component and retrieve the LOV from the ADF REST resource.

What Happens When You Create an LOV

When you drag and drop an attribute from the Data Controls panel, JDeveloper does many things for you. For a full description of what happens and what is created when you use the Data Controls panel, see What Happens When You Create a Text Field.

When you drag and drop an attribute defined as an LOV from the Data Controls panel onto a JSF page as an `inputListOfValues` or `inputComboboxListOfValues` component, JDeveloper adds code to the page similar to that shown in the following example. The `State` attribute in the `CustomerVO` view object has been defined as an LOV and its default component is set to be an `inputComboboxListOfValues`. The component gets its properties from the control hints defined declaratively in the view object. If you want to include the auto-suggest behavior, you must manually add that tag from the Components window.

```
<af:inputComboboxListOfValues id="stateId"
        popupTitle="Search and Select: #{bindings.State.hints.label}"
        value="#{bindings.State.inputValue}"
        label="#{bindings.State.hints.label}"
        model="#{bindings.State.listOfValuesModel}"
        required="#{bindings.State.hints.mandatory}"
        columns="#{bindings.State.hints.displayWidth}"
        shortDesc="#{bindings.State.hints.tooltip}">
        <f:validator binding="#{bindings.State.validator}"/>
</af:inputComboboxListOfValues>
```

In the page definition file, JDeveloper adds code as shown in the following example. The bindings section of the page definition specifies that the LOV is for the `State` attribute and the name of the LOV is `LOV_State`. JDeveloper adds the definitions for the iterator binding objects into the `executables` element, and the list of values binding object into the `bindings` element.

```
<executables>
    <variableIterator id="variables"/>
    <iterator Binds="Customers" RangeSize="25"
              DataControl="BackOfficeAppModuleDataControl"
              id="CustomersIterator"/>
```

```
</executables>
<bindings>
    <listOfValues StaticList="false" IterBinding="CustomersIterator"
                  Uses="LOV_State" id="State"/>
</bindings>
```

For more information about the page definition file and ADF data binding expressions, see Working with Page Definition Files, and Creating ADF Data Binding EL Expressions.

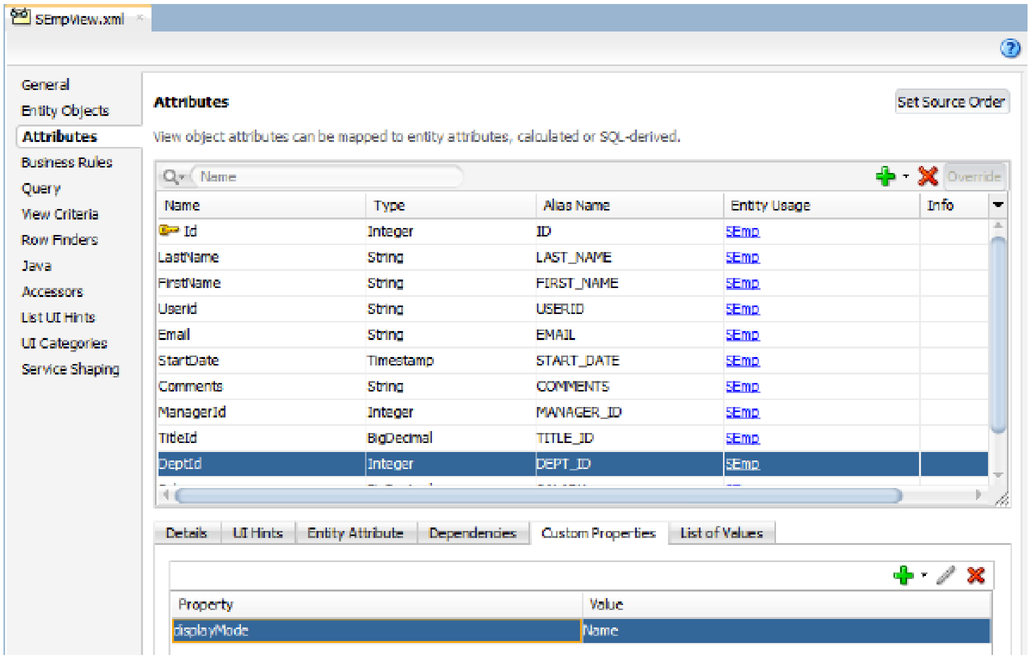
What You May Need to Know About List Validators and LOV

If you are defining an attribute in the view object to be a list of values attribute, you can define an equivalent list validation rule in the underlying entity object as long as the list validation rule is based on the same list. You should not define other types of validation rules in the entity object for an attribute already defined as a list of values in the view object. Doing so may result in unexpected behavior and validation errors. For more information about defining validation rules, see Figure 10-3, and Using the Built-in Declarative Validation Rules.

What You May Need to Know About InputComboboxListOfValues and Null Values

If you create an `inputComboboxListOfValues` component with a view object list data source and that list may be modified at runtime so that list items can be deleted, you must define a custom property `displayMode` on the LOV attribute to support null values. By default, the `inputComboboxListOfValues` component will display the value of the list attribute defined by the LOV attribute when no display value is available. To ensure a deleted display attribute value is not substituted by its corresponding attribute list value, define the custom property `displayMode` with the name of the LOV display attribute as a value. For example, Figure 35-10 shows the LOV attribute `DeptId` that defines the custom property with the LOV display attribute `Name` as a value. Then, at runtime, when a department name is deleted from the list of available departments, the custom property ensures that entering the deleted name returns no match.

Figure 35-10 View Object LOV Attribute with Custom Property displayMode Defined



SEmpView.xml

General
Entity Objects
Attributes
Business Rules
Query
View Criteria
Row Finders
Java
Accessors
List UI Hints
UI Categories
Service Shaping

Attributes

Set Source Order

View object attributes can be mapped to entity attributes, calculated or SQL-derived.

Name	Type	Alias Name	Entity Usage	Info
Id	Integer	ID	SEmp	
LastName	String	LAST_NAME	SEmp	
FirstName	String	FIRST_NAME	SEmp	
Userid	String	USERID	SEmp	
Email	String	EMAIL	SEmp	
StartDate	Timestamp	START_DATE	SEmp	
Comments	String	COMMENTS	SEmp	
ManagerId	Integer	MANAGER_ID	SEmp	
TitleId	BigDecimal	TITLE_ID	SEmp	
DeptId	Integer	DEPT_ID	SEmp	

Details | UI Hints | Entity Attribute | Dependencies | Custom Properties | List of Values

Property	Value
displayMode	Name

Creating a Selection List

ADF Faces provides a number of different selection components, ranging from simple boolean radio buttons to list boxes that allow the user to select multiple items. You can create a model-driven selection list, a fixed selection list, or a dynamic selection list using different procedures.

ADF Faces Core includes components for selecting a single value and multiple values from a list. For example, `selectOneChoice` allows the user to select an item from a dropdown list, and `selectManyChoice` allow the user to select several items from a list of checkboxes. Selection lists are described in Table 35-1.

Table 35-1 ADF Faces Single and Multiple List Components

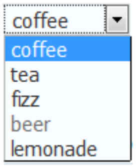
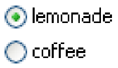
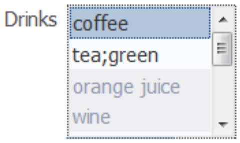
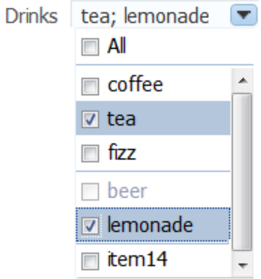
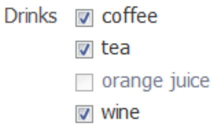
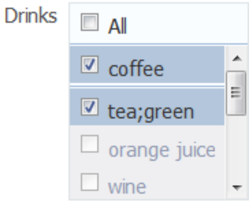
ADF Faces component	Description	Example
`SelectOneChoice`	Select a single value from a list of items.	

Table 35-1 (Cont.) ADF Faces Single and Multiple List Components

ADF Faces component	Description	Example
`SelectOneRadio`	Select a single value from a set of radio buttons.	
`SelectOneListbox`	Select a single value from a scrollable list of items.	
`SelectManyChoice`	Select multiple values from a scrollable list of checkboxes. Each selection displays at the top of the list.	 Drinks tea; lemonade □ All □ coffee ☑ tea □ fizz □ beer ☑ lemonade □ item14
`SelectManyCheckbox`	Select multiple values from a group of checkboxes.	
`SelectManyListbox`	Select multiple values from a scrollable list of checkboxes,	 Drinks □ All ☑ coffee ☑ tea;green □ orange juice □ wine

You can create selection lists using the `SelectOneChoice` ADF Faces component. The steps are similar for creating other single-value selection lists, such as `SelectOneRadio` and `SelectOneListbox`.

A databound selection list displays values from a data control collection or a static list and updates an attribute in another collection or a method parameter based on the user's selection. When adding a binding to a list, you use an attribute from the data control that will be populated by the selected value in the list.

> **Note:**
>
> By default, ADF Model list binding passes the index to the component. If you set `valuePassThru=true`, then you should set the list binding entry in the corresponding page definition file `Mode` attribute to `Object`. When `Mode=Object`, the list binding will pass an object instead of an index to the component. See What Happens When You Create a Fixed Selection List.

To create a selection list, you choose a base data source and a list data source in the Edit List Binding dialog:

- Base data source: Select the data collection that you want to bind to your control and that contains the attributes to be updated from user selections.
- List data source: Select the data collection that contains the attributes to display.

The data collection is based on the view object. For information about creating a view object, see How to Create an Entity-Based View Object.

You can create three types of selection lists in the Edit List Binding dialog:

- Model-driven list: List selections are based on a list of values bound to a data collection. This type of selection list offers significant advantages over the other two, as described in How to Create a Model-Driven List.
- Static list: List selections are based on a fixed list that you create a manually by entering values one at a time into the editor. See How to Create a Selection List Containing Fixed Values.
- Dynamic list: List selections are generated dynamically based on one or more databound attribute values. See How to Create a Selection List Containing Dynamically Generated Values.

How to Create a Model-Driven List

A model-driven list is based on a list of values that is bound to a view data object. Lists of Values are typically used in forms to enable an end user to select an attribute value from a dropdown list instead of having to enter it manually. When the user submits the form with the selected values, ADF data bindings in the ADF Model layer update the value on the view object attributes corresponding to the databound fields.

Note:

One way to create a model-driven list is to drag a collection from the Data Controls panel onto a JSF page, choose one of the ADF Forms in the context menu, and accept the defaults. The advantage is that if there are LOVs defined on the underlying view object attributes, all the LOVs on the entire form will be configured automatically. For more information, see How to Define a Single LOV-Enabled View Object Attribute.

You can also use the list of values as the basis for creating a selection list. The advantages of creating a model-driven list based on a list of values are:

- Reuse: The list of values is bound to a view data collection. Any selection list that you create based on the data collection can use the same list of values. Because you define the LOV on the individual attributes of view objects in a data model project with ADF Business Components, you can customize the LOV usage for an attribute once and expect to see the changes anywhere that the business component is used in the user interface.
- Translation: Values in the list of values can be included in resource bundles used for translation.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create an selection list component. For more information, see Creating a Selection List.

You may also find it helpful to understand functionality that can be used with selection lists. For more information, see Additional Functionality for Selection Lists and Shuttles.

You will need to complete these tasks:

1. Create a view object.
2. Create a view accessor on the object.
3. Create a list of values that is bound to an attribute on the base data source for the selection list. For example, you can create a list of values bound to the `CountryId` attribute of the `Addresses` view data object.

   For more information, see How to Define a Single LOV-Enabled View Object Attribute.

To create a model-driven selection list:

1. From the Data Controls panel, drag and drop the attribute onto the JSF page and, from the context menu, choose **Create > Single Selections > ADF Select One Choice**.
2. In the Edit List Binding dialog, click the **Model Driven List** radio button.
3. In the **Base Data Source Attribute** dropdown list, select an attribute on which you want to display a list of values.

   The **Base Data Source** should be the collection you dragged from the Data Controls panel. If the collection does not appear in the dropdown list, click the **Add** button to select the collection you want.

4. In the **Server List Binding Name** dropdown list, select the name of the list-of-values definition that was created for the base attribute.

   If a list-of-values definition was created for the attribute you have selected, the definition will appear in the dropdown list. For example, if you have defined `LOV_State` as the list-of-value definition for the **State** attribute, then **LOV_State** should appear in the **Server List Binding Name** dropdown list.

5. Click **OK**.

What Happens When You Create a Model-Driven Selection List

When you drag and drop an attribute from the Data Controls panel, JDeveloper does many things for you. For a full description of what happens and what is created when you use the Data Controls panel, see What Happens When You Create a Text Field.

The following example shows the page source code after you add a model-driven `SelectOneChoice` component to it.

```
<af:selectOneChoice value="#{bindings.State.inputValue}"
                    label="#{bindings.State.label}"
                    required="#{bindings.State.hints.mandatory}"
                    shortDesc="#{bindings.State.hints.tooltip}"
                    id="soc1">
                    <f:selectItems value="#{bindings.State.items}" id="si1"/>
</af:selectOneChoice>
```

The `f:selectItems` tag, which provides the list of items for selection, is bound to the `items` property on the `State` list binding object in the binding container. For more information about ADF data binding expressions, see Configuring the ADF Binding Filter.

In the page definition file, JDeveloper adds the list binding object definitions in the bindings element, as shown in the following example.

```
<executables>
    <variableIterator id="variables"/>
    <iterator Binds="Customers" RangeSize="25"
              DataControl="BackOfficeAppModuleDataControl"
              id="CustomersIterator"/>
</executables>
<bindings>
    <list IterBinding="CustomersIterator"
          StaticList="false" Uses="LOV_State"
          id="State" DTSupportsMRU="true"
          SelectItemValueMode="ListObject"/>
</bindings>
```

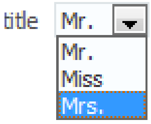
In the `list` element, the `id` attribute specifies the name of the list binding object. The `IterBinding` attribute references the variable iterator, whose current row is a row of attributes representing each variable in the binding container. The variable iterator exposes the variable values to the bindings in the same way as do other collections of data.

For more information about the page definition file and ADF data binding expressions, see Working with Page Definition Files, and Creating ADF Data Binding EL Expressions.

How to Create a Selection List Containing Fixed Values

You can create a selection list containing selections that you code yourself, rather than retrieving the values from another data source. See How to Create a Selection List Containing Dynamically Generated Values, for information about populating selection lists with values that are dynamically generated from another data source.

Figure 35-11 Selection List Bound to a Fixed List of Values



Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create a selection list component. For more information, see Creating a Selection List.

You may also find it helpful to understand functionality that can be used with selection lists. For more information, see Additional Functionality for Selection Lists and Shuttles.

You will need to complete this task:

Prepare a list of values that you will enter into the component as a fixed list.

To create a list bound to a fixed list of values:

1. From the Data Controls panel, drag and drop the attribute onto the JSF page and, from the context menu, choose **Create > Single Selections > ADF Select One Choice**.
2. In the Edit List Binding dialog, click the **Fixed List** radio button.
3. In the **Base Data Source Attribute** dropdown list, select an attribute on which you want to display a list of values.

   The **Base Data Source** should be the collection you dragged from the Data Controls panel. If the collection does not appear in the dropdown list, click the **Add** button to select the collection you want.
4. In the **Set of Values** box, enter each value you want to appear in the list. Press the Enter key to set a value before typing the next value. For example, if your attribute is Country, you could add the country codes `India`, `Japan`, and `Russia`.

   The order in which you enter the values is the order in which the list items are displayed in the `SelectOneRadio` control at runtime.

   The `SelectOneRadio` component supports a `null` value. If the user has not selected an item, the label of the item is shown as blank, and the value of the component defaults to an empty string. Instead of using blank or an empty string, you can specify a string to represent the `null` value. By default, the new string appears at the top of the list.

5. Click **OK**.

What Happens When You Create a Fixed Selection List

When you add a fixed selection list, JDeveloper adds source code to the JSF page and list and iterator binding objects to the page definition file.

The following example shows the page source code after you add a fixed `SelectOneChoice` component to it.

```
<af:selectOneChoice value="#{bindings.CountryIdStatic.inputValue}"
                            label="#{bindings.CountryIdStatic.label}">
          <f:selectItems value="#{bindings.CountryIdStatic.items}"/>
        </af:selectOneChoice>
```

The `f:selectItems` tag, which provides the list of items for selection, is bound to the `items` property on the `CountryId` list binding object in the binding container. For more information about ADF data binding expressions, see Creating ADF Data Binding EL Expressions.

In the page definition file, JDeveloper adds the definitions for the iterator binding objects into the `executables` element, and the list binding object into the `bindings` element, as shown in the following example.

```
<executables>
    <iterator Binds="Addresses1" RangeSize="10"
              DataControl="BackOfficeAppModuleDataControl"
              id="Addresses1Iterator"/>
  </executables>
  <bindings>
    <list IterBinding="Addresses1Iterator" id="CountryIdStatic" ListOperMode="0"
          StaticList="true">
      <AttrNames>
        <Item Value="CountryIdStatic"/>
      </AttrNames>
      <ValueList>
        <Item Value="India"/>
        <Item Value="Japan"/>
        <Item Value="Russia"/>
      </ValueList>
    </list>
  </bindings>
```

For complete information about page definition files, see Working with Page Definition Files.

How to Create a Selection List Containing Dynamically Generated Values

You can populate a selection list component with values dynamically at runtime.

> **Tip:**
>
> Another option is to create a static view object or a database view object within a **shared application module**. Then use a model- driven LOV to create the list. This provides caching and translatability.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create a selection list component. For more information, see Creating a Selection List.

You may also find it helpful to understand functionality that can be used with selection lists. For more information, see Additional Functionality for Selection Lists and Shuttles.

You will need to complete these tasks:

1. Define a data source for the list data source that provides the dynamic list of values.
2. Define a data source for the base data source that is to be updated based on the user's selection.

To create a selection list bound containing dynamically generated values:

1. From the Data Controls panel, drag and drop the attribute onto the JSF page and, from the context menu, choose **Create > Single Selections > ADF Select One Choice**.
2. In the Edit List Binding dialog, click the **Dynamic List** radio button.
3. Select the **Base Data Source** from the dropdown list or click **Add**.
4. In the Add Data Source dialog, select the data collection that contains the attribute to be updated and click **OK**.
5. Select the **List Data Source** from the dropdown list or click **Add**.
6. In the Add Data Source dialog, select the data collection that contains the attribute that will be displayed and click **OK**.

   > **Note:**
   >
   > The list and base collections do not have to form a **master-detail relationship**, but the attribute in the list collection must have the same type as the base collection attributes.

7. In the Data Mapping section, you can accept the default mapping or select different attributes items from the **Data Value** and **List Attribute** lists to update the mapping.

   The **Data Value** control contains the attribute on the data collection that is updated when the user selects an item in the selection list. The **List Attribute** control contains the attribute that populates the values in the selection list.

   To add a second mapping, click **Add**.

8. In the List Items section, select values for:
   - **Display Attribute**: Select the attribute you want to be displayed.
   - **"No Selection" Item**: Specify whether or not the list displays a NULL value selection and how it is displayed.
   - **MRU**: Select to show a list of most recently used items. Specify the number of items to display in the MRU list and the separator to be used.
9. Click **OK**.

What Happens When You Create a Dynamic Selection List

When you add a dynamic selection list to a page, JDeveloper adds source code to the JSF page and list and iterator binding objects to the page definition file.

The following example shows the page source code after you add a dynamic `SelectOneChoice` component to it.

```
<af:selectOneChoice value="#{bindings.SalesRepId.inputValue}"
                    label="#{bindings.SalesRepId.label}"
                    required="#{bindings.SalesRepId.hints.mandatory}"
                    shortDesc="#{bindings.SalesRepId.hints.tooltip}" id="soc2">
        <f:selectItems value="#{bindings.SalesRepId.items}" id="si4"/>
        <f:validator binding="#{bindings.SalesRepId.validator}"/>
</af:selectOneChoice>
```

The `f:selectItems` tag, which provides the list of items for selection, is bound to the `items` property on the `SalesRepId` list binding object in the binding container. For more information about ADF data binding expressions, see Creating ADF Data Binding EL Expressions.

In the page definition file, JDeveloper adds the definitions for the iterator binding objects into the `executables` element, and the list binding object into the `bindings` element, as shown in the following example.

```
<executables>
    <variableIterator id="variables"/>
    <iterator Binds="SummitAppModuleDataControl.dataProvider.
              BackOfficeAM.Customers"
              DataControl="SummitAppModuleDataControl" RangeSize="25"
              id="CustomersIterator"/>
    <iterator Binds="Countries" RangeSize="10"
              DataControl="BackOfficeAppModuleDataControl"
              id="CountriesIterator"/>
    <iterator Binds="Customers" RangeSize="-1"
              DataControl="BackOfficeAppModuleDataControl"
              id="CustomersIterator1"/>
</executables>
<bindings>
    <list IterBinding="CountriesIterator" StaticList="false" id="SalesRepId"
          DTSupportsMRU="true" SelectItemValueMode="ListObject"
          ListIter="CustomersIterator1">
      <AttrNames>
        <Item Value="Id"/>
      </AttrNames>
      <ListAttrNames>
        <Item Value="Id"/>
      </ListAttrNames>
      <ListDisplayAttrNames>
```

```
        <Item Value="Id"/>
      </ListDisplayAttrNames>
    </list>
</bindings>
```

By default, JDeveloper sets the RangeSize attribute on the iterator element for the Customers iterator binding to a value of -1 thus allowing the iterator to furnish the full list of valid products for selection. In the list element, the id attribute specifies the name of the list binding object. The IterBinding attribute references the iterator that iterates over the Countries collection. The ListIter attribute references the iterator that iterates over the Customers collection. The AttrNames element specifies the base data source attributes returned by the base iterator. The ListAttrNames element defines the list data source attributes that are mapped to the base data source attributes. The ListDisplayAttrNames element specifies the list data source attribute that populates the values users see in the list at runtime.

For complete information about page definition files, see Working with Page Definition Files.

What You May Need to Know About Values in a Selection List

Once you have created a list binding, you may want to access a value in the list. You can set the component's valuePassThru attribute to true either in the visual editor or in the Property Inspector. The following example shows the code for a selectOneChoice component.

```
 <af:selectOneChoice value="#{bindings.Language.inputValue}"
                     label="#{bindings.Language.label}"
                     required="#{bindings.Language.hints.mandatory}"
                     shortDesc="#{bindings.Language.hints.tooltip}"
                     valueChangeListener="#{myBean.valueChanged}"
                     id="soc1" valuePassThru="true">
      <f:selectItems value="#{bindings.Language.items}" id="si1"/>
</af:selectOneChoice>
```

Only when the valuePassThru attribute is set to true and the Mode attribute in the corresponding page definition file is set to Object can the value passed to the valueChangeListener be of type Object (which is the actual value). By default, Mode is set to Index. The following example shows the Mode attribute in the page definition file.

```
<list IterBinding="AvailableLanguagesView1Iterator" id="Language"
      DTSupportsMRU="true" StaticList="false"
      ListIter="CountryCodesView1Iterator"
      Mode="Object"/>
```

Creating a List with Navigation List Binding

In ADF Faces, navigation list binding lets users navigate through the objects in a collection. As the user changes the current object selection using the navigation list component, any other component that is also bound to the same collection through its attributes will display from the newly selected object.

In addition, if the collection whose current row you change is the master view object instance in a data model master-detail relationship, the row set in the detail view object instance is automatically updated to show the appropriate data for the new current master row.

How to Create a List with Navigation List Binding

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create a navigation list. For more information, see Creating a List with Navigation List Binding.

You may also find it helpful to understand functionality that can be used with navigation lists. For more information, see Additional Functionality for Selection Lists and Shuttles.

To create a list that uses navigation list binding:

1. From the Data Controls panel, drag and drop the master collection to the page and choose **Create > Navigation > ADF Navigation List**.

   The master collection is where the user selects from.

2. In the Edit List Binding dialog, from the **Base Data Source** dropdown list, select the collection whose members will be used to create the list.

   This should be the collection you dragged from the Data Controls panel. If the collection does not appear in the dropdown list, click the **Add** button to select the collection you want.

3. From the **Display Attribute** dropdown list, select the attribute that will display in the list. You can choose the selection that includes all the attributes, and you can choose **Select Multiple** to launch a selection dialog.
4. In the Select Multiple Display Attributes dialog, shuttle the attributes you want to display from the **Available Attributes** list to the **Attributes to Display** list. Click **OK** to close the dialog.
5. Click **OK**.
6. From the Data Controls panel, drag and drop the detail collection inside the **Details** section of the page as an output component.

   The output component can be any component that is used to display the data. If you decide to use a table, you can choose **Create> Table > ADF Table**.

   The **Detail** section is an `af:panelHeader` component that was added by JDeveloper when you created the navigation list in Step 1.

What Happens When You Create a Navigational List Binding

When you add a navigation list to a page, JDeveloper adds source code to the JSF page and list and iterator binding objects to the page definition file.

On the page, JDeveloper adds a `selectOneChoice` component with a `f:selectItems` tag. The `selectOneChoice` component allows the user to select from a list of values based on the selected attribute in the source data collection. JDeveloper also adds an `af:panelHeader` component that will contain the component you chose to display the data based on the selection. In this example, a table component is added to display the data. Note that the `af:panelHeader` component has `partialtriggers` set to the `selectOneChoice` component `id` so that when the selection changes, new data will be displayed. The following example shows the page source code after you add a navigation list.

```
<af:selectOneChoice id="nl1" autoSubmit="true"
    value="#{bindings.Customers.inputValue}"
    label="#{bindings.Customers.label}">
    <f:selectItems value="#{bindings.Customers.items}" id="si1"/>
</af:selectOneChoice>
<af:panelHeader text="Details" partialTriggers="nl1" id="ph1">
    <af:table value="#{bindings.OrdersForCustomer.collectionModel}" var="row"
        rows="#{bindings.OrdersForCustomer.rangeSize}"
        emptyText="#{bindings.OrdersForCustomer.viewable ?
                   'No data to display.' : 'Access Denied.'}"
        fetchSize="#{bindings.OrdersForCustomer.rangeSize}"
rowBandingInterval="0"
        id="t1">
     <af:column headerText="#{bindings.OrdersForCustomer.hints.Id.label}"
id="c1">
        <af:inputText value="#{row.bindings.Id.inputValue}"
...
```

In the page definition file, JDeveloper adds the definitions for the iterator binding objects into the `executables` element. The following example shows the iterator bindings and navigation list bindings in the page definition file.

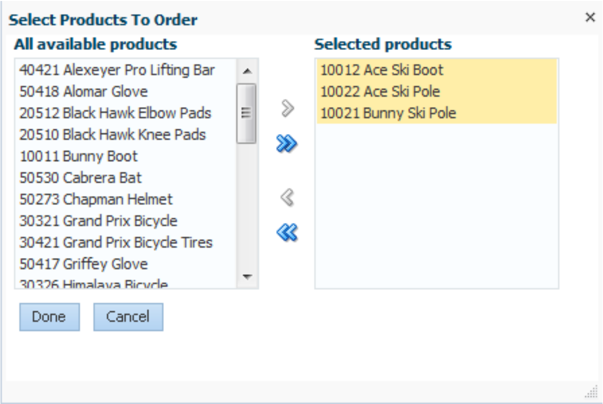
```
<executables>
    <variableIterator id="variables"/>
    <iterator Binds="Customers" RangeSize="25"
         DataControl="BackOfficeAppModuleDataControl" id="CustomersIterator"/>
    <iterator Binds="OrdersForCustomer" RangeSize="25"
         DataControl="BackOfficeAppModuleDataControl"
              id="OrdersForCustomerIterator"/>
</executables>
<bindings>
    <navigationlist IterBinding="CustomersIterator" ListOperMode="navigation"
              ListIter="CustomersIterator"
              id="Customers" DTSupportsMRU="false">
      <AttrNames>
        <Item Value="Name"/>
      </AttrNames>
    </navigationlist>
    <tree IterBinding="OrdersForCustomerIterator" id="OrdersForCustomer">
      <nodeDefinition DefName="oracle.summit.model.views.OrdVO"
             Name="OrdersForCustomer0">
        <AttrNames>
          <Item Value="Id"/>
          <Item Value="CustomerId"/>
          <Item Value="DateOrdered"/>
          <Item Value="LastName"/>
          <Item Value="Name"/>
        </AttrNames>
      </nodeDefinition>
    </tree>
</bindings>
```

Creating a Databound Shuttle

ADF Faces provides shuttle components that provides a mechanism for selecting multiple values from a list of values by allowing the user to move items between two lists. Use the selectManyShuttle and selectOrderShuttle components to create a databound shuttle.

The selectManyShuttle and selectOrderShuttle components render two list boxes and buttons that allow the user to select multiple items from the leading (or available) list box and to move or shuttle those items over to the trailing (or selected) list box, and vice versa. You can also double-click an item to move it to the other list box. Figure 35-12 shows an example of a rendered selectManyShuttle component that is implemented in the Summit ADF sample application. You can specify any text you want for the headers that display above the list boxes.

Figure 35-12 SelectManyShuttle Component



40421 Alexeyer Pro Lifting Bar
50418 Alomar Glove
20512 Black Hawk Elbow Pads
20510 Black Hawk Knee Pads
10011 Bunny Boot
50530 Cabrera Bat
50273 Chapman Helmet
30321 Grand Prix Bicycle
30421 Grand Prix Bicycle Tires
50417 Griffey Glove
30326 Himalaya Bicycle

10012 Ace Ski Boot
10022 Ace Ski Pole
10021 Bunny Ski Pole

The only difference between selectManyShuttle and selectOrderShuttle is that in the selectOrderShuttle component, the user can reorder the items in the trailing list box by using the up and down arrow buttons on the side.

The Summit ADF sample application uses a selectManyShuttle component to allow the user to select the products to order from a **All available products** list box to an **Selected Products** list box. The leading list box on the left displays all the available products. The trailing list box on the right displays the products the customer has selected.

Like other ADF Faces selection list components, the selectManyShuttle and selectOrderShuttle components can use the f:selectItems tag to provide the list of items available for display and selection in the leading list.

Before you can bind the f:selectItems tag, create a generic class that can be used by any page that requires a shuttle. In the class, declare and include getter and setter methods for the properties that describe the view object instance names that should be used for the list of all available choices (leading list or available product categories) and the list of selected choices (trailing list or assigned product categories). In the Summit ADF sample application, f:selectItems uses the binding container to retrieve the list of available products. Example 35-1 shows the ShuttleBean class that is created to manage the population and selection state of the shuttle component on the showshuttle.jsf page.

The getSelectedProducts() method returns a List that defines the items in the shuttle's trailing list. The values returned by getSelectedProducts() are a subset of the values in the leading list. Values that are in the leading list but not in getSelectedProducts() will be ignored (and will generate an error in the server log).

Note that the ShuttleBean class can call utility methods in the ADFUtils class if required. Also note that this class uses values for several properties of the base bean.

Example 35-2 shows the managed bean and managed properties configured in order-select-many-item.xml task flow for working with the shuttle component.

Example 35-1 Shuttle Bean Class

```
public class ShuttleBean {

    private List<oracle.jbo.domain.Number> _selectedProducts =
         new ArrayList<oracle.jbo.domain.Number>();

    public ShuttleBean()
    {
    }

    public void setSelectedProducts(List<oracle.jbo.domain.Number>
         _allocatedRoles) {
        this._selectedProducts = _allocatedRoles;
    }

    public List<oracle.jbo.domain.Number> getSelectedProducts() {
        return _selectedProducts;
    }

    public void onAddMultipleProducts(ActionEvent actionEvent) {
      DCBindingContainer dcBindings =

(DCBindingContainer)BindingContext.getCurrent().getCurrentBindingsEntry();
      OperationBinding createMethod =
         dcBindings.getOperationBinding("CreateInsert");
      if ( createMethod != null && _selectedProducts.size() > 0 ) {
            DCIteratorBinding iterator =
                 dcBindings.findIteratorBinding("ItemsForOrderIterator");
            for ( oracle.jbo.domain.Number productId : _selectedProducts )
            {
                createMethod.execute();
                iterator.getCurrentRow().setAttribute("ProductId", productId );
            }
        }
    }
}
```

Example 35-2 Managed Bean for the Shuttle Component in the order-select-many-items Task Flow

```
<managed-bean id="__3">
      <managed-bean-name>ShuttleBean</managed-bean-name>
      <managed-bean-class>oracle.summit.backing.ShuttleBean</managed-bean-class>
      <managed-bean-scope>backingBean</managed-bean-scope>
</managed-bean>
<view id="showshuttle">
      <page>/orders/showshuttle.jsf</page>
```

How to Create a Databound Shuttle

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create a shuttle. For more information, see Creating a Databound Shuttle.

You may also find it helpful to understand functionality that can be used with selection lists. For more information, see Additional Functionality for Selection Lists and Shuttles.

You will need to complete these tasks:

1. Create the relevant iterator bindings in the page definition file. Use Example 35-3 as a guide.

   In the `Summit` ADF sample application, a list binding for `ProductId` with `ItemsForOrderIterator` is added to the `showshuttlePageDef.xml` file to provide the list of products for populating the leading list.

2. Create a class similar to the `ShuttleBean` class and provide methods to process the values as required. Use Example 35-1 as a guide.

3. Configure the required managed bean and managed properties in the task flow definition or, in the case of an unbounded task flow, the `adfc-config.xml` file. Use Example 35-2 as a guide.

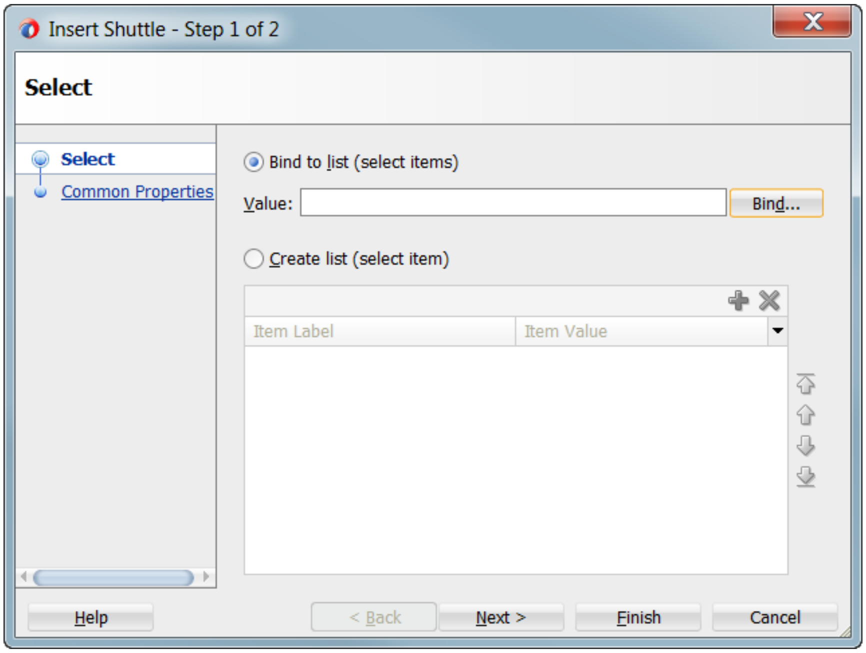
To create a shuttle component:

1. In the Applications window, double-click the web page that will display the component.

2. In the ADF Faces page of the Components window, from the Text and Selection panel, drag and drop **SelectManyShuttle** or **SelectOrderShuttle** onto the page.

   JDeveloper displays the Insert Shuttle wizard, as illustrated in Figure 35-13.

Figure 35-13   Insert SelectManyShuttle Wizard



3. Select **Bind to list (select items)** and click **Bind**.

4. In the Expression Builder, expand tree structure to create an expression that will populate the values for the leading list and click **OK.**

In the Summit ADF sample application, select **ADF Bindings > bindings > ProductId > items** to build the expression #{bindings.ProductId.items} and click **Finish**. This binds the f:selectedItem tag of the selectManyShuttle component to the data collection that populates the shuttle's leading list with a list of products.

If you decide to retrieve the values for the leading list using a method, for example, getAllProducts(), you should create this method in the managed bean. You would then bind to this method in the expression builder.

5. Click **Next**.
6. In the Common Properties page, click **Bind** next to the **Value** field.
7. In the Expression Builder, expand the tree to create an expression that will populate the values for the trailing list and click **OK**.

   In the Summit ADF sample application, select **ADF Managed Beans > BackingBeanScope > ShuttleBean > selectedProducts** to build the expression #{backingBeanScope.ShuttleBean.selectedProducts} and click **Finish**. This binds the value attribute to the selectedProducts() method that populates the shuttle's trailing list.
8. You may want to add additional controls to process the selected values.

   In the Summit ADF sample application, an af:button component labeled **Done** is added to the page to process the selected products by calling the onAddMultipleProducts method defined in the ShuttleBean.

What Happens When You Create a Databound Shuttle

All the bindings of the showshuttle.jsf page are defined in the file showshuttlePageDef.xml. Example 35-3 shows part of the page definition file for the showshuttle.jsf page.

Example 35-4 shows the code for the selectManyShuttle component after you complete the Insert SelectManyShuttle dialog.

Example 35-3 Page Definition File for the showshuttle Page

```
<?xml version="1.0" encoding="UTF-8" ?>
<pageDefinition xmlns="http://xmlns.oracle.com/adfm/uimodel"
 version="12.1.2.61.86" id="showshuttlePageDef" Package="orders">
  <parameters/>
  <executables>
    <variableIterator id="variables"/>
    <iterator Binds="ItemsForOrder" RangeSize="25"
              DataControl="BackOfficeAppModuleDataControl"
              id="ItemsForOrderIterator"/>
  </executables>
  <bindings>
    <list IterBinding="ItemsForOrderIterator" StaticList="false"
          Uses="LOV_ProductId" id="ProductId"
          DTSupportsMRU="true" SelectItemValueMode="ListObject"/>
    <action IterBinding="ItemsForOrderIterator" id="CreateInsert"
            RequiresUpdateModel="true" Action="createInsertRow">
    </action>
  </bindings>
</pageDefinition>
```

Example 35-4   SelectManyShuttle Component in showshuttle.jsff File

```
<af:selectManyShuttle id="sms1"
    value="#{backingBeanScope.ShuttleBean.selectedProducts}"
    leadingHeader="All available products" trailingHeader="Selected products">
    <f:selectItems value="#{bindings.ProductId.items}" id="si1"/>
</af:selectManyShuttle>
```

36
Creating ADF Databound Search Forms

This chapter describes how to create search forms from data modeled from ADF Business Components, using ADF data binding and ADF Faces components to perform complex searches on multiple attributes and search forms to search on a single attribute. For complex query search forms, it describes how to set up the query search form mode, results table, saved searches list, and personalization. For single attribute search forms, it describes how to configure the form layout. In addition, it includes information on using named bind variables and Query-by-Example (QBE) filtered table searches.
This chapter includes the following sections:

About Creating Search Forms

Oracle ADF supports a context option that helps to create a quick and effective search page. Based on different search criteria, you can either use query search forms or quick query search forms.

You can create search forms that allow users to enter search criteria into input fields for known attributes of an object. The search criteria can be entered via input text fields or selected from a list of values in a popup list picker or dropdown list box. The entered criteria is constructed into a query to be executed. **Named bind variables** can be used to supply attribute values during runtime for the query. The results of the query can be displayed as a table, a form, or another UI component.

Oracle ADF supports two types of search forms: query and quick query. The **query search form** is a full-featured search form. The **quick query search form** is a simplified form with only one search criteria. Each of these search forms can be combined with a filtered table to display the results, thereby enabling additional search capabilities. You can also create a standalone filtered table to perform searches without the query or quick query search panel.

Search forms are based either on **view criteria** defined in **view objects** or on implicit view criteria defined by JDeveloper. Search forms are region-based components that are reusable and personalizable. They encapsulate and automate many of the actions and iterator management operations required to perform a query. You can create several search forms on the same page without any need to change or create new iterators.

A **filtered table** is a table that has additional Query-by-Example (QBE) search criteria fields above each searchable column. When the filtering option of a table is enabled, you can enter QBE-style search criteria for each column to filter the query results. For more information about tables, see Creating ADF Databound Tables.

For information about individual query and table components, see the Using Query Components and the Using Tables, Trees, and Other Collection-Based Components chapters of *Developing Web User Interfaces with Oracle ADF Faces*.

Implicit and Named View Criteria

When you create data controls, all data collections will automatically include an implicit view criteria node in the **Named Criteria** node with an **All Queriable Attributes** criteria. This is the default view criteria that includes all the searchable attributes or columns of the data collection. You cannot edit or modify this implicit view criteria. Implicit view criteria can be used in the same way as declaratively created (or named) view criteria during the creation of query and quick query search forms. For more information about creating named view criteria, see Working with Named View Criteria.

When you add additional named view criteria for that view object or collection, the new view criteria will be added to the **Named Criteria** node in the Data Controls panel with the default implicit view criteria. In the Data Controls panel, a data collection's Named Criteria node will always include the implicit view criteria, regardless of whether any named view criteria were defined. The implicit view criteria is always available for every data collection.

> **Note:**
>
> Query search forms have certain restrictions for working with expressions that have nested view criteria and may not work with all types of nested expressions. For more information about the nested expressions supported by search forms, see What You May Need to Know About Nested View Criteria Expressions.

List of Values (LOV) Input Fields

List of values (LOV) components are input components that allow the user to enter values by picking from a list that is generated by a query. ADF Faces provides the `af:inputListOfValues` and `af:inputComboboxListOfValues` components. If you are using dependent LOVs as part of your search form, you must use them with the `af:query` component. For more information about LOV components, see Creating List of Values (LOV) Components.

If an attribute is defined as an LOV, you can set the **Support Multiple Value Selection** control hint in its view criteria to enable users to make multiple selections in the search criteria field. If multiple selection is enabled on an LOV attribute, and the `Equal to` or `Not equal to` operator is chosen, a `selectManyChoice` component will render in the query panel. The user can select multiple items as the search criteria.

When the LOV is in a query component, if the **Support Multiple Value Selection** hint is not set and the `Equal to` or `Not equal to` operator is chosen, the query component will render a search criteria component according to the **Default List Type** control hint for the corresponding attribute in the view object.

The quick query component does not support multiple selection. It will always render the component specified by the **Default List Type** control hint. Table 36-1 shows the control hint selection and the default list component.

Table 36-1 Query and Quick Query Search Criteria Field Input Components

Default List Type Control Hint	Component
Input Text with List of Values	`af:inputListOfValues`
Combo Box with List of Values	`af:inputComboboxListOfValues`
Choice List, Combo Box, List Box, Radio Group	`af:selectOneChoice`

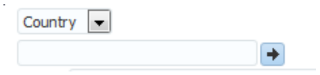
For more information about view criteria options, see How to Create Named View Criteria Declaratively, and What You May Need to Know About Bind Variables in View Criteria.

Search Form Use Cases and Examples

The search forms are based on the model-driven `af:query` and `af:quickQuery` components. Because these underlying components are model-driven, the search form will change automatically to reflect changes in the model. The view layer does not need to be changed. For example, if you define a list of values (LOV) on an attribute in the view object or entity object, the LOV will automatically show up in the search form as an LOV component. Or, if you modify a view criteria to include a new attribute for the `WHERE` clause, the search panel using this view criteria will automatically reflect that change by adding a search field for that attribute.

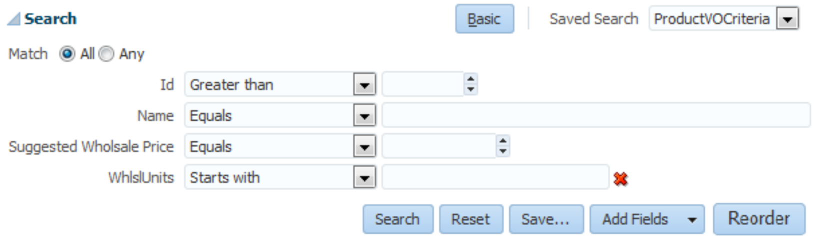
Figure 36-1 shows a quick query component.

Figure 36-1 Quick Query



For more complex searches, such as including criteria for product name, price ranges, and availability, a query component is used. You can create the query component in several different modes to give the user different capabilities, as shown in Figure 36-2. The user can even add their own search criteria using the **Add Fields** button. The user can save the searches that were created to be used later.

Figure 36-2 Query Component



Saved Search ProductVOCriteria

Match ● All ○ Any

Id Greater than

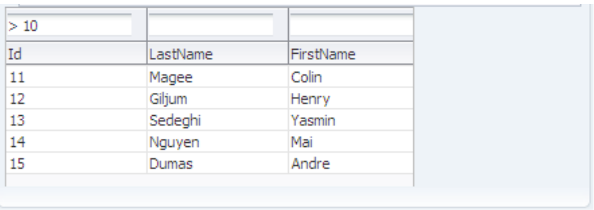
Name Equals

Suggested Wholsale Price Equals

WhlslUnits Starts with

For simple searches, you can use the filtered table to provide Query-by-Example (QBE) searches, as shown in Figure 36-3.

Figure 36-3 Filtered Table Searches



> 10		
Id	LastName	FirstName
11	Magee	Colin
12	Giljum	Henry
13	Sedeghi	Yasmin
14	Nguyen	Mai
15	Dumas	Andre

Additional Functionality for Search Forms

You may find it helpful to understand other ADF features before you configure or use the ADF Model layer. Additionally, you may want to read about what you can do with your model layer configurations. Following are links to other functionality that may be of interest.

- Both the query and quick query components are based on the view criteria defined in a view object. You use the view criteria to set up the initial search fields and search criteria. You use control hints to further define the query component. For information about creating named view criteria, see Working with Named View Criteria.
- Since filtered tables also provide search functions, you should consider them for simple searches. For information about tables, see Creating ADF Databound Tables.
- List of value components use the query component in their search panel for creating the list of values selection list. For information about LOV components, see Creating List of Values (LOV) Components.
- If your application is set up with a Metadata Storage (MDS), you can persist saved searches across transactions and personalize your searches. Apart from column order and column display, you can save the sort order of the results layout. The sort order is saved as part of a Saved Search. You can restore the saved search including the sort order of the results layout by re-invoking this Saved Search entry. This means that apart from display index and visible properties of the column, the sort order property is also saved as part of the saved layout. For information about using MDS, see Customizing Applications with MDS.

Creating Query Search Forms

Query search form is a full-featured search form that is based on a view criteria defined in ADF Business Components view object. You can use different methods to create query search forms with binding variables.

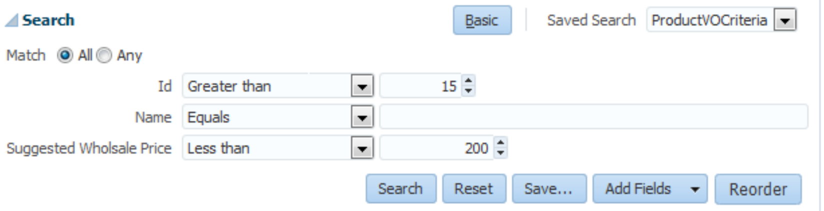
The query search form is the standard form for complex transactional searches. You can build complex search forms with multiple search criteria fields each with a dropdown list of built-in operators. You can also add custom operators and customize the list. The query search form supports lists of values, `AND` and `OR` conjunctions, and saving searches for future use.

A query is associated with the view object that it uses for its query operation. In particular, a query component is the visual representation of the view criteria defined for that view object. For information on creating view criteria, see Working with Named View Criteria. The view criteria items defined in the view criteria is the source for the search criteria in the search panel of the query component. The **Rendered Mode** option in the view criteria editor controls the search criteria will be displayed.

Search criteria can be grouped together to provide a logical grouping of attributes. For instance, you can group the address attributes together.

A query search form has a basic mode and an advanced mode. The user can toggle between the two modes using the basic/advanced button. At design time, you can declaratively specify form properties (such as setting the default state) to be either basic or advanced. Figure 36-4 shows an advanced mode query search form with three search criteria.

Figure 36-4 Advanced Mode Query Search Form



Saved Search: ProductVOCriteria

Match ● All ○ Any

Id: Greater than 15

Name: Equals

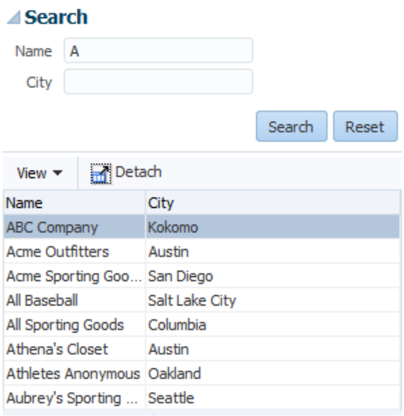
Suggested Wholsale Price: Less than 200

The advanced mode query form features are:

- Selecting search criteria operators from a dropdown list
- Adding custom operators and deleting standard operators
- Selecting `WHERE` clause conjunctions of either `AND` or `OR` (match all or match any)
- Dynamically adding and removing search criteria fields at runtime
- Saving searches for future use
- Personalizing saved searches

Typically, the query search form in either mode is used with an associated results table or tree table. In the Summit sample application for Oracle ADF, the query panel and results table is shown in Figure 36-5.

Figure 36-5   Results Table for a Query Search

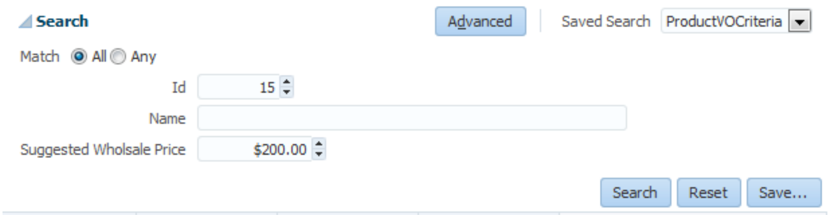


Name: A
City:

Name	City
ABC Company	Kokomo
Acme Outfitters	Austin
Acme Sporting Goo...	San Diego
All Baseball	Salt Lake City
All Sporting Goods	Columbia
Athena's Closet	Austin
Athletes Anonymous	Oakland
Aubrey's Sporting ...	Seattle

The basic mode has all the features of the advanced mode except that it does not allow the user to dynamically add search criteria fields. Figure 36-6 shows a basic mode query search form with three search criteria field. Notice the lack of a dropdown list next to the **Save** button used to add search criteria fields in the advanced mode.

Figure 36-6   Basic Mode Query Form with Three Search Criteria Field



Saved Search: ProductVOCriteria
Match ● All ○ Any
Id: 15
Name:
Suggested Wholsale Price: $200.00

In either mode, each search criteria field can be modified by selecting operators such as `Greater Than` and `Equal To` from a dropdown list, and the entire search panel can be modified by the **Match All/Any** radio buttons. Partial page rendering is also supported by the search forms in almost all situations. For example, if a `Between` operator is chosen, another input field will be displayed to allow the user to select the upper range.

A `Match All` selection implicitly uses `AND` conjunctions between the search criteria in the `WHERE` clause of the query. A `Match Any` selection implicitly uses `OR` conjunctions in the `WHERE` clause. Here is a simplified `WHERE` clause (the real `WHERE` in the view criteria is different) when `Match All` is selected for the search criteria shown in Figure 36-4:

```
WHERE (Id < 4) AND (InStock > 20) AND (Name="Shoes")
```

This is a simplified `WHERE` clause if **Match Any** is selected for the search criteria shown in Figure 36-4.

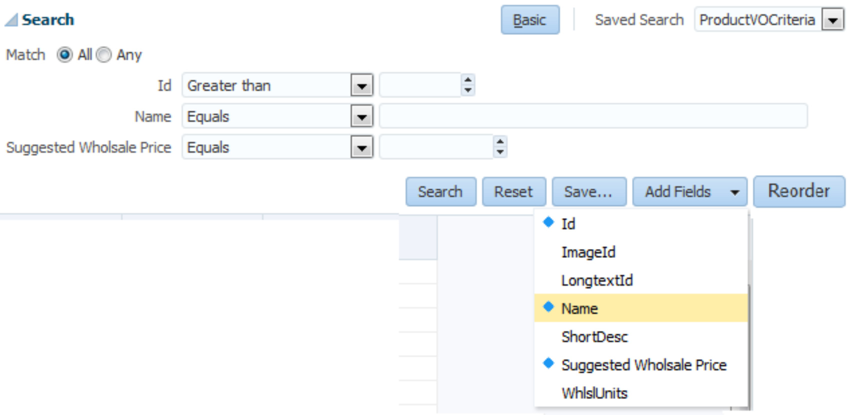
```
WHERE (Id < 4) OR (InStock > 20) OR (Name="Shoes")
```

If the view criteria for the query has mixed `AND` and `OR` conjunctions between the criteria items, then neither **Match All** nor **Match Any** will be selected when the component first renders. However, the user can select **Match All** or **Match Any** to override the conjunctions defined as the initial state in the view criteria.

Advanced mode query forms allow users to dynamically add search criteria fields to the query panel to perform more complicated queries. These user-created search criteria fields can be deleted, but the user cannot delete existing fields. You use the **Add Fields** dropdown list to add a search criteria field or all the fields within the group.

Figure 36-7 shows a query search form and how the **Add Fields** dropdown list is selected to add a criteria field to the search form.

Figure 36-7   Dynamically Adding Search Criteria Fields at Runtime



Saved Search: ProductVOCriteria

Match ● All ○ Any

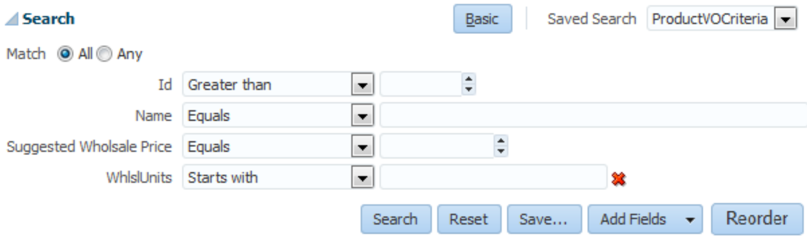
Id: Greater than

Name: Equals

Suggested Wholsale Price: Equals

Figure 36-8 shows a query search form with a new user-added search criteria with the delete icon to its right. Users can click the delete icon to remove the criteria.

Figure 36-8   User-Added Search Criteria with Delete Icon



Saved Search: ProductVOCriteria

Match ● All ○ Any

Id: Greater than

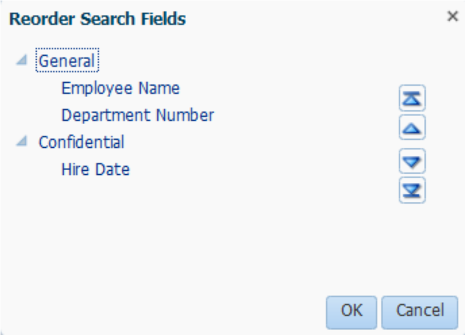
Name: Equals

Suggested Wholsale Price: Equals

WhlslUnits: Starts with ✖

You can use the **Reorder** button to open the Reorder Search Fields dialog to reorder the search fields in the search panel, as shown in Figure 36-9.

Figure 36-9 Reorder Search Fields dialog

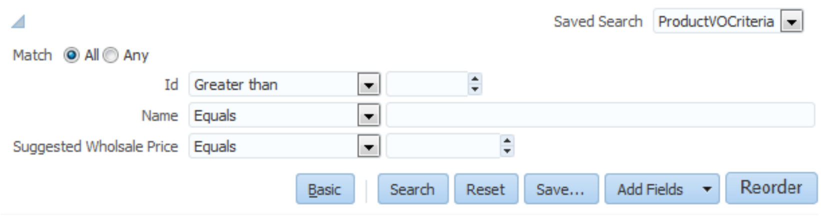


If either **Match All** or **Match Any** is selected and then the user dynamically adds the second instance of a search criterion, then both **Match All** and **Match Any** will be deselected. The user must reselect either **Match All** or **Match Any** before clicking the **Search** button.

If you intend for a query search form to have both a basic and an advanced mode, you can define each search criteria field to appear only for basic, only for advanced, or for both. When the user switches from one mode to the other, only the search criteria fields defined for that mode will appear. For example, suppose three search fields for basic mode (A, B, C) and three search fields for advanced mode (A, B, D) are defined for a query. When the query search form is in basic mode, search criteria fields A, B, and C will appear. When it is in advanced mode, then fields A, B, and D will appear. Any search data that was entered into the search fields will also be preserved when the form returns to that mode. If the user entered 35 into search field C in basic mode, switched to advanced mode, and then switched back to basic, field C would reappear with value 35.

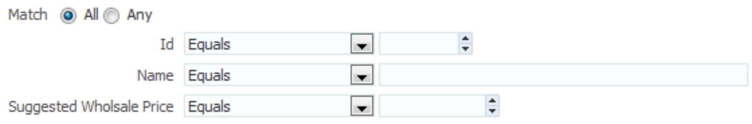
Along with using the basic or advanced mode, you can also determine how much of the search form will display. The default setting displays the whole form. You can also configure the query component to display in compact mode or simple mode. The compact mode has no header or border, and the **Saved Search** dropdown lists moves next to the expand/collapse icon. Also, the **Basic** button has been moved to the bottom of the form and the Search label is omitted. Figure 36-10 shows a query search form set to compact mode.

Figure 36-10 Query Component in Compact Mode


Saved Search ProductVOCriteria
Match ● All ○ Any
Id Greater than
Name Equals
Suggested Wholsale Price Equals

The simple mode displays the component without the header and footer, and without the buttons normally displayed in those areas. Figure 36-11 shows the same query search form set to simple mode.

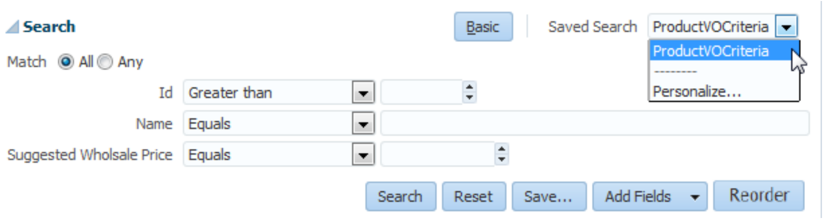
Figure 36-11 Query Component in Simple Mode



If there are multiple view criteria defined, each can be selected from the **Saved Search** dropdown list. These saved searches are created at design time and are called **system searches**. For example, if you have defined two view criteria in the view object, both view criteria are available for selection in the query search form Saved Search dropdown list, as shown in Figure 36-12.

> **Note:**
>
> Be aware that opening and closing the query panel by clicking the panel expansion/collapse icon does not reset the applied view criteria. You can click the **Reset** button to reset the applied view criteria.

Figure 36-12 Query Form Saved Search Dropdown List



If there are no explicitly defined view criteria for a view object, you can use the default implicit view criteria.

Users can also create saved searches at runtime to save the state of a search for future use. Apart from display index and visible properties of the column, the sort order property is also saved as part of the saved layout. In other words, apart from column order and column display, the sort order of each column is saved as part of a saved search. This means that the user can sort any column on the results table layout in ascending or descending order and subsequently save the sorted layout of the results table as part of the saved search. When the user invokes this saved search again, the sorted layout of the results table displays. If no sort order is specified on any columns on the layout of the results table of a query search, and the user saves the search as a Saved Search, the sort order of the layout of the results table that is saved, is the sort order specified in the view object. The entered search criteria values, the basic/advanced mode state, and the layout of the results table/ component can be saved by clicking the **Save** button to open a Save Search dialog, as shown in Figure 36-13. Saved searches appear in alphabetical order in the **Saved Search** dropdown list. User-created saved searches persist for the session. If they are intended to be available beyond the session, you must configure a persistent data store to store them. For Oracle ADF, you can use an access-controlled data source such as MDS. For more information about using MDS, see Customizing Applications with MDS .

When you perform a saved search, you can specify whether the layout of the results component is also saved. Creating saved searches and saving the results components layout require that MDS is configured. If you set the query component's `saveResultsLayout` attribute to `always`, the results component layout will be saved. If `saveResultsLayout` is set to `never`, the layout is not saved.

```
<context-param>
   <description>Saving results layout</description>
       <param-name>oracle.adf.view.rich.query.SAVE_RESULTS_LAYOUT</param-name>
   <param-value>true</param-value>
</context-param>
```

If `saveResultsLayout` is not defined, saving layout defaults to the application-level property `oracle.adf.view.rich.query.SAVE_RESULTS_LAYOUT` in the `web.xml` file. The default value of `oracle.adf.view.rich.query.SAVE_RESULTS_LAYOUT` is `true`.

If the user made changes to the layout of a saved search and proceeds without saving, a warning message appears to remind the user to save, otherwise the changes will be lost. In addition, if the user adds or deletes search fields and proceeds without saving, a warning message also appears.

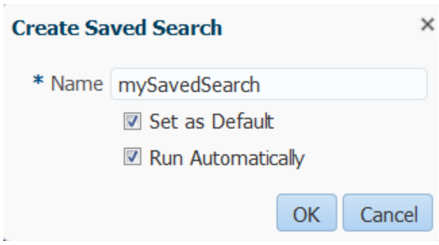
Table 36-2 shows the `saveResultsLayout` and `SAVE_RESULTS_LAYOUT` values and their resultant action. Note that MDS must be configured to save layout.

Table 36-2 Save Results Component Layout Attributes

web.xml SAVE_RESULT_LAYOUT	Query Component saveResultsLayout	Resultant action
true	always	Saves layout
true	never	Not saved
true	not defined	Saves layout
false	always	Saves layout
false	never	Not saved
false	not defined	Not saved

Figure 36-13 shows the Create Saved Search dialog with an entry field for the name of the saved search and checkbox selections for **Set as Default** and **Run Automatically**.

Figure 36-13 Runtime Saved Search Dialog Window



Create Saved Search

* Name: mySavedSearch

☑ Set as Default

☑ Run Automatically

Table 36-1 lists the possible scenarios for creators of saved searches, the method of their creation, and their availability.

Table 36-3 Design Time and Runtime Saved Searches

Creator	Created at Design time as View Criteria	Created at Runtime with the Save Button
Developer	Developer-created saved searches (system searches) are created during application development and typically are a part of the software release. They are created at design time as view criteria. They are usually available to all users of the application and appear in the lower part of the **Saved Search** dropdown list.	
Administrator		Administrator-created saved searches are created during predeployment by site administrators. They are created before the site is made available to the general end users. Administrators can create saved searches (or view criteria) using the JDeveloper design time when they are logged in with the appropriate role. These saved searches (or view criteria) appear in the lower part of the **Saved Search** dropdown list.
End User		End-user saved searches are created at runtime using the query form **Save** button. They are available only to the user who created them. End-user saved searches appear in the top part of the **Saved Search** dropdown list.

You can use the **runQueryAutomatically** property as listed in the Property Inspector to specify whether a saved search will run the query automatically. Setting the property to **allSavedSearches** means all system and user-created saved searches will run automatically when the query component is initially loaded, whenever the saved search has changed, or the reset button is pressed. If you set the **runQueryAutomatically** property to **searchDependent**, you can specify at design time whether that specific query will run automatically. **searchDependent** is the default value.

For new user-created saved searches with **runQueryAutomatically** property set to **searchDependent**, the Create Saved Search dialog, as shown in Figure 36-13, will have the **Run Automatically** option set as the default. For new user-created saved searches with **runQueryAutomatically** property set to **allSavedSearches**, the Create Saved Search dialog will not display the **Run Automatically** option, but it will be set implicitly.

End users can manage their saved searches by using the Personalize function in the **Saved Search** dropdown list to bring up the Personalize Saved Searches dialog, as shown in Figure 36-14.

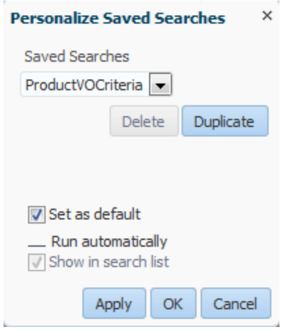
End users can use the Personalize function to:

- Update a saved search.

  Users can add and/or remove search criterion, update operators, update values, modify conjunctions and save it using the existing saved search name (to override an existing saved search).
- Delete a saved search (including the currently active Saved Search)

  Users can delete the default saved search, in which case the next USER level saved search will become the default.
- Duplicate an existing saved search
- Set a saved search as the default
- Set a saved search to run automatically
- Set the saved search to show or hide from the **Saved Search** dropdown list

Figure 36-14 Personalize Saved Searches Dialog



Personalize Saved Searches

Saved Searches: ProductVOCriteria

☑ Set as default
Run automatically
☑ Show in search list

> **Note:**
>
> If in you are changing the value of a view criteria item programmatically, you must invoke the `ViewCriteria.saveState()` method to prevent the searchRegion binding from resetting the value of the view criteria item to the value that was specified at design time.

You create a query search form by dropping a named view criteria item from the Data Controls panel onto a page. You have a choice of dropping only a search panel, dropping a search panel with a results table, or dropping a search panel with a tree table.

If you choose to drop the search panel with a table, you can select the filtering option in the dialog to turn the table into a filtered table.

Typically, you would drop a query search panel with the results table or tree table. JDeveloper will automatically create and associate a results table or tree table with the query panel.

If you drop a query panel by itself and want a results component or if you already have an existing component for displaying the results, you will need to match the query panel's `ResultsComponentId` with the results component's `Id`.

> **Note:**
>
> When you drop a named view criteria onto a page, that view criteria will be the basis for the initial search form. All other view criteria defined against that data collection will also appear in the **Saved Search** dropdown list. Users can then select any of the view criteria search forms, and also any end-user created saved searches.

How to Create a Query Search Form with a Results Table or Tree Table

You create a search form by dragging and dropping a view criteria from the Data Controls panel onto the page. You have the option of having a results table or only the query panel.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create a query search form. For more information, see Creating Query Search Forms.

You may also find it useful to understand functionality that can be used with query search forms. For more information, see Additional Functionality for Search Forms.

You will need to complete these tasks:

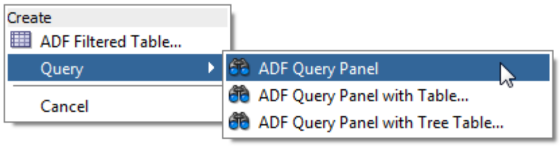
1. Create a view object. For more information about view objects, see Populating View Object Rows from a Single Database Table, and Working with Multiple Tables in Join Query Results.
2. Create view criteria if you intend to create search forms based on view criteria. You can also use the default implicit view criteria, which would include all queriable attributes in the collection. For more information about view criteria, see Working with Named View Criteria.
3. Set the search form default properties. For more information about setting the default state of the search form, see How to Set Search Form Properties on the View Criteria. For information on how to create view criteria, see Working with Named View Criteria.

To create a query search form with a results table or tree table:

1. From the Data Controls panel, select the data collection and expand the **Named Criteria** node to display a list of named view criteria.
2. Drag the named view criteria item and drop it onto the page or onto the Structure window.

3. From the context menu, choose **Create > Query > ADF Query Panel with Table** or **Create > Query > ADF Query Panel with Tree Table**, as shown in Figure 36-15.

Figure 36-15 Data Controls Panel with Query Context Menu


Create
ADF Filtered Table...
Query
Cancel
ADF Query Panel
ADF Query Panel with Table...
ADF Query Panel with Tree Table...

4. In the Edit Table Columns dialog, you can rearrange any column and select table options. If you choose the filtering option, the table will be a filtered table.

After you have created the form, you may want to set some of its properties or add custom functions. For more information on how to do this, see Setting Up Search Form Properties.

How to Create a Query Search Form and Add a Results Component Later

You create a search form by dragging and dropping a view criteria from the Data Controls panel onto the page. You have the option of having a results table or only the query panel.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create a query search form. For more information, see Creating Query Search Forms.

You may also find it useful to understand functionality that can be used with query search forms. For more information, see Additional Functionality for Search Forms.

You will need to complete these tasks:

1. Create a view object.
2. Create view criteria if you intend to create search forms based on view criteria. You can also use the default implicit view criteria, which would include all queriable attributes in the collection.
3. Set the search form default properties. For more information about setting the default state of the search form, see How to Set Search Form Properties on the View Criteria. For information on how to create view criteria, see Working with Named View Criteria.

To create a query search form and add a results component in a separate step:

1. From the Data Controls panel, select the data collection and expand the **Named Criteria** node to display a list of named view criteria.
2. Drag the named view criteria item and drop it onto the page or onto the Structure window.
3. From the context menu, choose **Create > Query > ADF Query Panel**, as shown in Figure 36-15.

4. If you do not already have a results component, then drop the data collection associated with the view criteria as a component.
5. In the Properties window for the table, copy the value of the **Id** field.
6. In the Properties window for the query panel, paste the value of the table's ID into the query's **ResultsComponentId** field.

After you have created the search form, you may want to set some of its properties or add custom functions. See Setting Up Search Form Properties, for more information.

How to Persist Saved Searches into MDS

If you want saved searches to be persisted to MDS, you need to define the `/persdef` namespace in the `adf-config.xml` file. In addition, you need to perform the regular MDS configuration, such as specifying `metadatapath`. The following example shows an `adf-config.xml` file with the `/persdef` namespace defined.

```
<persistence-config>
   <metadata-namespaces>
        <namespace path="/persdef" metadata-store-usage="mdsstore"/>
   </metadata-namespaces>
   <metadata-store-usages>
        <metadata-store-usage id="mdsstore" deploy-target="true"
                 default-cust-store="true">
        </metadata-store-usage>
   </metadata-store-usages>
</persistence-config>
```

In order for the added saved searches to be available the next time the user logs in, `cust-config` needs to be defined as part of the MDS configuration. For more information about setting `cust-config` and MDS, see How to Create Customization Classes.

If you are also saving the layout of the results component, the application must have the ADF PageFlow Runtime and ADF Controller Runtime libraries installed. Set the project's technology scope to include **ADF Page Flow** or automatically include these libraries by using the ADF Fusion Web Application application template.

What You May Need to Know About Named Bind Variables in Search Forms

Instead of specifying a literal operand in a view criteria to be used in a search form, you have the option of specifying a named bind variable. The named bind variable performs like a parameter whose value can change at runtime without the need to change the SQL statement. It must be defined in the view object before it can be used in a view criteria.

If you specify a literal operand in the view criteria and leave the value blank, it will not appear in the SQL preview. When the view criteria is applied as a search form at runtime, that attribute is rendered as a blank input search field. If a value is specified for the literal operand, then the SQL preview will generate a SQL clause for it. When the view criteria is applied as a search form at runtime, the SQL statement is not automatically applied even though the value specified in the view criteria appears in the input search field. The SQL statement won't get applied until the user clicks **Search** (or when auto-execute is set to true).

If a named bind variable is used in the query defined in the view criteria, the SQL preview will display the WHERE clause with the bind variable. When the view criteria is applied as a search form at runtime, the bind variable will be rendered with a prompt and an input search field based on the name of the attribute (not on the name of the bind variable). The named bind variable input field may be NULL, it may contain the default value, or it may contain a value that has loaded from previous processing, such as from another page. The user can enter values for the named bind variable as in any other search criteria. When the search is executed, the value of the named bind variable will be evaluated with the other criteria, as defined by the SQL query statement.

> **Note:**
>
> To avoid the possibility of SQL injection for search forms with literal values, you can force all input search fields to use bind variables by setting the useBindVarsForViewCriteriaLiterals property to true in the adf-config.xml file. The default value is true. When set to true, each input search field in the application will display the default value defined by the literal operand based on the query component's associated view criteria.
>
> For more information about the adf-config.xml file, see adf-config.xml.
>
> You can also use the overview editor to edit this property. In the overview editor for adf-config.xml, click the **Model** tab and check or uncheck the **Use Bind Variable for ViewCriteria literals** checkbox.

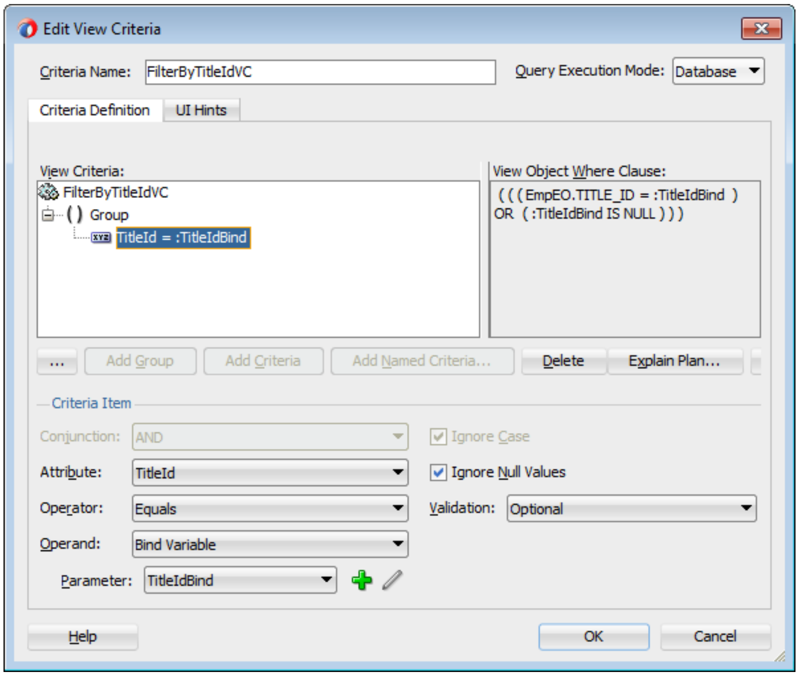
If the bind variable is used more than once in the same view criteria, each occurrence of the bind variable will be rendered as an individual input field. Because there is only one bind variable backing all the input fields, the value of all the fields will be synchronized. For instance, if you specify a view criteria that uses the same bind variable three times, then three input fields will be rendered. When the user enters a value into one input field, the other two input fields will have the same value. Using bind variables in this way eliminates the need for the user to enter the same value multiple times.

Another use of the bind variable is to pass a value from the base row into a search for an LOV search form.

Bind variables can also be used to pass parameter values from one page to another, such as when a customer ID is passed to another page for more detail processing. And, depending on the construct of the SQL statement, using the named bind variable may speed up the query because it may lessen the need to prepare a new statement, which means that the database does not need to reparse. For more information about creating and using named bind variables, see Working with Bind Variables.

For example, in the listCustomerAddresses view criteria, the WHERE clause checks to see whether the AssociatedOwnerId is the same as the value of the paramCustomerId named bind variable. This view criteria is in the AddressesLookupVO view object. The listCustomerAddresses view criteria as defined in the Edit View Criteria dialog is shown in Figure 36-16.

Figure 36-16 View Criteria with Named Bind Variable



Criteria Name: FilterByTitleIdVC

Query Execution Mode: Database

View Criteria:
FilterByTitleIdVC
( ) Group
TitleId = :TitleIdBind

View Object Where Clause:
(((EmpEO.TITLE_ID = :TitleIdBind)
OR (:TitleIdBind IS NULL)))

Conjunction: AND

☑ Ignore Case

Attribute: TitleId

☑ Ignore Null Values

Operator: Equals

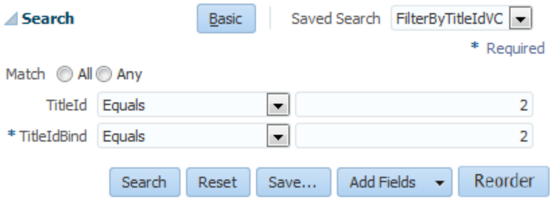
Validation: Optional

Operand: Bind Variable

Parameter: TitleIdBind

A query search form for a view criteria with a named bind variable will render with a search field for the variable using the `inputText` component. Figure 36-17 shows the **AssociatedOwnerId** search field displayed as an `inputText` component.

Figure 36-17 Query Search Form with Named Bind Variable



Saved Search FilterByTitleIdVC

* Required

Match ○ All ○ Any

TitleId Equals 2

* TitleIdBind Equals 2

What You May Need to Know about Default Search Binding Behavior

A **search binding** provides the model-driven behavior for a search form at runtime. The search binding is related to a particular iterator and the view criteria on that iterator, whose runtime defaults are specified by the Binds and Criteria properties of the search binding, respectively. The behavior of the search binding depends on the **Query Automatically** control hint. For more information about creating named view criteria, see Working with Named View Criteria.

The first time a search form is rendered in a task flow, the search binding's runtime behavior, also known as the initial AutoQuery-or-ClearRowSet behavior, is as follows:

- If **Query Automatically** is set to `true`: The search binding automatically executes the iterator when the search binding is initialized in a task flow. The user will be presented with the results of the search based on the view criteria.
- If **Query Automatically** is set to `false`: The search binding clears the row set related to its iterator. The user will be presented with an empty search results component, which allows the user to enter search criteria and click the **Search** button to execute the search.

The **Query Automatically** control hint is available declaratively only for named view criteria. For the All Queriable Attributes view criteria, you can use the `setProperty()` method on the view criteria API interface to configure the hint at runtime. You can also use this method to set the hint on named view criteria. In the method, set the `ViewCriteriaHints.CRITERIA_AUTO_EXECUTE` property to `true`.

If the user changes the search form by selecting a different view criteria from the dropdown **Saved Search** list, the runtime behavior is as follows:

- If **Query Automatically** is set to `true`: The search binding automatically executes the iterator after applying the new view criteria. The user will be presented with the results of the newly selected saved search.
- If **Query Automatically** is set to `false`: The search binding applies the new view criteria only, leaving any existing search results intact. The user is presented with the results of the previous search.

The search binding's `queryPerformed` property evaluates to `true` if the search binding has performed the query, either automatically or by the user clicking the **Search** button. Clicking the **Reset** button in the search form resets the view criteria and sets the `queryPerformed` property to `false`.

The search binding offers two properties that allow you to customize its default behavior: `TrackQueryPerformed` and `InitialQueryOverridden`. You can set the `TrackQueryPerformed` and the `InitialQueryOverridden` properties in the Properties window.

A search binding's `TrackQueryPerformed` property controls whether it manages its `queryPerformed` property during runtime at the page flow level or at the individual page or page fragment level. The valid values for `TrackQueryPerformed` are `pageFlow` and `page`, and the default is `pageFlow`. The search binding's behavior is as follows:

- If `TrackQueryPerformed` is set to `pageFlow`, then `queryPerformed` is initialized once per page flow and tracked at the page flow level. The user can navigate away from the search form page, return to the page within the life of the page flow, and the value of the `queryPerformed` flag will remain the same.
- If `TrackQueryPerformed` is set to `page`, then `queryPerformed` is initialized each time the user navigates to the page or page fragment. This happens when the page is navigated to for the first time and when the page is returned from another page within the page flow.

When `TrackQueryPerformed` of a search binding is set to `pageFlow`, its initial AutoQuery-or-ClearRowSet behavior is performed once during the page flow. In contrast, when `TrackQueryPerformed` is set to `page`, the initial AutoQuery-or-ClearRowSet behavior is performed each time the user visits the page or page fragment.

A search binding's `InitialQueryOverridden` property controls whether it should suppress its initial AutoQuery-or-ClearRowSet behavior the first time the search binding is used in a page flow. If `InitialQueryOverridden` is `true`, then all valid values, including `true`, `false`, or a boolean-valued EL expression, are suppressed. The default value is `false`.

When you set the `InitialQueryOverridden` property to `true`, you are responsible for writing custom application logic to execute the query. Typically, the query should execute after the code applies some view criteria, sets some bind variables, or performs some other programmatic query setup. If your custom code fails to execute the query as expected with `InitialQueryOverridden` set to `true`, unless **Query Automatically** is set to `false`, the framework will still implicitly execute your query the first time during a user session in which there is an iterator binding reference. This occurs because when **Query Automatically** is not set to `false`, the iterator binding's default `executeQueryIfNeeded` behavior takes effect and executes the query.

When `InitialQueryOverridden` evaluates to `true` or boolean `true`, then the initial AutoQuery-or-ClearRowSet behavior is suppressed the first time the search binding is used in a page flow. If `TrackQueryPerformed` is set to `pageFlow`, then only the initial AutoQuery-or-ClearRowsSet behavior (that would have occurred for this search binding) is suppressed.

In contrast, if a search binding's `TrackQueryPerformed` property is set to `page`, then only the initial AutoQuery-or-ClearRowSet behavior is suppressed. Subsequent initial AutoQuery-or-ClearRowSet behaviors that occur due to the user's navigating back to the same page (or page fragment) are not affected by the `InitialQueryOverridden` property.

If you want to avoid the search binding's performing any initial AutoQuery-or-ClearRowSet behavior, then leave the `TrackQueryPerformed` set to `pageFlow` and set `InitialQueryOverridden` to `true`.

You should not use the `RefreshCondition` property of an iterator to reference the `queryPerformed` property of a search binding. Doing so will inadvertently prevent new rows from being created in that iterator's row set until after the search binding's query has been performed.

What You May Need to Know About Dependent Criterion

There are situations when one search criteria is dependent on the value of another criteria. For example, a bug database has a search form with a `Component` and a `Subomponent` search criteria, both defined as LOVs. When the user selects a value from the Component search criteria, that value must be submitted to the model so that `Subomponent` can be filtered and populated with the appropriate search list for the user to choose from.

If you have a search criteria that has dependents, you must set the root and dependent criteria's underlying view attribute's **Auto Submit** control hint to `true`. You can create a custom listener to trigger a partial submit to update the model and refresh the search panel when the query component detects a value change for the root criteria. You need to create a custom listener because the standard `QueryOperationListener` does not handle this event type. You can create a custom `QueryOperationListener` class using the `QueryOperationListener` interface. You then register this class by implementing it in a managed bean or by directly setting it in the JSF page.

What Happens When You Create a Query Form

When you drop a query search form onto a page, JDeveloper creates an `af:query` tag on the page. If you drop a query with table or tree table, then an `af:table` tag or `af:treeTable` tag will follow the `af:query` tag.

Under the `af:query` tag are several attributes that define the query properties. They include:

- The `id` attribute, which uniquely identifies the query.
- The `resultsComponentId` attribute, which identifies the component that will display the results of the query. Typically, this will be the table or tree table that was dropped onto the page together with the query. You can change this value to be the `id` of a different results component. For more information, see How to Create a Query Search Form and Add a Results Component Later.

In the page definition file, JDeveloper creates an iterator and a `searchRegion` entry in the `executables` section. The following example shows the sample code for a page definition file that was created when the `CustomerViewCriteria` from the `Customers` collection of the `BackOfficeAppModuleDataControl` is dropped onto the page.

In the page definition file `executable` section:

- The iterator `binds` property is set to the name of the data collection. In the example, the value is set to `Customers.`
- The iterator `id` property is set to a data collection iterator. In the example, the value is set to `CustomersIterator`
- If there is more than one view criteria defined for that data collection, the `searchRegion Criteria` property is set to the name of the view criteria that was dropped. In the example, the value is set to `CustomerViewCriteria`. If there is only one view criteria defined, then there will not be a `Criteria` property.
- The `searchRegion Binds` property is set to the same value as the `iterator id` property. In the example, the value is set to `CustomersIterator`
- The `searchRegion id` property is set to the name of the view criteria concatenated with `Query`. In the example, the value is set to `CustomerViewCriteria.`

If the query was dropped onto the page with a table or tree, then in the page definition file `bindings` section, a tree element is added with the `Iterbinding` property set to the search iterator. In this example, the value is set to `CustomersIterator`. This should be the same iterator defined in the `executable` section.

```
<?xml version="1.0" encoding="UTF-8" ?>
<pageDefinition xmlns="http://xmlns.oracle.com/adfm/uimodel"
version="12.1.2.66.41" id="searchPageDef"
                Package="oracle.summit.view.pageDefs">
 <parameters/>
 <executables>
    <variableIterator id="variables"/>
    <iterator Binds="Customers" RangeSize="25"
          DataControl="BackOfficeAppModuleDataControl" id="CustomersIterator"/>
          <searchRegion Criteria="CustomerViewCriteria"
                Customizer="oracle.jbo.uicli.binding.JUSearchBindingCustomizer"
                Binds="CustomersIterator" id="CustomerViewCriteriaQuery"/>
 </executables>
 <bindings>
    <tree IterBinding="CustomersIterator" id="Customers">
      <nodeDefinition DefName="oracle.summit.model.views.CustomerVO"
           Name="Customers0">
        <AttrNames>
          <Item Value="Id"/>
          <Item Value="Name"/>
          <Item Value="City"/>
          <Item Value="State"/>
          <Item Value="CountryId"/>
          <Item Value="LastName"/>
        </AttrNames>
      </nodeDefinition>
    </tree>
  </bindings>
</pageDefinition>
```

What Happens at Runtime: Search Forms

At runtime, the search form displays as a search panel on the page. The search panel will display in either basic mode or advanced mode, depending on the mode control hint when its corresponding view criteria was created. The **Saved Search** dropdown list will contain all the view criteria that are enabled (**Show in List** control hint enabled). The **Match All/Any** conjunction radio button may be enabled.

A search criteria field will be rendered for each search criteria defined in the view criteria. If the **Default List Type** control hint in the view object has been declared as an LOV or a selection list component, the search criteria field component is as shown in Table 36-1.

After the user enters the search criteria and clicks **Search**, a query against the view criteria is executed and the results are displayed in the associated table, tree table, or component.

When the users changes the mode of the query component from advanced to basic, the operators displayed by the query component may change depending upon how you have configured the view criteria control hint **Show Operators** and depending on whether the view criteria is a predefined, named view criteria or the automatically defined implicit criteria. In the case of named view criteria, the query component will only preserve the user's selection between modes when you set **Show Operators** to **Always** to allow the full list of operators in both modes. However, if you set **Show Operators** to **In Advanced Mode** to only show the full list of operators in advanced mode, then after the user makes their selections and switches to basic mode, the operators will default to the ones defined by the view criteria and for this reason the user's selections may appear to change. In the case of an implicit view criteria, because the implicit criteria does not have predefined operators, the query component retains the operators selected by the user when user switches between advanced mode and basic mode.

Setting Up Search Form Properties

For a query search form in Oracle ADF, you can set search form properties on view criteria or on the query component. You can set the properties either during the creation of view criteria or after you have dropped the query search form onto a page.

A query search form is based on a view criteria defined in a view object. When you create the view criteria, you also specify some of the search form properties. Later on, when you drop the named criteria onto the page to create a query component, you can specify other search form properties.

Search form properties that can be set on the view object include grouping related and dependent fields together in the query panel. For information on setting hints in the view object, see How to Define UI Category Hints.

Search form properties that can be set when the view criteria is being created include:

- Default mode in basic or advanced mode
- Automatic query execution when the page loads
- Rendering of the search criteria field
- Enabling multiple selections for attributes defined as an LOV

Search form properties that can be set after the query component has been added to the JSF page include:

- `id` of the results table or results component
- Show or hide of the basic/advanced button
- Position of the mode button
- Default, simple, or compact mode for display

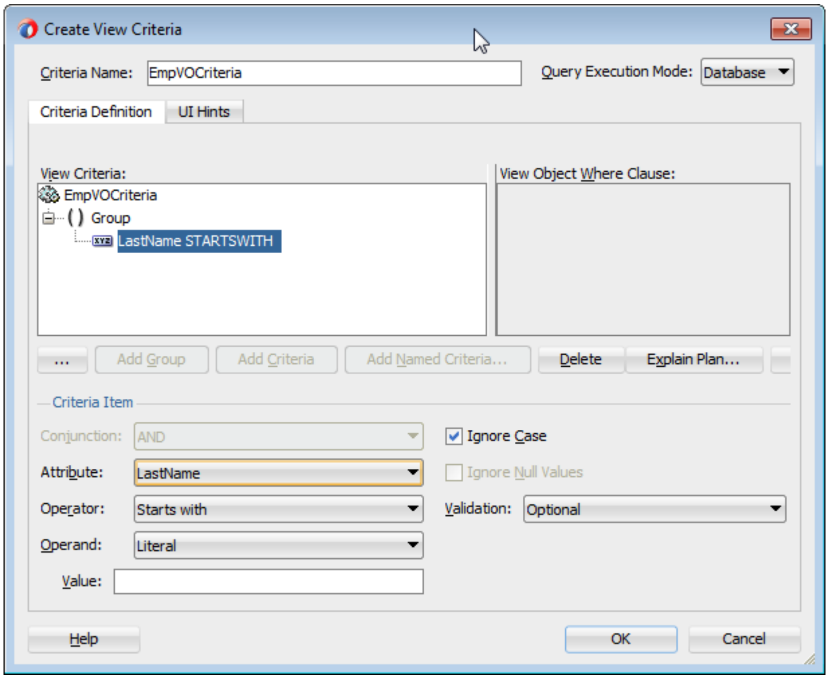
Search form attribute properties that can be set when the view object is being created include:

- `timezone` control hint for a `timestamp` attribute

How to Set Search Form Properties on the View Criteria

When you are creating a view criteria, you can declaratively set the initial state of several properties. Figure 36-18 shows the Edit View Criteria dialog for setting default options. For more information about view criteria, see Working with Named View Criteria.

Figure 36-18   Create View Criteria Dialog



Criteria Name: EmpVOCriteria

Query Execution Mode: Database

Conjunction: AND

☑ Ignore Case

Attribute: LastName

☐ Ignore Null Values

Operator: Starts with

Validation: Optional

Operand: Literal

Value:

You must select the default mode of the query search form as either basic or advanced. The default is basic.

You also must declare whether each individual search criteria field will be available only in basic mode, only in advanced mode, available in both modes, or never displayed. If a search criteria field is declared only for basic mode, it will not appear when the user switches to advanced mode, and the reverse is true. If the field is declared for all, then it will appear in all modes. The default for search criteria field rendering is all modes.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create a query search form. See Setting Up Search Form Properties.

You may also find it useful to understand functionality that can be used with query search forms. See Additional Functionality for Search Forms.

You will need to complete this task:

> Create a view criteria for the query search form, as described in Working with Named View Criteria.

To set the default mode and search criteria field display option:

1. In the Applications window, double-click the view object that contains the view criteria you created for the query search form.
2. In the overview editor for the view object, click the View Criteria navigation tab.
3. In the View Criteria page, select the view criteria from the list and click **Edit**.
4. In the Edit View Criteria dialog, click the **Item UI Hints** tab on the Criteria Definition page.

5. In the **Rendered Mode** dropdown list, select **All**, **Basic**, **Advanced**, or **Never**.
6. In the Edit View Criteria dialog, click the **Criteria UI Hints** tab.
7. From the **Search Region Mode** dropdown list, select either **Basic** or **Advanced**.
8. In the **Criteria Item UI Hints** section, select the criteria item you want to set.
9. In the **Show Operators** dropdown list, select **Always**, **In Basic Mode**, **Never**, **In Advanced Mode**, or **Never** to specify whether or not the user may change the operators displayed by the query component backed by this view criteria.

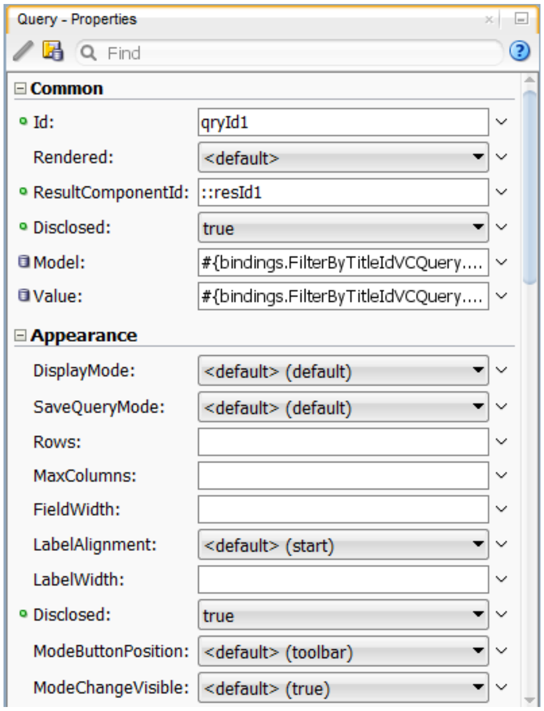
   The operators displayed by the query component when the users changes the mode of the query component from advanced to basic may change depending upon how you have configured **Show Operators**. The query component will only preserve the user's selection between modes when you set **Show Operators** to **Always** to allow the full list of operators in both modes. However, if you set **Show Operators** to **In Advanced Mode** to only show the full list of operators in advanced mode, then after the user makes their selections and switches to basic mode, the operators will default to the ones defined by the view criteria and for this reason the user's selections may appear to change. Select **Never** when you want the view criteria to be executed using ONLY the operators it defines (where the full list of operators will not be exposed in either basic or advance modes).
10. Click **OK**.

How to Set Search Form Properties on the Query Component

After you have dropped the query search form onto a page, you can edit other form properties in the Properties window, as shown in Figure 36-19. Some of the common properties you may set are:

- Enabling or disabling the basic/advanced mode button
- Setting the ID of the query search form
- Setting the ID of the results component (for example, a results table)
- Selecting the default, simple, or compact mode for display
- Setting the `criterionFeatures` to `matchCaseDisplayed` to require all string-based search criterion to be case-sensitive or to `requiredDisplayed` to require all the criterion to be displayed.

Figure 36-19 Properties window for a Query Component



Query - Properties

Common

Id:	qryId1
Rendered:	<default>
ResultComponentId:	::resId1
Disclosed:	true
Model:	#{bindings.FilterByTitleIdVCQuery....
Value:	#{bindings.FilterByTitleIdVCQuery....

Appearance

DisplayMode:	<default> (default)
SaveQueryMode:	<default> (default)
Rows:	
MaxColumns:	
FieldWidth:	
LabelAlignment:	<default> (start)
LabelWidth:	
Disclosed:	true
ModeButtonPosition:	<default> (toolbar)
ModeChangeVisible:	<default> (true)

One common option is to show or hide the basic/advanced button. For more information on some of the other properties, see the How to Add the Query Component section in *Developing Web User Interfaces with Oracle ADF Faces*.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create a query search form. For more information, see Setting Up Search Form Properties.

You may also find it useful to understand functionality that can be used with query search forms. For more information, see Additional Functionality for Search Forms.

To enable or hide the basic/advanced button in the query form:

1. In the Structure window, double-click **af:query**.
2. In the Properties window, click the **Appearance** tab.
3. To enable the basic/advanced mode button, select **true** from the **ModeChangeVisible** field. To hide the basic/advance mode button, select **false** from the **ModeChangeVisible** field.

How to Set Timezone Control Hint for Timestamp Attribute

If a query includes a `Date` attribute, you can set its `timezone` using control hints on the view object or entity object where the attribute is defined. For instance, you can set the `Hiredate` of an employee to have the correct `timezone` for the employee's local office.

Before you begin:

Create the desired view objects as described in How to Create an Entity-Based View Object, and How to Create a Custom SQL Mode View Object.

To customize view object timestamp attribute with control hints:

1. In the Applications window, double-click the view object.
2. In the overview editor, click the **Attributes** navigation tab and select the `Date` attribute that you want to customize with control hints.
3. Click the **UI Hints** tab to expose the **Timezone** property.
4. Select a time zone from the **Timezone** dropdown list.

   The time zone values are listed as Universal Time Coordinated (UTC) +/- hours.

How to Create Custom Operators

You can create custom operators for each view criteria item by adding code to the view object XML file. For example, you can create a new operator called `more than a year`, which operates on a date attribute (greater than 365 days from the current date).

You can add custom operators for a view object attribute by adding code for that attribute in the view object XML file.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create a query search form. For more information, see Setting Up Search Form Properties.

You may also find it useful to understand functionality that can be used with query search forms. For more information, see Additional Functionality for Search Forms.

To add a custom operator:

1. In the Applications window, select the view object for the view criteria in which you want to add a custom operator.
2. In the overview editor, click the **Source** tab.
3. In the XML editor, locate the code for the view criteria attribute, and add the `CompOper` code statements after the last item within the `ViewCriteriaItem` group. Example 36-1 shows code for the view criteria: the `CompOper` code statements appear in bold.

   The `CompOper` properties are:

   - `Name`: Specify an `id` for the operation.
   - `ToDo`: Set to `1` to add this custom operator. Set to `-1` to remove an operator.
   - `OperDescStrCode`: Specify the `id` used in the message bundle to map to the description string, as described in Step 4.
   - `Oper`: Set to a value that will be used programmatically to denote this operation in the SQL statement. In Example 36-1, `Oper` was set to `>Y` to denote greater than 1 year.
   - `MinCardinality`: If there is an input range, set this property to the minimum value for the range. For example, if the range is months in a year, this value should be set to `1`. If there is no range, set it to `0`.

- MaxCardinality: If there is an input range, set this property to the maximum value for the range. For example, if the range is months in a year, this value should be set to 12. If there is no range, set it to 0.
- TransientExpression: Set the expression to perform the custom operator function. In Example 36-1, the expression is ![CDATA[return " > SYSDATE -365"]], which returns the string " > SYSDATE -365".

4. Open the message bundle file for the view object and add an entry for the custom operator, using the OperDescStrCode identifier defined in the view object XML in Step 3.

   Example 36-2 shows the message bundle code for the LastUpdateDate custom operator described in Example 36-1.

Example 36-1   Adding the Custom Operator Code to the View Object XML

```
<ViewCriteriaRow
  Name="vcrow50"
  UpperColumns="1">
   <ViewCriteriaItem
     Name="LastUpdateDate"
     ViewAttribute="LastUpdateDate"
     Operator="="
     Conjunction="AND"
     Required="Optional">
        <CompOper
            Name="LastUpdateDate"
            ToDo="1"
            OperDescStrCode="LastUpdateDate_custOp_grt_year"
            Oper=">Y"
            MinCardinality="0"
            MaxCardinality="0" >
            <TransientExpression><![CDATA[ return " < SYSDATE - 365"
                ]]></TransientExpression>
        </CompOper>
   </ViewCriteriaItem>
```

Example 36-2   Adding the Custom Operator Entry for LastUpdateDate to the Message Bundle

```
public class AvailLangImplMsgBundle extends JboResourceBundle {
     static final Object[][] sMessageStrings =
  {
       { "LastUpdateDate_custOp_grt_year", "more than a year old"
   },
```

How to Remove Standard Operators

You can remove standard operators from a view criteria item. For example, you can remove the standard operator before from the list.

For a list of standard operators, see the Using Query Components chapter of *Developing Web User Interfaces with Oracle ADF Faces*.

You can remove standard operators for a view object attribute by adding code for that attribute in the view object XML file.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create a query search form. For more information, see Setting Up Search Form Properties.

You may also find it useful to understand functionality that can be used with query search forms. For more information, see Additional Functionality for Search Forms.

To remove a standard operator:

1. In the Applications window, select the view object for the view criteria in which you want to remove a standard operator.

   > **Note:**
   >
   > Before you attempt to remove the standard operator, make sure you do not remove the default operator for that view criteria item.

2. In the overview editor, click the **Source** tab.
3. In the XML editor, locate the code for the view criteria attribute, and add the `CompOper` code statements after the last item within the `ViewCriteriaItem` group.

   In the following example, the `CompOper` code statements appear in bold. The code in this example removes the `BEFORE` operator from the list of operators for the `LastUpdateDate` attribute.

   ```
   <ViewCriteriaRow
     Name="vcrow50"
     UpperColumns="1">
      <ViewCriteriaItem
        Name="LastUpdateDate"
        ViewAttribute="LastUpdateDate"
        Operator="="
        Conjunction="AND"
        Required="Optional">
           <CompOper
               Name="LastUpdateDate"
               ToDo="-1"
               Oper="BEFORE"
           </CompOper>
      </ViewCriteriaItem>
   ```

   The `CompOper` properties are:

   - `Name`: Specify an `id` for the operation.
   - `ToDo`: Set to `-1` to remove an operator. Set to `1` to add an operator.

     Do not set `ToDo` to `-2`, which would remove all the operators.
   - `Oper`: Set to the standard operator you want to remove from the list.

How to Set Uniform Operator Field Width

If you want a uniform width for all the operator dropdown fields in the search panel, you can set the following skinning property:

```
af|query {
 -tr-operator-size: constant;

}
```

Having a uniform operator field width would vertically align the operator and search boxes for all the search fields, which results in a more organized appearance.

For more information about skins, see the Customizing the Appearance Using Styles and Skins chapter in *Developing Web User Interfaces with Oracle ADF Faces*.

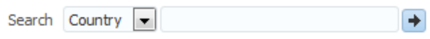
Creating Quick Query Search Forms

In Oracle ADF, quick query search form is a simplified form with only one search criteria that creates a dropdown list of all searchable attributes in the associated view object. You can use different methods to create quick query search forms and can also set quick query layout format.

A quick query search form is intended to be used in situations where a single search will suffice or as a starting point to evolve into a full query search. Both the query and quick query search forms are ADF Faces components. A quick query search form has one search criteria field with a dropdown list of the available searchable attributes from the associated data collection. Typically, the searchable attributes are all the attributes in the associated view object. You can exclude attributes by setting the attribute's **Display Hint** property in the Control Hints page of the Edit Attribute dialog to **Hide**. The user can search against the selected attribute or search against all the displayed attributes. The search criteria field type will automatically match the type of its corresponding attribute. An **Advanced** link built into the form offers you the option to create a managed bean to control switching from quick query to advanced mode query search form. For more information, see the Using Query Components chapter in *Developing Web User Interfaces with Oracle ADF Faces*.

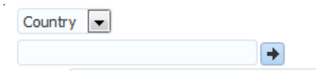
You can configure the form to have a horizontal layout, as shown in Figure 36-20.

Figure 36-20 Quick Query Search Form in Horizontal Layout



You can also choose a vertical layout, as shown in Figure 36-21.

Figure 36-21 Quick Query Search Form in Vertical Layout



You can use quick query search forms to let users search on a single attribute of a collection. Quick query search form layout can be either horizontal or vertical. Because they occupy only a small area, quick query search forms can be placed in different areas of a page. You can create a managed bean to enable users to switch from a quick query to a full query search. For more information about switching from quick query to query using a managed bean, see the Using Query Components chapter in *Developing Web User Interfaces with Oracle ADF Faces*.

If you drop a quick query panel with a results table or tree, JDeveloper will automatically create the results table, as described in How to Create a Quick Query Search Form with a Results Table or Tree Table. If you drop a quick query panel by itself and subsequently want a results table or component or if you already have one, you will need to match the quick query `Id` with the results component's `partialTrigger` value, as described in How to Create a Quick Query Search Form and Add a Results Component Later.

> **Note:**
>
> A quick query search creates a dropdown list of all searchable attributes defined in the underlying view object. If you want to show only a subset of those attributes, you can set the attribute's **Display** control hint to **Hide** for those attributes you want to exclude. For more information about setting control hints on view objects, see Defining SQL Queries Using View Objects.

How to Create a Quick Query Search Form with a Results Table or Tree Table

You can create quick query searches using the full set of searchable attributes and simultaneously add a table or tree table as the results component.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create a query search form. For more information, see Creating Quick Query Search Forms.

You may also find it useful to understand functionality that can be used with query search forms. For more information, see Additional Functionality for Search Forms.

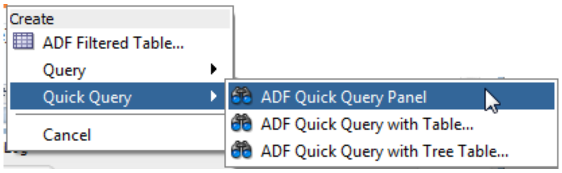
You will need to complete this task:

Create a view object to be the basis of the search form.

To create a quick query search form with a results table:

1. From the Data Controls panel, select the data collection and expand the **Named Criteria** node to display a list of named view criteria.
2. Drag the **All Queriable Attributes** item and drop it onto the page or onto the Structure window.
3. From the context menu, choose **Create > Quick Query > ADF Quick Query Panel with Table** or **Create > Quick Query > ADF Quick Query Panel with Tree Table**, as shown in Figure 36-22.
4. In the Edit Table Columns dialog, you can rearrange any column and select table options. If you choose the filtering option, the table will be a filtered table.
5. Click **OK**.

Figure 36-22 Data Controls Panel with Quick Query Context Menu



How to Create a Quick Query Search Form and Add a Results Component Later

You can create quick query searches using the full set of searchable attributes and add a table or tree table as the results component later.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create a query search form. For more information, see Creating Quick Query Search Forms.

You may also find it useful to understand functionality that can be used with query search forms. For more information, see Additional Functionality for Search Forms.

You will need to complete this task:

Create a view object to be the basis of the search form.

To create a quick query search form and add a results component in a separate step:

1. From the Data Controls panel, select the data collection and expand the **Named Criteria** node to display a list of named view criteria.
2. Drag the **All Queriable Attributes** item and drop it onto the page or onto the Structure window.
3. From the context menu, choose **Create > Quick Query > ADF Quick Query Panel**.
4. If you do not already have a results component, then drop the data collection associated with the view criteria as a component.
5. In the Properties window for the quick query panel, copy the value of the **Id** field.
6. In the Properties window for the results component (for example, a table), paste or enter the value into the **PartialTriggers** field.

How to Set the Quick Query Layout Format

The default layout of the form is horizontal. You can change the layout option using the Properties window.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create a query search form. For more information, see Creating Quick Query Search Forms.

You may also find it useful to understand functionality that can be used with query search forms. For more information, see Additional Functionality for Search Forms.

To set the layout:

1. In the Structure window, double-click **af:quickQuery**.
2. In the Properties window, on the Commons page, select the **Layout** field using the dropdown list to specify **default**, **horizontal**, or **vertical**.

What Happens When You Create a Quick Query Search Form

When you drop a quick query search form onto a page, JDeveloper creates an `af:quickQuery` tag. If you have dropped a quick query with table or tree table, then an `af:table` tag or `af:treeTable` tag is also added.

Under the `af:quickQuery` tag are several attributes and facets that define the quick query properties. Some of the tags are:

- The `id` attribute, which uniquely identifies the quick query. This value should be set to match the results table or component's `partialTriggers` value. JDeveloper will automatically assign these values when you drop a quick query with table or tree table. If you want to change to a different results component, see How to Create a Quick Query Search Form and Add a Results Component Later.
- The `layout` attribute, which specifies the quick query layout to be default, horizontal, or vertical.
- The `end` facet, which specifies the component to be used to display the **Advanced** link (that changes the mode from quick query to the query). For more information about creating this function, see the Using Query Components chapter of *Developing Web User Interfaces with Oracle ADF Faces*.

What Happens at Runtime: Quick Query

At runtime, the quick query search form displays a single search criteria field with a dropdown list of selectable search criteria items. If there is only one searchable criteria item, then the dropdown list box will not be rendered. An input component that is compatible with the selected search criteria type will be displayed, as shown in Table 36-4. For example, if the search criteria type is date, then `inputDate` will be rendered.

Table 36-4 Quick Query Search Criteria Field Components

Attribute Type	Rendered Component
`DATE`	`af:inputDate`
`VARCHAR`	`af:inputText`
`NUMBER`	`af:inputNumberSpinBox`

If the **Default List Type** control hint in the view object has been declared as an LOV or a selection list component, the search criteria field component appears as shown in Table 36-1 in List of Values (LOV) Input Fields.

In addition, a **Search** button is rendered to the right of the input field. If the end facet is specified, then any components in the end facet are displayed. By default, the end facet contains an **Advanced** link.

Creating Standalone Filtered Search Tables from Named View Criteria

In Oracle ADF, you can create a filtered table to perform searches without the search forms or as a results table of the search forms. You can use different methods to create a standalone filtered table and Query-by-Example searches.

A filtered table can be created standalone or as the results table of a query or quick query search form. Filtered table searches are based on Query-by-Example and use the QBE text or date input field formats. The input validators are turned off to allow for entering characters such as > and <= to modify the search criteria. For example, you can enter `>1500` as the search criteria for a number column. Wildcard characters may also be supported. If a column does not support QBE, the search criteria input field will not render for that column. The input field rendered will depend on the attribute's data type. For instance, a date column will have a date picker component rendered and a column for an attribute defined as a List of Values will have the corresponding List of Values component rendered.

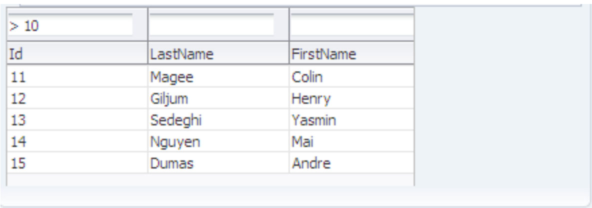
The filtered table search criteria input values are used to build the query `WHERE` clause with the `AND` operator. If the filtered table is associated with a query or quick query search panel, the composite search criteria values are also combined to create the `WHERE` clause.

Note:

If the filtered table is used with a query component in a search form and the search region is using an existing named criteria, the results of the query will be filtered by all the view criteria rows in an iterative manner. For example, a filtered table has two view criteria rows: `PersonId` and `DeptId`. The first view criteria row has `PersonId` > `1`. When the user enters > 100 in the filter field, then the second view criteria row `DeptId` is used to accept input to further filter the results. This process iterates through all the view criteria rows until the final query result is reached.

Figure 36-23 shows a filtered table. When the user enters a QBE search criteria, such as `>10` for the `Id` field, the query result is the `AND` of the query search criteria and the filtered table search criteria.

Figure 36-23 Filtered Table



> 10		
Id	LastName	FirstName
11	Magee	Colin
12	Giljum	Henry
13	Sedeghi	Yasmin
14	Nguyen	Mai
15	Dumas	Andre

Table 36-5 lists the acceptable QBE search operators that can be used to modify the search value.

Table 36-5 Query-by-Example Search Criteria Operators

Operator	Description
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
`AND`	And
`OR`	Or
=	Equal to
<>	Does not contain
`!=`	Does not contain

You use query search forms for complex searches, but you can also perform simple QBE searches using the filtered table. You can create a standalone ADF-filtered table without the associated search panel and perform searches using the QBE-style search criteria input fields. For information about filtered tables, see How to Create Filtered Table and Query-by-Example Searches.

You can set the QBE search criteria for each filterable column to be a case-sensitive or case-insensitive search using the `filterFeature` attribute of `af:column` in the `af:table` component. See the Enabling Filtering in Tables section of *Developing Web User Interfaces with Oracle ADF Faces*.

How to Create Filtered Table and Query-by-Example Searches

When creating a table, you can make almost any table a filtered table by selecting the filtering option if the option is enabled. There are three ways to create a standalone filtered table:

- You can drop a table onto a page from the Components window, bind it to a data collection, and set the filtering option. For more information, see the Using Query Components chapter of *Developing Web User Interfaces with Oracle ADF Faces*.
- You can create a filtered table by dragging and dropping a data collection onto a page and setting the filtering option. For more information, see How to Create a Basic Table.
- You can also create a filtered table or a read-only filtered table by dropping named criteria onto a page. You can use either the implicitly created named criteria **All Queriable Attributes** or any declaratively created named view criteria. The resulting filtered table will have a column for each searchable attribute and an input search field above each column.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create a query search form. For more information, see Creating Standalone Filtered Search Tables from Named View Criteria.

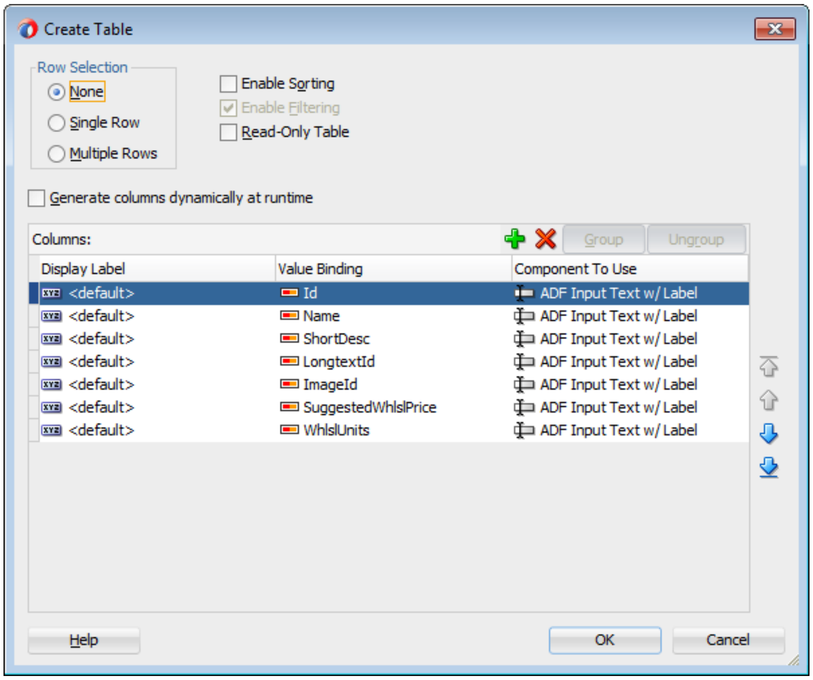
You may also find it useful to understand functionality that can be used with query search forms. For more information, see Additional Functionality for Search Forms.

To create a filtered table using named view criteria:

1. From the Data Controls panel, select the data collection and expand the **Named Criteria** node to display a list of named view criteria.
2. Drag the named view criteria item and drop it onto the page or onto the Structure window.
3. From the context menu, choose **Create > ADF Filtered Table**.
4. In the Edit Table Columns dialog, you can rearrange any column and select table options.

   Because the table is created by JDeveloper during quick query creation, the filtering option is automatically enabled and not user-selectable, as shown in Figure 36-24.

Figure 36-24   Edit Table Columns Dialog for Filtered Table



Create Table

Row Selection

● None
○ Single Row
○ Multiple Rows

□ Enable Sorting
☑ Enable Filtering
□ Read-Only Table

□ Generate columns dynamically at runtime

Columns:

Display Label	Value Binding	Component To Use
<default>	Id	ADF Input Text w/ Label
<default>	Name	ADF Input Text w/ Label
<default>	ShortDesc	ADF Input Text w/ Label
<default>	LongtextId	ADF Input Text w/ Label
<default>	ImageId	ADF Input Text w/ Label
<default>	SuggestedWhlsPrice	ADF Input Text w/ Label
<default>	WhlslUnits	ADF Input Text w/ Label

37

Creating Databound Calendar and Carousel Components

This chapter describes how to create calendar and carousel components from data modeled with ADF Business Components, using ADF data controls and ADF Faces components.
This chapter includes the following sections:

About Databound ADF Faces Calendar and Carousel Components

The ADF Faces calendar component displays user activities by day, week, month, or by list view. The carousel component gives the user the ability to view an image from a series of images.

ADF Faces calendar and ADF Faces carousel are complex components that you can use to include calendar functions or display rotating images in your application. The ADF Faces calendar displays activities in daily, weekly, monthly, or list views for a given provider. The calendar is configurable to display only some of the views. The calendar includes a toolbar with built-in functionality that allows a user to change the view (between daily, weekly, monthly, or list), go to the previous or next day, week, or month, and return to today. The toolbar is customizable and allows you to choose which buttons and text to display, and you can also add buttons or other components.

The ADF Faces carousel displays images in a revolving loop that the user can select by using the slider or clicking on images. It can be configured to have either a horizontal or vertical orientation. You can use other components in conjunction with the carousel. You can add a toolbar or menu bar, and then add buttons or menu items that allow users to perform actions on the current object.

Databound ADF Faces Calendar and Carousel Components Use Cases and Examples

The ADF Faces calendar can be used whenever you want to add a calendar feature to your application. You will need to have the relevant data in your data store that represents the content provided by the calendar, such as the date, time, title, location, and owner. You can use the familiar patterns of **entity objects** and **view objects** to model the data and then use drag and drop from the Data Controls panel to create the calendar.

The carousel component gives the user the ability to view an image from a series of images. The user can see partial views of the images before and after the image being viewed and can scroll through each image in the sequence. The user can do so using a slider or navigation buttons. The carousel is useful for showing objects that require a highly visual presentation. For example, it can be used to display a photographic collection or merchandise in a catalog.

Additional Functionality of Databound ADF Faces Calendar and Carousel Components

You may find it helpful to understand other **Oracle ADF** features before you configure or use ADF Faces calendar and carousel components. Additionally, you may want to read about what you can do with your calendar and carousel components. Following are links to other functionality that may be of interest.

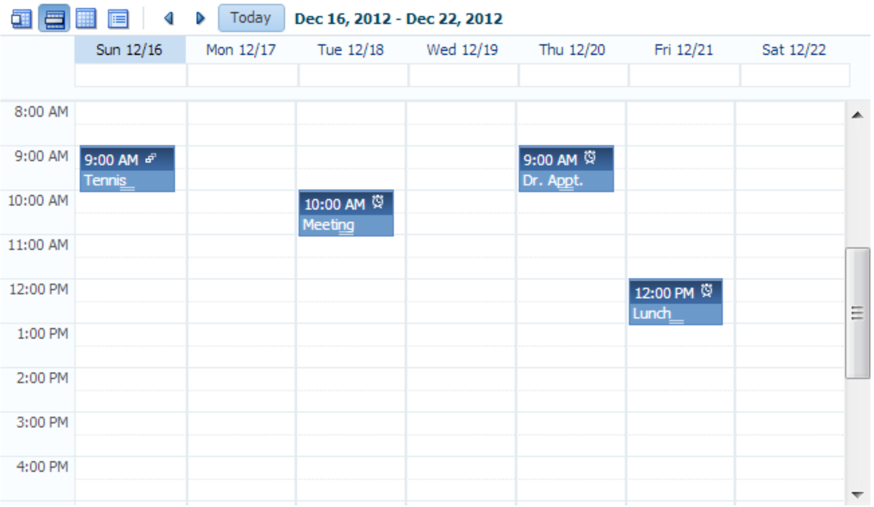
- You can customize the calendar for individual users so that the calendar appears in the selected configuration when that user accesses the calendar. For more information, see Allowing User Customizations at Runtime.
- Calendars are based on entity objects. For information on creating entity objects, see Creating a Business Domain Layer Using Entity Objects.
- For more information about partial page rendering and the partialTriggers attribute, see the Rerendering Partial Page Content chapter of *Developing Web User Interfaces with Oracle ADF Faces*.
- The carousel component uses a tree binding to iterate over the data. For more information about the tree binding, see Iterator and Value Bindings for Tables.

Using the ADF Faces Calendar Component

The ADF Faces calendar can be used whenever you want to add a calendar feature to your application. You will need to have the relevant data in your data store that represents the content provided by the calendar, such as the date, time, title, location, and owner.

ADF Faces includes a calendar component that displays activities in daily, weekly, or monthly views. Figure 37-1 shows an ADF Faces calendar in weekly view mode with some sample activities.

Figure 37-1 ADF Faces Calendar



The calendar component also includes the following functionality:

- A toolbar that allows users to switch between monthly, weekly, daily, and list views.

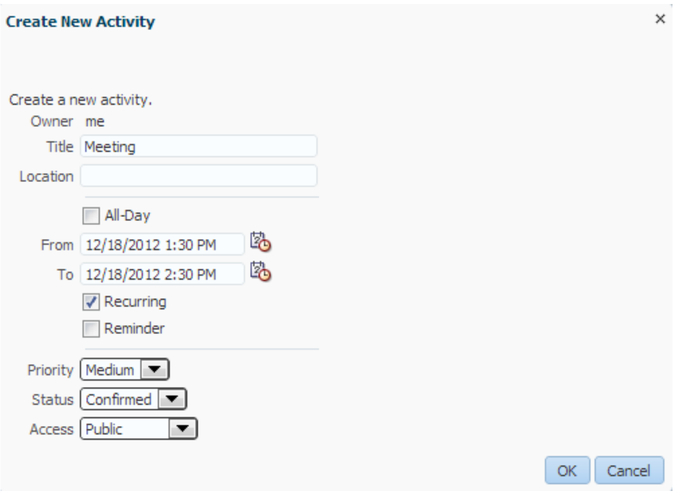
> **Tip:**
>
> When these toolbar buttons are used, attribute values on the calendar are changed. You can configure these values to be persisted so that they remain for a particular user whenever they accesses the calendar. For more information, see Allowing User Customizations at Runtime.

- Configurable start of the week days and start of the day hours. For example, a calendar's week might start on Sunday and the day might show 8:00 am at the top.
- Configurable styles using skinning keys.

Additionally, you can implement the following functionality using other ADF Faces components and the rich client framework:

- Popup functionality. Components placed in supported facets that respond to certain events and allow the user to act on activities or the calendar. For example, when a user clicks an activity in the calendar, the `CalendarActivityEvent` is invoked and any popup component in the `ActivityDetail` facet is displayed. You might use a dialog component that contains a form where users can view and edit the activity, as shown in Figure 37-2.

Figure 37-2 Edit Dialog for ActivityDetail Facet



Create New Activity

Create a new activity.

Owner me

Title Meeting

Location

□ All-Day

From 12/18/2012 1:30 PM

To 12/18/2012 2:30 PM

☑ Recurring

□ Reminder

Priority Medium

Status Confirmed

Access Public

- Drag and drop capability: You can add the `calendarDropTarget` tag that allows a user to drag an activity to another place on the calendar. You then implement the functionality so that the time is actually changed on the activity and persisted to the data store.
- Toolbar customization: By default, the toolbar contains buttons that allow the user to switch between the different views, along with previous and next buttons and a button that returns to the current date. The toolbar also displays the current date range (or the date when in day view). You can customize the toolbar by adding facets that contain additional buttons of your choosing.
- Skinning: The calendar uses skinning keys to determine things like colors and icons used. You can extend the skin to change the appearance of the calendar.

Details for configuring the built-in functionality or for implementing additional functionality can be found in the Using a Calendar Component chapter of *Developing Web User Interfaces with Oracle ADF Faces*.

An ADF Faces Calendar component must be bound to a `CalendarModel` class. This class can be created for you when you use ADF Business Components to manage your calendar's data. For example, say you have data in your data store that represents the details of an activity, such as the date, time, title, location, and owner. When you create an entity object to represent that data, and then a view object to display the data, you can drag and drop the associated collection from the Data Controls panel to create the calendar. JDeveloper will declaratively create the model and bind the view to that model so that the correct data will display when the calendar is launched. However, in order for the model to be created, your entity objects in the data model project with ADF Business Components and your view objects in the same project must contain date-effective attributes. Additionally, the view objects must contain variables that will be used to modify the query to return the correct activities for the given date range.

How to Create the ADF Faces Calendar

Before you can create a calendar on a JSF page, you must first create an entity object with specific attributes that represent attributes on a calendar. You then must create a view object from that entity object, and modify the query to use named bind variables that represent the date range and current time zone to display. This will allow the query to return only the activities that should be displayed in the given view on the calendar.

For example, say you have a database table that represents an activity. It has a column for title, start time, end time, and a reference to a provider object that represents the owner. You would create an entity object and a view object based on that table (ensuring that it meets the requirements, as described in the following steps). To the view object, you would then add named bind variables for the start and end times currently displayed on the calendar, along with the time zone currently in use by the calendar, so that the query returns only those activities that fall within that time range.

Once you add the calendar component to a JSF page, you can configure it and add functionality.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create a calendar. For more information, see Using the ADF Faces Calendar Component.

To create an ADF Faces calendar:

1. Create an entity object based on your data source. The entity object must include the attributes shown in Table 37-1. The attributes do not have to use the names shown in the table; they can be named anything. However, they must be of one of the types noted. You will map these attributes to attributes in the `CalendarModel` in a later step.

   Table 37-1 Required Attributes for a Calendar

Attribute	Valid Types	Description

Start time	`java.util.Date,`
	`java.sql.Date,`
	`oracle.jbo.domain.Date,`
	`oracle.jbo.domain.TimeStamp`
		Start time for the activity

End time	`java.util.Date,`
	`java.sql.Date,`
	`oracle.jbo.domain.Date,`
	`oracle.jbo.domain.TimeStamp`
		End time for the activity

ID	`String`	Unique ID
Provider ID	`String`	ID of the provider object that represents the owner of the activity
Title	`String`	Short description of the activity

   The entity object can also contain the known (but not required) attributes shown in Table 37-2:

Table 37-2 Optional Attributes for a Calendar

Attribute	Type	Description
Recurring	`String` or `CalendarActivity.Recurring`	Status of recurrence for the activity. Valid values are `SINGLE` (does not recur), `RECURRING`, or `CHANGED` (this activity was part of the recurring activity but has been modified to be different from parent activity).
Reminder	`String` or `CalendarActivity.Reminder`	Whether or not the activity has an associated reminder. Valid values are `ON` or `OFF`.
Time Type	`String` or `CalendarActivity.TimeType`	Type of time associated with the activity. Valid values are `ALLDAY` and `TIME`. Activities that have a value of `ALLDAY` do not have any time associated with them. They are considered to span the entire day. Activities with a value of `TIME` have a specific time duration.
Location	`String`	Location of an activity.
Tags	`Set` of `String` values or a semicolon-separated list of `String` values.	Keywords for the activity.

Your entity objects can also contain other attributes that the `CalendarModel` has no knowledge of. You will be able to add these to the model as custom properties in a later step.

For information on creating entity objects, see Creating a Business Domain Layer Using Entity Objects.

2. Create an associated view object. In the Query page of the overview editor, create named bind variables for the following:
   - A string that represents the time zone
   - A date that represents the start time for the current date range shown on the calendar
   - A date that represents the end time for the current date range shown on the calendar

   > **Tip:**
   >
   > Dates in an ADF Faces calendar are "half-open," meaning that the calendar returns all activities that start on or after the start time and before (but not on) the end time.

   For more information about creating named bind variables, see Working with Bind Variables.

3. Create an entity object that represents the provider (owner) of activities. The entity object must include the attributes shown in Table 37-3. The attributes do not have to use the names shown in the table; they can be named anything. However,

they must be of the type noted. You will map these attributes to attributes in the `CalendarProvider` class in a later step.

Table 37-3 Attributes for a CalendarProvider Class

Attribute	Type	Description
Id	`String`	Unique ID.
Display Name	`String`	The name of the provider that can be displayed in the calendar.

4. Create a view object for the provider.
5. Ensure that the new view objects are part of the **application module**, and if needed, refresh the Data Controls panel.
6. Create your JSF page, as documented in Creating a Web Page.
7. From the Data Controls panel, drag the collection that represents the view object for the activity created in Step 2 and drop it as a **Calendar**.

> **Tip:**
>
> The Calendar option will appear in the context menu only if the view object contains the required attributes documented in Table 37-1 and the bind variables described in Step 2.

8. Complete the Calendar Bindings dialog to map the bind variables and attributes to the `CalendarModel` and the `CalendarProvider` classes. For additional help, click **Help** or press F1.
9. By default, the calendar will be read-only and will return only those activities currently in the data store. You will need to configure the calendar and implement additional functionality as described in the Using a Calendar Component chapter of *Developing Web User Interfaces with Oracle ADF Faces*.

   For example, to allow creation of a new activity, you might create an input form in a dialog (as described in the How to Create a Dialog section of *Developing Web User Interfaces with Oracle ADF Faces*) using the same data control collection used to create the calendar. For more information about creating input forms, see Creating an Input Form.

What Happens When You Create a Calendar

When you drop a collection as a calendar, JDeveloper:

- Defines an iterator binding to the collection of activities, and another iterator binding to the collection of providers.
- Defines an action binding to the `executeWithParams` operation on the activities collection. It is this operation that will be invoked to execute the query to return the activities to display. Because the operation requires parameters to determine the date range and time zone, `NamedData` elements are also created for each of the parameters (created as named bind variables on the view object). For more information about `NamedData` elements, see Method Parameters.

Note:

A runtime error will occur if you specify `java.sql.Date` as the `NDType` in the calendar page definition file. Use one of the following supported data types instead:

- `oracle.jbo.domain.Timestamp`
- `oracle.jbo.domain.Date`
- `java.sql.Timestamp`
- `java.util.Date`

- Defines a calendar binding. This binding contains a `node` element that represents a row in the collection and maps the data control attributes to the calendar activity's attributes, as defined when using the wizard.The `value` is the data control attribute and the `type` is the calendar attribute. For any custom defined attributes, the `type` will be `custom` and `value` will be the data control attribute. Each row (node) is represented by a `rowKey`, which is the activity ID.

  There is also a `providerDefinition` element that determines the source and mapping of available providers. This mapping allows the calendar model to filter activities based on the state of the provider (either enabled or disabled).

Tip:

To access a custom attribute, use the `CalendarActivity.getCustomAttributes()` method, passing in the name of the attribute as defined by the `value` element.

The following example shows the page definition code for a calendar.

```
<executables>
    <iterator Binds="ActivityView1" RangeSize="-1"
              DataControl="AppModuleDataControl" id="ActivityView1Iterator"/>
    <iterator Binds="EmployeesView1" RangeSize="25"
              DataControl="AppModuleDataControl" id="EmployeesView1Iterator"/>
  </executables>
  <bindings>
    <action IterBinding="ActivityView1Iterator" id="ExecuteWithParams"
            RequiresUpdateModel="true" Action="executeWithParams">
      <NamedData NDName="startTime"
                 NDValue="#{bindings.ActivityView1.startDate}"
                 NDType="oracle.jbo.domain.Date"/>
      <NamedData NDName="endTime" NDValue="#{bindings.ActivityView1.endDate}"
                 NDType="oracle.jbo.domain.Date"/>
      <NamedData NDName="timeZone"
                 NDValue="#{bindings.ActivityView1.timeZoneId}"
                 NDType="java.lang.String"/>
    </action>
    <calendar IterBinding="ActivityView1Iterator" id="ActivityView1"
              xmlns="http://xmlns.oracle.com/adf/faces/binding"
              ActionBindingName="ExecuteWithParams">
      <nodeDefinition DefName="model.ActivityView">
        <AttrNames>
```

```
              <Item Type="id" Value="Id"/>
              <Item Type="providerId" Value="ProviderId"/>
              <Item Type="title" Value="Title"/>
              <Item Type="startTime" Value="StartTime"/>
              <Item Type="endTime" Value="EndTime"/>
            </AttrNames>
          </nodeDefinition>
          <providerDefinition IterBindingName="EmployeesView1Iterator">
            <AttrNames>
              <Item Type="id" Value="EmployeeId"/>
              <Item Type="displayName" Value="FirstName"/>
            </AttrNames>
          </providerDefinition>
        </calendar>
      </bindings>
```

JDeveloper inserts code onto the JSF page that binds the calendar value to the `CalendarModel` class, as shown in the following example.

```
<af:form>
  <af:calendar value="#{bindings.ActivityView1.calendarModel}"/>
</af:form>
```

The `CalendarModel` class uses `CalendarActivityDefinition` class to access the calendar binding.

What Happens at Runtime: How the Calendar Binding Works

When the calendar is accessed, the `executeWithParams` operation is invoked, with the value of the `startDate` and `endDate` parameters determined by the value of the calendar component's `view` and `activeDay` attributes. For example, if the `view` attribute is set to `month` and the `activeDay` is set to the current date (say, May 5, 2013), then the value for the `startDate` would be `May 1, 2013` and the `endDate` value would be `May 31, 2013`. By default, the time zone value is taken from the `time-zone` setting in the `trinidad-config.xml` file (for more information, see the Configuration in trinidad-config.xml section of *Developing Web User Interfaces with Oracle ADF Faces*). Therefore, the query would be restricted to return only activities that fall within that date range.

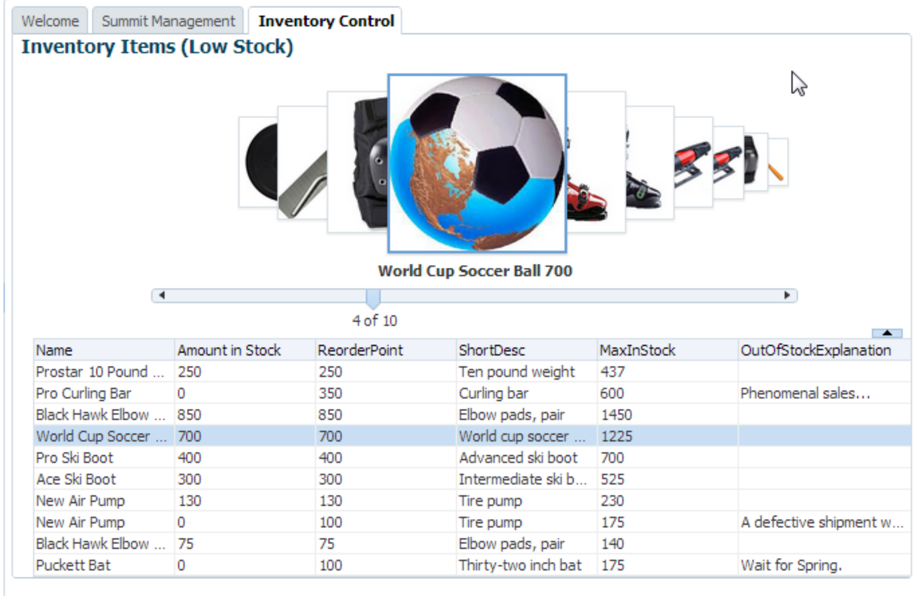
When the query returns data, because the calendar component is bound to the `CalendarModel`, the `CalendarModel` uses the `CalendarActivityDefinition` class to access the calendar binding class and map the values from the data source to the calendar, using the mappings provided by the binding.

Using the ADF Faces Carousel Component

The ADF Faces carousel component gives the user the ability to view an image from a series of images. The user can see partial views of the images before and after the image being viewed and can scroll through each image in the sequence.

You can display images in a revolving carousel, as shown in Figure 37-3. Users can change the image at the front by using either the slider at the bottom or by dragging another image to the front.

Figure 37-3   Carousel Component



Inventory Items (Low Stock)

World Cup Soccer Ball 700

4 of 10

Name	Amount in Stock	ReorderPoint	ShortDesc	MaxInStock	OutOfStockExplanation
Prostar 10 Pound ...	250	250	Ten pound weight	437	
Pro Curling Bar	0	350	Curling bar	600	Phenomenal sales...
Black Hawk Elbow ...	850	850	Elbow pads, pair	1450	
World Cup Soccer ...	700	700	World cup soccer ...	1225	
Pro Ski Boot	400	400	Advanced ski boot	700	
Ace Ski Boot	300	300	Intermediate ski b...	525	
New Air Pump	130	130	Tire pump	230	
New Air Pump	0	100	Tire pump	175	A defective shipment w...
Black Hawk Elbow ...	75	75	Elbow pads, pair	140	
Puckett Bat	0	100	Thirty-two inch bat	175	Wait for Spring.

Instead of containing a child `carouselItem` component for each image to be displayed, and then binding these components to the individual images, the `carousel` component is bound to a complete collection and repeatedly renders one `carouselItem` component by stamping the value for each item, similar to the way a tree stamps out each row of data. As each item is stamped, the data for the current item is copied into a property that can be addressed using an EL expression using the `carousel` component's `var` attribute. Once the carousel has completed rendering, this property is removed or reverted back to its previous value. Carousels contain a `nodeStamp` facet, which is a holder for the `carouselItem` component used to display the text and short description for each item, and is also the parent component to the image displayed for each item. For more information about the carousel component, see the Displaying Images in a Carousel section of *Developing Web User Interfaces with Oracle ADF Faces*.

How to Create a Databound Carousel Component

When using a carousel component in a Fusion web application, you create the component using the Data Controls panel. You also use a managed bean to handle the carousel spin event and for other logic you may need to display your items.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create a carousel. For more information, see Using the ADF Faces Carousel Component.

You may also find it useful to understand functionality that can be used with carousels. For more information, see Additional Functionality of Databound ADF Faces Calendar and Carousel Components.

Perform the following in JDeveloper:

1. Create a view object for the collection to be displayed in the carousel. For example, to create the carousel shown in Figure 37-3, you use the `Inventory` view object.l
2. Create a managed bean to hold a method that handles the spinning of the carousel. The following example shows the handler method that might be used to display items in the carousel.

   ```
   public void handleCarouselSpin(CarouselSpinEvent event)
   {
   // This method is invoked when there is a spin event on the carousel in
   // InventoryControl.jsff. The purpose is to get the newly selected item in
   the
   // carousel and update the iterator current row with the value

   // get the bindingContainer
     DCBindingContainer dcBindings = (DCBindingContainer)
     BindingContext.getCurrent().getCurrentBindingsEntry();
     // get the NewItemKey from the event object. The first entry in the list
   is the
     // currently selected item
     List itemKeyList = (List) event.getNewItemKey();
     // create a Key ojbect instance and set it to the first key in the list
     Key currentItemKey = (Key) itemKeyList.get(0);
     // get the iteratorBinding
     DCIteratorBinding inventoryIterator =
    dcBindings.findIteratorBinding("InventoryIterator");
    // set the iterator
    inventoryIterator.setCurrentRowWithKey(currentItemKey.toStringFormat(true));
   }
   ```

To create a databound carousel component:

1. From the Data Controls panel, drag the collection for the view object on to the page and choose **Carousel** from the context menu.
2. In the Properties window, in the Behavior section, bind the **CarouselSpinListener** to a handler method that you created in the prerequisites.
3. In the Structure window, expand the `carousel` component and the `nodeStamp` facet, and select the `carouselItem` component.
4. Bind the `CarouselItem` component's `text` attribute to the associated property on the view object using variable value set on the carousel's `var` attribute, which by default is set to `item`. So for example, to use the `Name` attribute on the view object, the value of the `carouselItem`'s `text` attribute might be `#{item.Name}`.

   You can also bind other properties to the `text` attribute if you want to display more information. For example, the inventory carousel binds both the `Name` and the `AmountInStock` properties to the `text` attribute.
5. In the ADF Faces page of the Components window, from the General Controls panel, drag an **Image** and drop it as a child to the `carouselItem`.

   In the Insert Image dialog, enter the path to the source for the images, being sure to use the variable for the item in the carousel. For example, the path to the image files for the products would normally be:

   ```
   /images/products/#{item.Filename}
   ```

   Bind the `shortDesc` attribute to the text you want to appear when the mouse hovers over the image. For example:

```
#{item.Name}
```

For information about setting other attributes of the `carousel` and `carouselItem` components, see the How to Create a Carousel section of *Developing Web User Interfaces with Oracle ADF Faces*.

6. If you want to provide additional information about the items in the carousel, you can drag and drop the same view object onto the page, for example, as a table. For the components in the table to redisplay the information for the current item displayed once the carousel is spun, you need to set the `partialTrigger` attribute of the component containing the form to the carousel component's ID.

   For example, a table displays the information for each item in Figure 37-3. The `partialTrigger` attribute for the `table` component is set to `c1`, which is the carousel component's ID. This means that whenever the `carouselItem` invokes the `CarouselSpinEvent`, the `table` will be refreshed, causing the row that displays information about the item that was just made current in the carousel is also made current in the table.

   Additionally, the `partialTrigger` attribute for the carousel and carousel item is set to `t1`, which is the table's ID. This means that whenever a selection event occurs on the table, the carousel is refreshed to make the item that was just made current in the table, the current item in the carousel.

   For more information about partial page rendering and the `partialTriggers` attribute, see the Rerendering Partial Page Content chapter of *Developing Web User Interfaces with Oracle ADF Faces*.

   The following example shows the page code for the carousel displayed in Figure 37-3.

```
<f:facet name="first">
  <af:carousel
currentItemKey="#{bindings.Inventory.treeModel.rootCurrencyRowKey}"
               value="#{bindings.Inventory.treeModel}" var="item" id="c1"
               carouselSpinListener="#{InventoryControl.handleCarouselSpin}"
               partialTriggers="::t1">
    <f:facet name="nodeStamp">
      <af:carouselItem id="ci1" text="#{item.Name} #{item.AmountInStock}"
                       partialTriggers="::t1">
        <af:image source="/images/products/#{item.Filename}"
                  shortDesc="#{item.Name}" id="i1"/>
      </af:carouselItem>
    </f:facet>
  </af:carousel>
</f:facet>
<f:facet name="second">
  <af:table value="#{bindings.Inventory.collectionModel}" var="row"
            rows="#{bindings.Inventory.rangeSize}"
            emptyText="#{bindings.Inventory.viewable ? 'No data to
display.' :
                        'Access Denied.'}"
            fetchSize="#{bindings.Inventory.rangeSize}"
rowBandingInterval="0"

selectedRowKeys="#{bindings.Inventory.collectionModel.selectedRow}"

selectionListener="#{bindings.Inventory.collectionModel.makeCurrent}"
            rowSelection="single" id="t1"
            columnStretching="last" partialTriggers="::c1">
```

What Happens When You Create a Carousel

When you drop a collection from the Data Controls panel as a carousel, a tree value binding is created. A tree consists of a hierarchy of nodes, where each subnode is a branch off a higher-level node.

The tree binding iterates over the data exposed by the iterator binding. The carousel wraps the result set from the iterator binding in a `treeModel` object, which is an extension of the `collectionModel`. The `collectionModel` allows each item in the collection to be available within the carousel component using the `var` attribute. For more information about the tree binding, see Iterator and Value Bindings for Tables.

JDeveloper adds both a `carousel` component and its child `carouselItem` component onto the page, as shown in the following example.

```
<af:carousel
        currentItemKey="#{bindings.Products.treeModel.rootCurrencyRowKey}"
        value="#{bindings.Products.treeModel}" var="item"
        id="c1"
        carouselSpinListener="#{InventoryControl.handleCarouselSpin}">
  <f:facet name="nodeStamp">
    <af:carouselItem id="ci1" text="#{item.Name}"/>
  </f:facet>
</af:carousel>
```

The carousel value is bound to the `treeModel` for the associated collection, and the `currentItemKey` attribute of the carousel is bound to the `rootCurrencyRowKey` of the binding object. In this example, the carousel iterates over the items in the `Products` iterator binding. The iterator binding binds to a `rowKeySet` that keeps track of the current product. By default, the `currentItemKey` attribute of the carousel is bound to the `rootCurrencyRowKey` of the binding object, which causes the product currently displayed at the front of the carousel to be the root and the current item. The `carouselItem` component accesses the current data object for the current item presented to the carousel tag using the `item` variable.

38

Creating Databound Chart, Picto Chart, and Gauge Components

This chapter describes how to create charts, picto charts, or gauges from data modeled with ADF Business Components, using ADF data controls and ADF Faces components in a Fusion web application. Specifically, it describes how you can use ADF Data Visualization Tools (DVT) `area`, `bar`, `bubble`, `funnel`, `horizontal bar`, `line`, `pie`, `polar`, `radar`, `scatter`, and `spark` charts, `pictoCharts` and `dial`, `LED`, `rating`, and `status meter` gauges that visually represent business data. It describes how to use ADF data controls to create these components with data-first development.
If you are designing your page using simple UI-first development, then you can add the chart or gauge to your page and configure the data bindings later. For information about the data requirements, tag structure, and options for customizing the look and behavior of the chart and gauge components, see Using Chart Components, Using Picto Chart Components and Using Gauge Components in *Developing Web User Interfaces with Oracle ADF Faces*.

This chapter includes the following sections:

About ADF Data Visualization Chart, Picto Chart, and Gauge Components

ADF Data Visualization components provide extensive graphical and tabular capabilities for visually displaying and analyzing business data. Each component must be bound to data before it can be rendered since the appearance of the components is dictated by the data that is displayed.

Both chart and gauge components render graphical representations of data. However, charts allow you to evaluate multiple data points on multiple axes in a variety of ways. Many chart types assist in the comparison of results from one group with the results from another group. In contrast, gauges focus on a single data point and examine that point relative to minimum, maximum, and threshold indicators to identify problems.

Picto charts are versatile components that represent absolute numbers or parts of a population. They are arranged in a flowing layout, and hence can be used in a variety of ways to represent data points in an aesthetically pleasing manner.

The prefix `dvt:` occurs at the beginning of each data visualization component name indicating that the component belongs to the ADF Data Visualization Tools (DVT) tag library.

Data Visualization Components Use Cases and Examples

For detailed descriptions of data visualization use cases and examples, see the following:

- Chart components: Chart Component Use Cases and Examples section in *Developing Web User Interfaces with Oracle ADF Faces*.
- Gauge components: Gauge Component Use Cases and Examples section in *Developing Web User Interfaces with Oracle ADF Faces*.
- Picto Chart components: Picto Chart Component Use Cases and Examples section in *Developing Web User Interfaces with Oracle ADF Faces*.

End User and Presentation Features

Visually compelling data visualization components enable end users to understand and analyze complex business data. The components are rich in features that provide out-of-the-box interactivity support. For detailed descriptions of the end user and presentation features for each component, see the following:

- Chart components: End User and Presentation Features of Charts section in *Developing Web User Interfaces with Oracle ADF Faces*.
- Gauge components: End User and Presentation Features of Gauge Components section in *Developing Web User Interfaces with Oracle ADF Faces*.
- Picto Chart components: End User and Presentation Features of Picto Charts section in *Developing Web User Interfaces with Oracle ADF Faces*.

Additional Functionality for Data Visualization Components

You may find it helpful to understand other **Oracle ADF** features before you data bind your data visualization components. Additionally, once you have added a data visualization component to your page, you may find that you need to add functionality such as validation and accessibility. Following are links to other functionality that data visualization components use:

- Partial page rendering: You may want a data visualization component to refresh to show new data based on an action taken on another component on the page. For more information, see the Rerendering Partial Page Content chapter in *Developing Web User Interfaces with Oracle ADF Faces*.
- Personalization: Users can change the way the data visualization components display at runtime, those values will not be retained once the user leaves the page unless you configure your application to allow user customization. For more information, see the Allowing User Customization on JSF Pages chapter in *Developing Web User Interfaces with Oracle ADF Faces*.
- Accessibility: You can make your data visualization components accessible. For more information, see the Developing Accessible ADF Faces Pages chapter in *Developing Web User Interfaces with Oracle ADF Faces*.
- Skins and styles: You can customize the appearance of data visualization components using an ADF skin that you apply to the application or by applying CSS style properties directly using a style-related property (`styleClass` or `inlineStyle`). For more information, see the Customizing the Appearance Using

Styles and Skins chapter in *Developing Web User Interfaces with Oracle ADF Faces*.

- Placeholder data controls: If you know the data visualization components on your page will eventually use ADF data binding, but you need to develop the pages before the data controls are ready, then you should consider using placeholder data controls, rather than manually binding the components. Using placeholder data controls will provide the same declarative development experience as using developed data controls. For more information, see Designing a Page Using Placeholder Data Controls .

Creating Databound Charts

Charts are based on data collections. To create a databound chart, you drag and drop a collection from the Data Controls panel onto the JSF page and use a dialog to bind the data collection attributes in the chart.

Charts display series and groups of data. Series and groups are analogous to the rows and columns of a grid of data. Typically, the rows in the grid appear as a series in a chart, and the columns in the grid appear as groups.

For most charts, a series appears as a set of markers that are the same color. Typically, the chart legend shows the identification and associated color of each series. For example, in a bar chart, the yellow bars might represent the sales of shoes and the green bars might represent the sales of boots.

Groups appear differently in different chart types. For example, in a stacked bar chart, each stack is a group. A group might represent time periods, such as years. A group might also represent geographical locations such as regions.

Depending on the data requirements for a chart type, a single group might require multiple data values. For example, a scatter chart requires two values for each data marker. The first value determines where the marker appears along the x-axis while the second value determines where the marker appears along the y-axis. For details about chart data requirements, see Chart Component Data Requirements in *Developing Web User Interfaces with Oracle ADF Faces*.

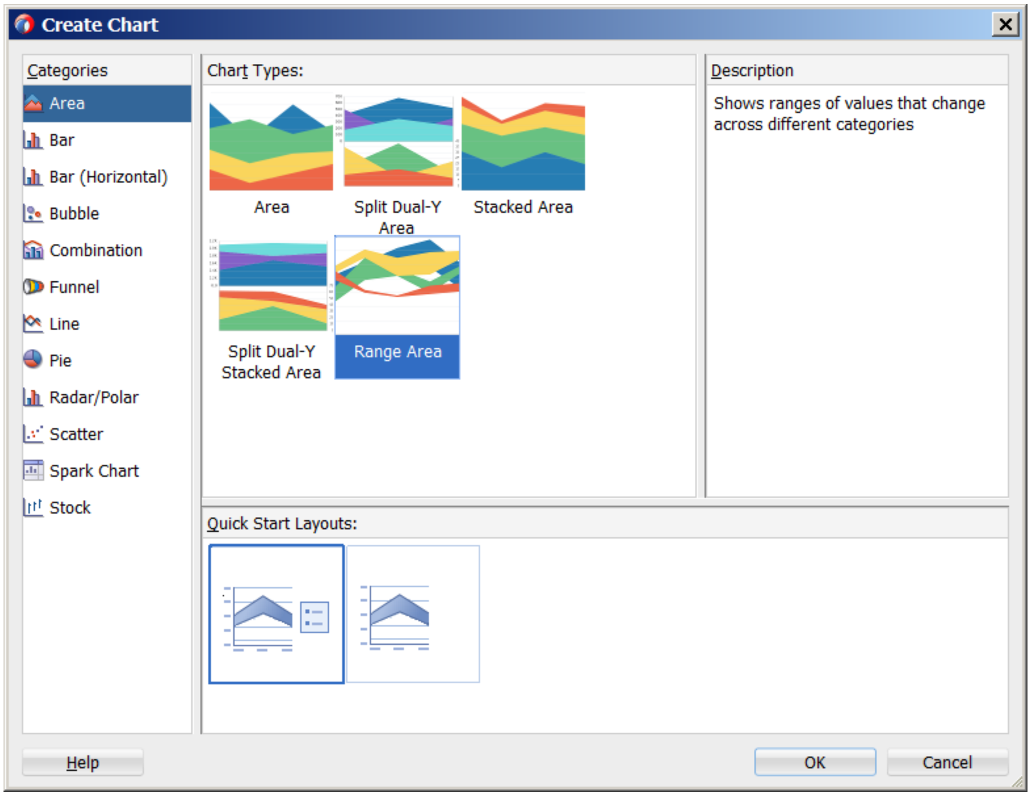
The attributes in a data collection can be data values or categories of data values. Data values are numbers represented by markers, like bar height or points in a scatter chart. Categories of data values are members represented as axis labels or appear as additional properties in a tooltip. The role that an attribute plays in the bindings (either data values or identifiers) is determined by both its data type and where it gets mapped (for example, bars vs. x-axis).

When you create a chart using a data collection inserted from the Data Controls panel, a Component Gallery allows you to choose from a wide number of chart categories, chart types, and layout options. Chart categories group together one or more types of chart. For example, the Area category includes the following types of charts:

- Area
- Split Dual-Y Area
- Stacked Area
- Split Dual-Y Stacked Area
- Range Area

Explore the Component Gallery that appears when you create a chart to view available chart categories, types, and descriptions for each one. Figure 38-1 shows the Component Gallery that appears for ADF charts when you use the Data Controls panel.

Figure 38-1 Component Gallery for Charts



Shows ranges of values that change across different categories

Table 38-1 lists the categories that appear in the Component Gallery for charts. Each category has one or more chart types associated with it.

Table 38-1 ADF Chart Categories in the Component Gallery

Image	Category	Description
	Area	Creates a chart in which data is represented as a filled-in area. Use area charts to show trends over time, such as sales for the last 12 months. Area charts require at least two groups of data along an axis. The axis is often labeled with time periods such as months.

Table 38-1 (Cont.) ADF Chart Categories in the Component Gallery

Image	Category	Description
Bar	Bar	Creates a chart in which data is represented as a series of vertical bars. Use to compare values across products or categories, or to view aggregated data broken out by a time period.
Bar (Horizontal)	Bar (Horizontal)	Creates a chart that displays bars horizontally along the y-axis. Use to provide an orientation that allows you to show trends or compare values.
Bubble	Bubble	Creates a chart in which data is represented by the location and size of round data markers (bubbles). Use to show correlations among three types of values, especially when you have a number of data items and you want to see the general relationships. For example, use a bubble chart to plot salaries (x-axis), years of experience (y-axis), and productivity (size of bubble) for your work force. Such a chart allows you to examine productivity relative to salary and experience.
Combination	Combination	Creates a chart that uses different types of data markers (bars, lines, or areas) to display different kinds of data items. Use to compare bars and lines, bars and areas, lines and areas, or all three.
Funnel	Funnel	Creates a chart that represents data related to steps in a process. Use to compare actual versus target values or to compare values in a sequence of steps. For examples, use the funnel chart to watch a process where the different sections of the funnel represent different stages in the sales cycle.
Line	Line	Creates a chart in which data is represented as a line, as a series of data points, or as data points that are connected by a line. Line charts require data for at least two points for each member in a group. For example, a line chart over months requires at least two months. Typically a line of a specific color is associated with each series of data such as the Americas, Europe, or Asia. Use to compare items over the same time.

Table 38-1 (Cont.) ADF Chart Categories in the Component Gallery

Image	Category	Description
Pie	Pie	Creates a chart that represents a set of data items as proportions of a total. The data items are displayed as sections of a circle causing the circle to look like a sliced pie. Use to show relationship of parts to a whole such as how much revenue comes from each product line.
Scatter	Scatter	Creates a chart in which data is represented by the location of data markers. Use to show correlation between two different kinds of data values such as sales and costs for top products. Scatter charts are especially useful when you want to see general relationships among a number of items.
Spark Chart	Spark	Creates a simple, condensed chart that displays trends or variations, often in the column of a table, or inline with text.
Stock	Stock	Creates a chart in which data shows the high, low, opening and closing prices of a stock and, optionally, the trading volume of the stock.

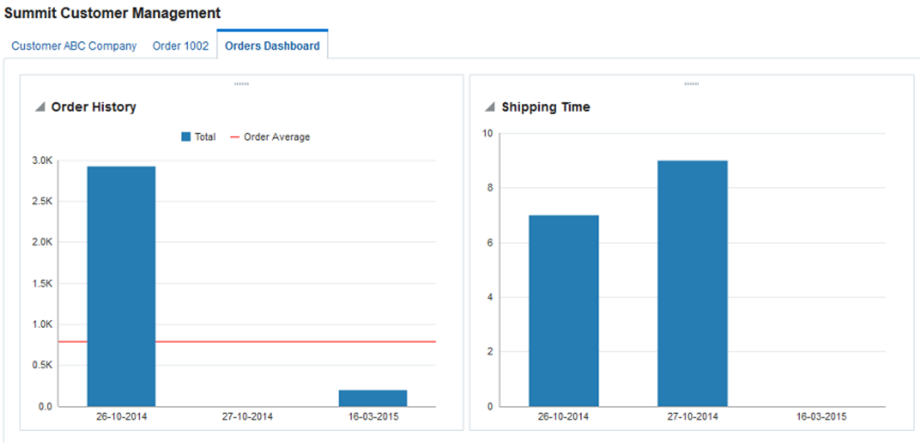
You can also create a chart by dragging a chart component from the Components window. This approach allows you the option of designing the chart's user interface before binding the component to data. A Create Chart dialog appears to view chart types, descriptions, and quick layout options. For more information about creating charts using UI-first development, see Using Chart Components in *Developing Web User Interfaces with Oracle ADF Faces*.

How to Create an Area, Bar, Combination, Horizontal Bar, or Line Chart Using Data Controls

Area, bar, horizontal bar, and line charts require at least two groups of data, with one or more series. The groups are displayed along the chart's X Axis for area, bar, and line charts and on the chart's Y Axis for horizontal bar charts. Series are displayed as the areas, bars, or lines on the chart.

Figure 38-2 shows two bar charts in the Summit ADF sample application. The bar chart on the left is configured to show the customer's order history and order average, and the bar chart on the right shows the ship time for each order. In these examples, the bar charts are configured with groups that represent order and ship dates, and the series are the order totals and ship time averages. The order average is displayed as a red reference line on the Order History chart.

Figure 38-2 Bar Charts in Summit ADF Sample Application



For information about adding reference lines to charts after the chart is created, see "Adding Reference Objects to a Chart" in *Developing Web User Interfaces with Oracle ADF Faces*.

Before you begin:

It may be helpful to have an understanding of databound data visualization charts. See Creating Databound Charts.

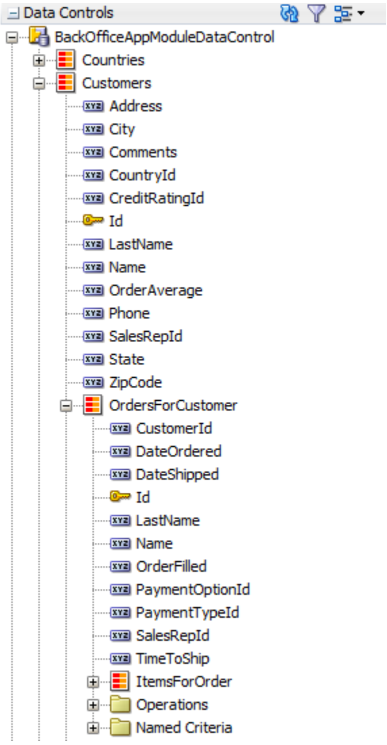
You may also find it helpful to understand functionality that can be added using other Oracle ADF features. See Additional Functionality for Data Visualization Components.

You will need to complete these tasks:

- Create an **application module** that contains instances of the **view objects** that you want in your data model, as described in Creating and Modifying an Application Module.

  For example, the Summit ADF Shipping Time bar chart is based on the `OrdersForCustomer` data collection. Figure 38-3 shows the `OrdersForCustomer` collection in the Data Controls panel.

Figure 38-3 Data Collection for Order Ship Times

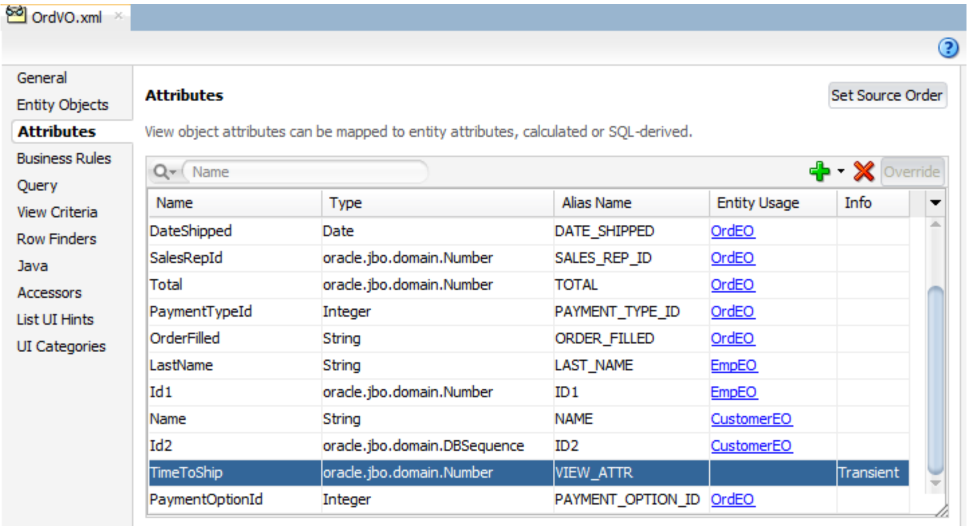


Data Controls

- BackOfficeAppModuleDataControl
  - Countries
  - Customers
    - Address
    - City
    - Comments
    - CountryId
    - CreditRatingId
    - Id
    - LastName
    - Name
    - OrderAverage
    - Phone
    - SalesRepId
    - State
    - ZipCode
    - OrdersForCustomer
      - CustomerId
      - DateOrdered
      - DateShipped
      - Id
      - LastName
      - Name
      - OrderFilled
      - PaymentOptionId
      - PaymentTypeId
      - SalesRepId
      - TimeToShip
      - ItemsForOrder
      - Operations
      - Named Criteria

The `OrdersForCustomer` data control is based on the `OrdVO` view object. Figure 38-4 shows the `OrdVO` view object which contains attributes from the `OrdEO`, `EmpEO`, and `CustomerEO` entity objects. The `OrdEO`, `EmpEO`, and `CustomerEO` entity objects are derived from the `SOrd`, `SEmp`, and `SCustomer` tables in the Summit schema.

Figure 38-4 OrdVO View Object in Summit ADF Sample Application



Attributes

View object attributes can be mapped to entity attributes, calculated or SQL-derived.

Name	Type	Alias Name	Entity Usage	Info
DateShipped	Date	DATE_SHIPPED	OrdEO	
SalesRepId	oracle.jbo.domain.Number	SALES_REP_ID	OrdEO	
Total	oracle.jbo.domain.Number	TOTAL	OrdEO	
PaymentTypeId	Integer	PAYMENT_TYPE_ID	OrdEO	
OrderFilled	String	ORDER_FILLED	OrdEO	
LastName	String	LAST_NAME	EmpEO	
Id1	oracle.jbo.domain.Number	ID1	EmpEO	
Name	String	NAME	CustomerEO	
Id2	oracle.jbo.domain.DBSequence	ID2	CustomerEO	
TimeToShip	oracle.jbo.domain.Number	VIEW_ATTR		Transient
PaymentOptionId	Integer	PAYMENT_OPTION_ID	OrdEO	

The OrdVO object also contains the TimeToShip transient attribute which is represented by the bars on the Summit ADF Shipping Time bar chart. The TimeToShip attribute's value is obtained by a call to the calculateTimeToShip() method: adf.object.calculateTimeToShip(DateOrdered, DateShipped). The calculateTimeToShip() method determines the number of days that the order took to ship and is added to OrdVORowImpl.java. OrdVORowImpl.java contains the implementation methods for the OrdVO view object.

The example below shows the calculateTimeToShip() method in the Summit ADF sample application. The method returns the number of days as an oracle.jbo.domain.Number, which is the type expected by the chart's value attribute.

```
public oracle.jbo.domain.Number calculateTimeToShip(Date ordered,
Date shipped) {
  if (null != shipped) {
    long days = (shipped.getTime() - ordered.getTime()) / (1000 *
60 * 60 * 24);
    return new Number(days);
  } else
    return new Number(0);
}
```

For additional information about adding transient attributes to a view object, see Adding Calculated and Transient Attributes to a View Object.

- Create a JSF page as described in How to Create JSF Pages in *Developing Web User Interfaces with Oracle ADF Faces*.

To create a databound area, bar, combination, horizontal bar, or line chart:

1. From the Data Controls panel, select a collection.

   For example, to create the bar chart that displays the shipping times for each order on the Orders Dashboard page of the Summit ADF sample application, select the OrdersForCustomer collection.

2. Drag the collection onto a JSF page and, from the context menu, choose **Charts.**

3. In the Component Gallery, select a chart category and a chart type and click **OK**.

   The name of the dialog and the input field labels that appear depend on the category, type of chart, and data collection that you select. For example, if you select **Bar** as the chart category and **Bar** as the chart type, then the name of the dialog that appears is Create Bar Chart and the input fields are labeled **Bars** and **X Axis**.

4. Do the following in the dialog to configure the chart to display data:

   - Drag attributes from the **Available** list to the input fields, depending on where you want the values for the attributes to appear at runtime.

     For example, to configure the bar chart in the Summit ADF sample application to display the order dates along the X Axis, drag the DateOrdered attribute from the **Available** list to the **X Axis** field in the Create Bar Chart dialog. To configure the chart to display the time to ship as bars, drag the TimeToShip attribute from the **Available** list to the **Bars** field in the Create Bar Chart dialog.

If you selected a chart type that includes a dual y-axis, the second area, bar, or line that you add will be rendered on the second y-axis. In addition, a `chartSeriesStyle` component will be added to the chart for the series represented on the second y-axis, and the `assignedToY2` attribute of the `chartSeriesStyle` component will be set to true.

If you selected a combination chart, each attribute that you add to the Data Points input field will be rendered as an area, bar, or line chart. By default, the first attribute will be rendered as a bar chart, the second attribute will be rendered as a line chart, and the third attribute will be rendered as an area chart. To change the chart type, you can add a `chartSeriesStyle` component to the combination chart's `seriesStamp` facet.

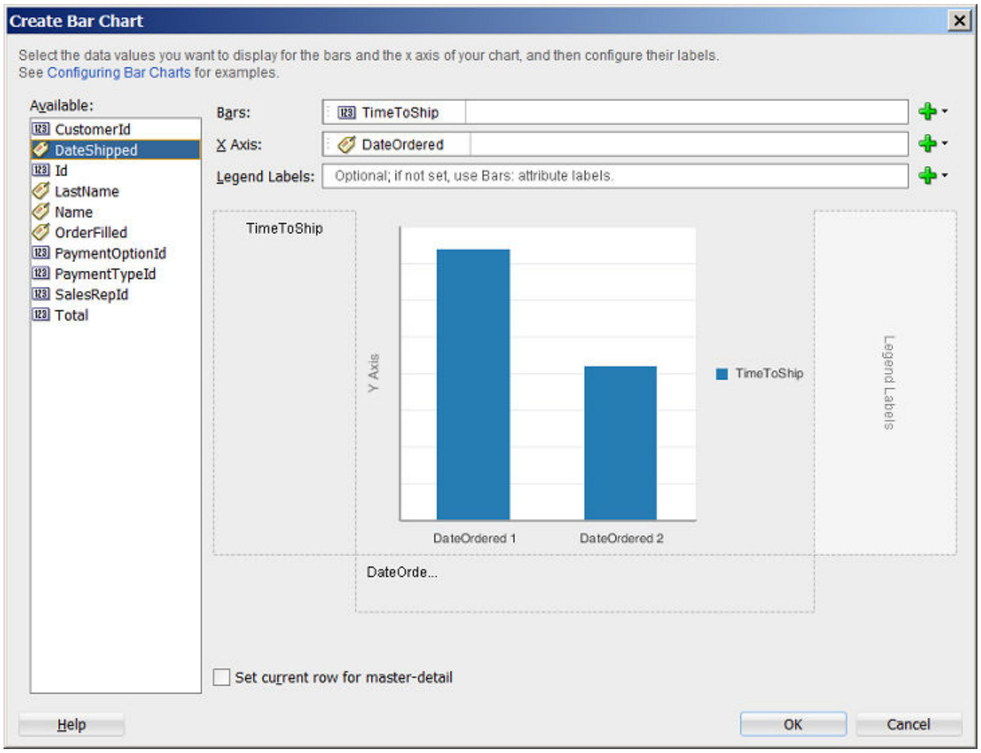
If you selected a range area or range bar chart, there will be two required fields for the y-axis. To configure the chart to display a range of data, drag attributes from the **Available** list to the **High Values** and **Low Values** fields in the Create Range Area Chart or Create Range Bar Chart dialog.

- In the **Attribute Labels** table, accept the default value or select a value from the dropdown list in the **Label** field to specify the label that appears at runtime.

  The underlying data type determines the choices available in the **Label** field. The choice you make determines how that attribute's label is rendered in the chart.

Figure 38-5 shows the Create Bar Chart dialog that generates the Summit ADF sample application Shipping Time bar chart using data from the `DateOrdered` and `TimeToShip` attributes in the `OrdersForCustomer` data collection.

Figure 38-5 Create Bar Chart Dialog for Summit ADF Shipping Time Chart



Create Bar Chart

Select the data values you want to display for the bars and the x axis of your chart, and then configure their labels.
See Configuring Bar Charts for examples.

Available:
CustomerId
DateShipped
Id
LastName
Name
OrderFilled
PaymentOptionId
PaymentTypeId
SalesRepId
Total

Bars: TimeToShip
X Axis: DateOrdered
Legend Labels: Optional; if not set, use Bars: attribute labels.

☐ Set current row for master-detail

5. Click **OK**.

After completing the data binding dialog, you can use the Properties window to specify settings for the chart attributes and you can also use the child tags associated with the chart tag to customize the chart further. For example, the legend display is turned off for the Shipping Time chart in the Summit ADF sample application. For more information about configuring chart components, see Using Chart Components in *Developing Web User Interfaces with Oracle ADF Faces*.

What Happens When You Use the Data Controls Panel to Create a Chart

Dropping a chart from the Data Controls panel has the following effect:

- Creates the bindings for the chart and adds the bindings to the page definition file
- Adds the necessary code for the UI components to the JSF page

The data binding XML that JDeveloper generates represents the physical model of the specific chart type you create. The example below shows the bindings that JDeveloper generated in the page definition file where a vertical bar chart was created using data from the `DateOrdered` and `TimeToShip` attributes in the `OrdersForCustomer` data collection.

```
<tree IterBinding="OrdersForCustomerIterator" id="OrdersForCustomer5"
      ChangeEventPolicy="ppr">
  <nodeDefinition DefName="oracle.summit.model.views.OrdVO"
                  Name="OrdersForCustomer5">
    <AttrNames>
      <Item Value="TimeToShip"/>
      <Item Value="DateOrdered"/>
    </AttrNames>
  </nodeDefinition>
</tree>
```

Charts use a standard tree binding as shown in the example. The node definition shows the name and instance of the data control (`OrdersForCustomer5`), and the view object (`OrdVO`) on which the data control is based. Each displayed attribute is listed in the `AttrNames` definition. For a chart configured for a dual y-axis, the `AttrNames` definition would include an additional item for the series displayed on the second y-axis.

The example below shows the code generated for a vertical bar chart when you drag the `OrdersForCustomer` data collection onto a JSF page and specify `DateOrdered` for the x-axis and `TimeToShip` for the bars.

```
<dvt:barChart id="barChart2" var="row"
              value="#{bindings.OrdersForCustomer5.collectionModel}">
<dvt:chartLegend id="cl2"/>
  <f:facet name="dataStamp">
    <dvt:chartDataItem id="di2" value="#{row.TimeToShip}"
                 group="#{row.DateOrdered}"

series="#{bindings.OrdersForCustomer5.hints.TimeToShip.label}"/>
  </f:facet>
</dvt:barChart>
```

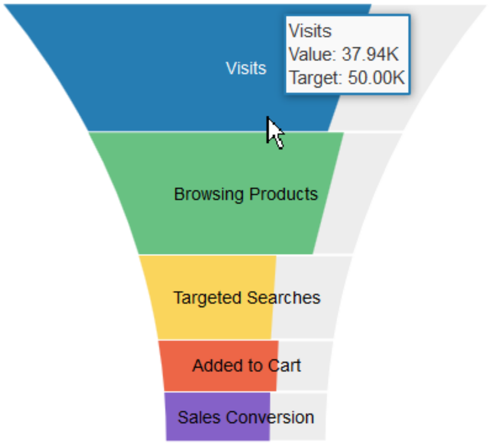
How to Create Databound Funnel Charts

A funnel chart is used to represent the steps in a process. You can create a funnel chart by using the Data Controls panel.

A funnel chart requires at least two values, with a maximum of three. The values represent the actual and target data values for each stage, and the stage value.

Figure 38-6 shows a sample funnel chart. The funnel shows the different stages in a sales conversion process for an online store. Each funnel section shows the number of customers who reached the given stage in the process versus the desired target number of customers for the stage. As the user moves over each section, a tooltip provides the numerical quantity of the actual value and target value for that stage.

Figure 38-6 Sample Funnel Chart



Before you begin:

It may be helpful to have an understanding of databound data visualization charts. For more information, see Creating Databound Charts.

You will need to complete these tasks:

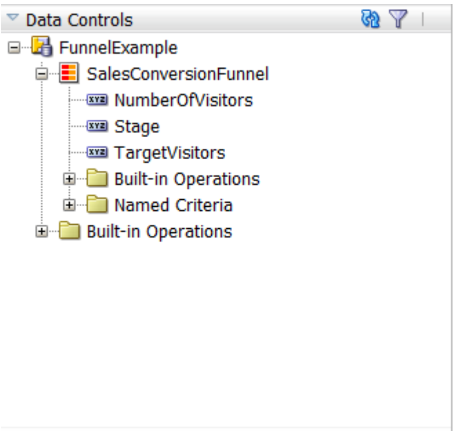
- Create an **application module** that contains instances of the **view objects** that you want in your data model, as described in Creating and Modifying an Application Module.
- Create a JSF page as described in the How to Create JSF Pages section of the *Developing Web User Interfaces with Oracle ADF Faces*.

To create a databound funnel chart:

1. From the Data Controls panel, select a collection.

   For example, the funnel chart displayed above is made using the `SalesConversionFunnel` collection. Figure 38-7 shows the `SalesConversionFunnel` collection in the Data Controls panel.

Figure 38-7 Data Collection for SalesConversionFunnel Collection



Data Controls
- FunnelExample
  - SalesConversionFunnel
    - NumberOfVisitors
    - Stage
    - TargetVisitors
    - Built-in Operations
    - Named Criteria
  - Built-in Operations

2. Drag the collection onto a JSF page and, from the context menu, choose **Charts.**
3. In the Component Gallery, select the funnel chart category and a chart type and click **OK**.

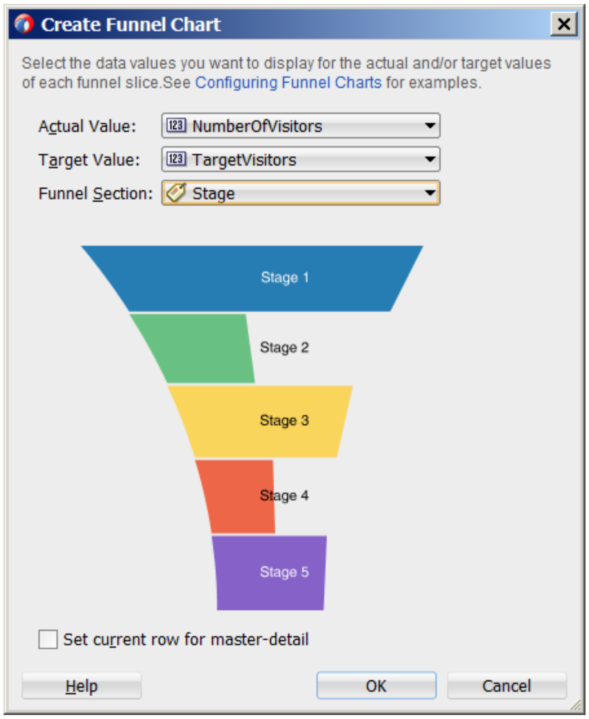
   For example, select the **Funnel** chart category and the **Funnel** chart type to create a vertical funnel chart.
4. Do the following in the dialog to configure the chart to display data:
   - From the dropdown menus of the **Actual Value** and **Target Value** fields, select an attribute to represent series of data.

     For example, to configure the funnel chart in the sample, select the `NumOfVisitors` attribute from the **Actual Value** field's dropdown menu, and the `TargetVisitors` attribute from the **Target Value** field's dropdown menu.
   - From the dropdown menu of the **Funnel Section** field, select an attribute to represent a group of data.

     For example, to configure the funnel chart in the sample, select the `Stage` attribute from the **Funnel Section** field's dropdown menu.

   Figure 38-8 shows the Create Funnel Chart dialog that generates a funnel chart using the `NumberOfVisitors`, `TargetVisitors`, `Stage` attributes in the `SalesConversionFunnel` collection.

Figure 38-8 Create Funnel Chart for SalesConversionFunnel



Create Funnel Chart
Select the data values you want to display for the actual and/or target values of each funnel slice.See Configuring Funnel Charts for examples.
Stage 1
Stage 2
Stage 3
Stage 4
Stage 5
Help
OK
Cancel

Actual Value: NumberOfVisitors

Target Value: TargetVisitors

Funnel Section: Stage

□ Set current row for master-detail

5. Click **OK**.

The example below shows the code generated for the funnel chart when you drag the `SalesConversionFunnel` data collection onto a JSF page and specify `NumOfVisitors` for the **Actual Value** attribute, `TargetVisitors` for the **Target Value** attribute and `Stage` for the **Funnel Section**.

```
<dvt:funnelChart id="funnelChart1" var="row"
         value="#{bindings.SalesConversionFunnel.collectionModel}">
<dvt:funnelDataItem id="di1" value="#{row.NumberOfVisitors}"
                             targetValue="#{row.TargetVisitors}"
                             label="#{row.Stage}"/>
</dvt:funnelChart>
```

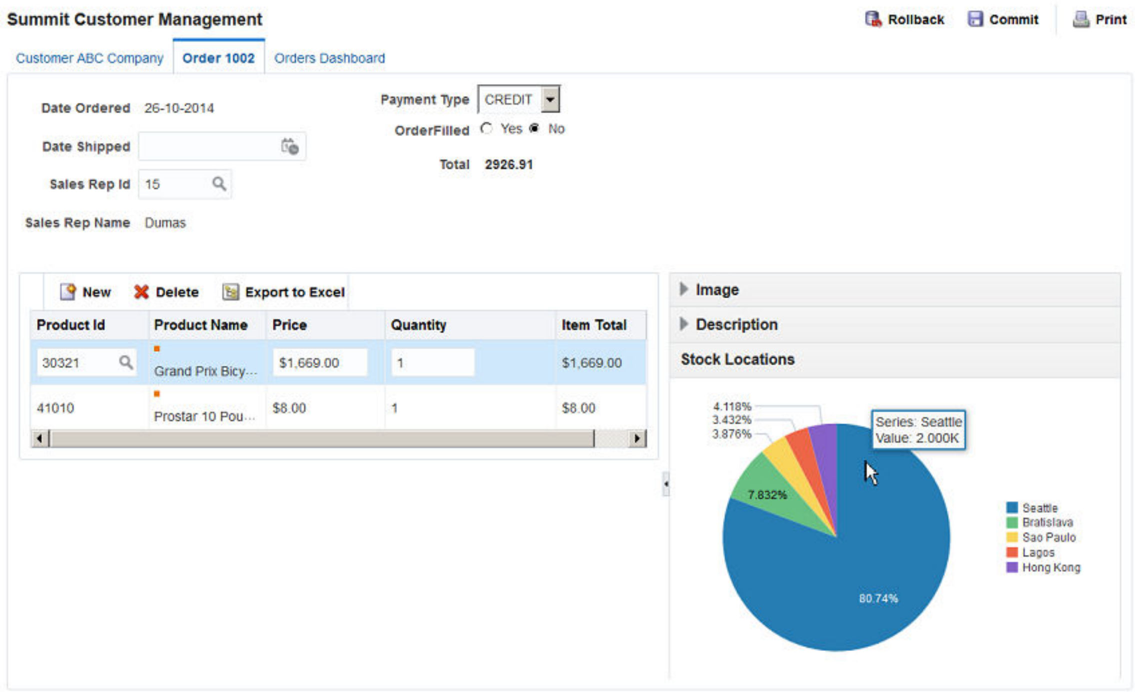
After completing the data binding dialog, you can use the Property Inspector to specify settings for the chart attributes and you can also use the child tags associated with the chart tag to customize the chart further. For more information about configuring chart components, see the Using Chart Components section in the *Developing Web User Interfaces with Oracle ADF Faces*.

How to Create Databound Pie Charts

A pie chart displays one group of data, each slice representing a different series. This chart type requires one column, with multiple rows, one for each slice. You can create a pie chart using the Data Controls panel.

Figure 38-9 shows a pie chart in the Summit ADF sample application that displays the inventory available for the ordered item at each stock location as a percentage of the total available inventory for the item. As the user moves the mouse over each slice, a tooltip shows the actual quantity at the location. The column of data in this example represents the total inventory for the ordered item, and the slices represent the inventory levels at each stocking location.

Figure 38-9 Pie Chart Showing Inventory Levels at Stock Locations



Summit Customer Management

Customer ABC Company | Order 1002 | Orders Dashboard

Date Ordered	26-10-2014	Payment Type	CREDIT
Date Shipped		OrderFilled	○ Yes ● No
Sales Rep Id	15	Total	2926.91
Sales Rep Name	Dumas		

Product Id	Product Name	Price	Quantity	Item Total
30321	Grand Prix Bicy...	$1,669.00	1	$1,669.00
41010	Prostar 10 Pou...	$8.00	1	$8.00

Before you begin:

It may be helpful to have an understanding of databound data visualization charts. For more information, see Creating Databound Charts.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Data Visualization Components.

You will need to complete these tasks:

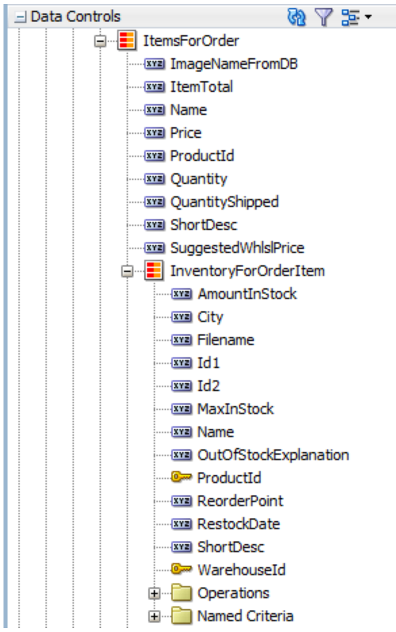
- Create an application module that contains instances of the view objects that you want in your data model, as described in Creating and Modifying an Application Module.
- Create a JSF page as described in the How to Create JSF Pages section of *Developing Web User Interfaces with Oracle ADF Faces*.

To create a databound pie chart:

1. From the Data Controls panel, select a collection.

   For example, to create the pie chart that displays the inventory levels at each stock location in the Summit Customer Management page of the Summit ADF sample application, select the **BackOfficeAppModuleDataControl** > **Customers** > **OrdersForCustomer** > **ItemsForOrder** > **InventoryForOrderItem** collection. Figure 38-10 shows the `InventoryForOrderItem` collection in the Data Controls panel.

Figure 38-10   Data Collection for InventoryForOrderItem Collection



- Data Controls
  - ItemsForOrder
    - ImageNameFromDB
    - ItemTotal
    - Name
    - Price
    - ProductId
    - Quantity
    - QuantityShipped
    - ShortDesc
    - SuggestedWhlslPrice
    - InventoryForOrderItem
      - AmountInStock
      - City
      - Filename
      - Id1
      - Id2
      - MaxInStock
      - Name
      - OutOfStockExplanation
      - ProductId
      - ReorderPoint
      - RestockDate
      - ShortDesc
      - WarehouseId
      - Operations
      - Named Criteria

2. Drag the collection onto a JSF page and, from the context menu, choose **Charts.**
3. In the Component Gallery, select the **Pie** chart category and a chart type and click **OK**.

   For example, select the **Pie** chart category and the **Pie** chart type to create the pie chart in Figure 38-9.
4. Do the following in the Create Pie Chart dialog to configure the pie chart to display data:
   - From the **Pie** field's dropdownx‘ menu, select an attribute to represent the column of data.

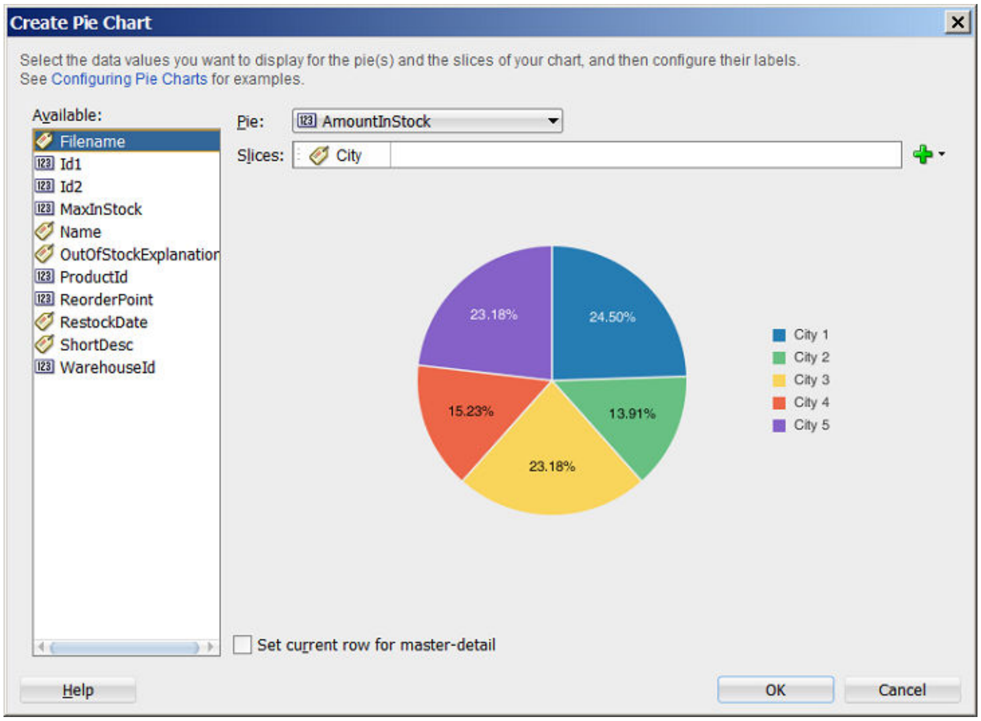
     For example, to configure the pie chart in the Summit ADF sample application to display inventory levels, select the `AmountInStock` attribute from the **Pie** field's dropdown menu.
   - Drag attributes from the **Available** list to the **Slices** field, depending on where you want the values for the attributes to appear at runtime.

     For example, to configure the pie chart in the Summit ADF sample application to display the ordered item's inventory levels at each stock location, drag the `City` attribute from the **Available** list to the **Slices** field in the Create Pie Chart dialog.
   - In the **Attribute Labels** table, accept the default value or select a value from the dropdown list in the **Label** field to specify the label that appears at runtime.

     The underlying data type determines the choices available in the **Label** field. The choice you make determines how that attribute's label is rendered in the chart. For more information, see What You May Need to Know About Using Attribute Labels.

   Figure 38-11 shows the Create Pie Chart dialog that generates a pie chart using the `AmountInStock` and `City` attributes in the `InventoryForOrderItem` collection.

Figure 38-11 Create Pie Chart for InventoryForOrderItem



Create Pie Chart

Select the data values you want to display for the pie(s) and the slices of your chart, and then configure their labels.
See Configuring Pie Charts for examples.

Pie: AmountInStock

Slices: City

☐ Set current row for master-detail

5. Click **OK**.

   The example below shows the code generated for the pie chart when you drag the `InventoryForOrderItem` data collection onto a JSF page and specify `AmountInStock` for the **Pie** value and `City` for the **Slices**.

```
<dvt:pieChart id="pieChart1" var="row"

value="#{bindings.InventoryForOrderItem1.collectionModel}">
  <dvt:chartLegend/>
  <dvt:pieDataItem id="di1" value="#{row.AmountInStock}"
label="#{row.City}"/>
</dvt:pieChart>
```

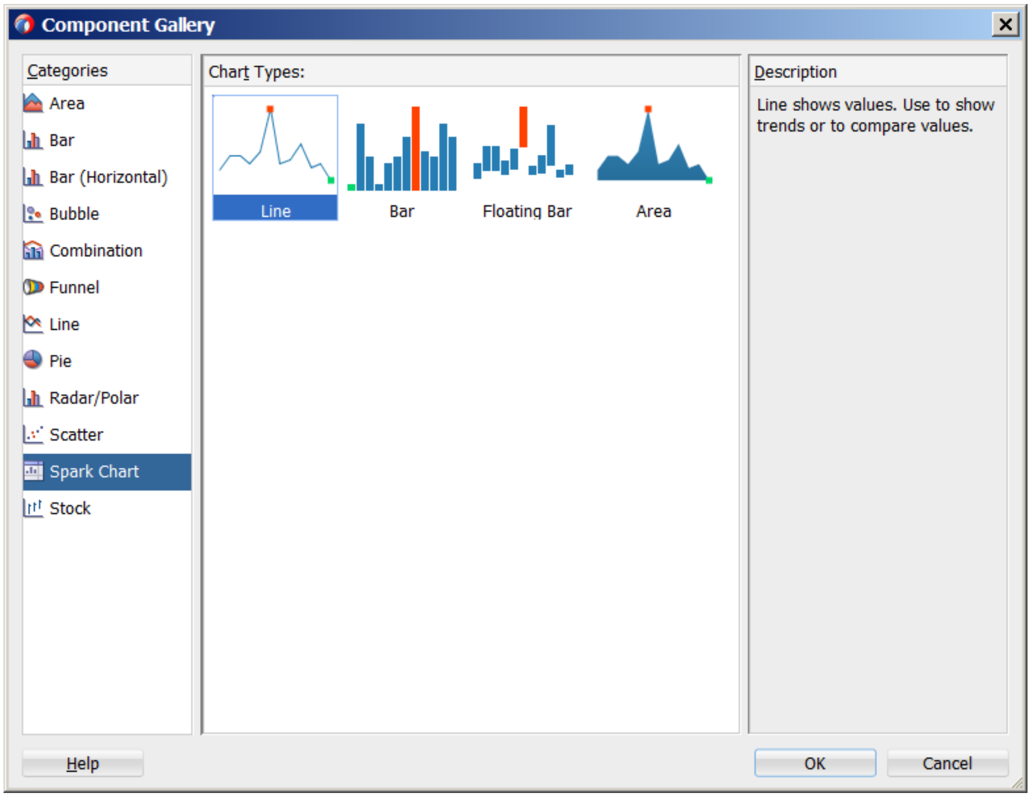
After completing the data binding dialog, you can use the Properties window to specify settings for the chart attributes and you can also use the child tags associated with the chart tag to customize the chart further. For example, you can configure an exploding pie slice which causes one slice of the pie to appear separated from the other slices. For more information, see Using Chart Components in *Developing Web User Interfaces with Oracle ADF Faces*.

Creating a Databound Spark Chart Using Data Controls

Spark charts are simple, condensed charts that display trends or variations, often in the column of a table, or inline with text. You can create a sparkchart by inserting a data control from the Data Controls Panel.

Figure 38-12 shows the Component Gallery that displays when you drag a spark chart onto your page from the Data Controls panel.

Figure 38-12 Create Spark Chart Component Gallery



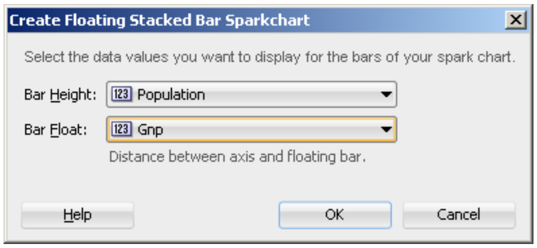
Line shows values. Use to show trends or to compare values.

A binding dialog prompts you to specify the value you wish to display for the selected spark chart type. Line, bar, and area spark charts require a single series of data values, for example the changing value of a stock. Floating bar spark charts require two series of data values, one for the float offset, and one for the bar value. For example, in the Create Floating Stacked Bar Sparkchart dialog you specify:

- **Bar Height**: Use to select the data value to use for the bar value.
- **Bar Float**: Use to select the data value to use for the float offset, the distance between the axis and the floating bar.

Figure 38-13 shows a completed Create Floating Stacked Bar Sparkchart dialog.

Figure 38-13 Sparkchart Binding Dialog



Select the data values you want to display for the bars of your spark chart.

Bar Height: Population

Bar Float: Gnp

Distance between axis and floating bar.

In a simple UI-first development scenario you can insert a sparkchart using the Components window and bind it to data afterwards. For additional information about providing data to spark charts, see How to Add Data to Spark Charts in *Developing Web User Interfaces with Oracle ADF Faces*.

How to Create Databound Bubble and Scatter Charts

Bubble and scatter charts represent data by the location of the data marker. Bubble charts also use the size of the markers to represent an additional aspect of the data.

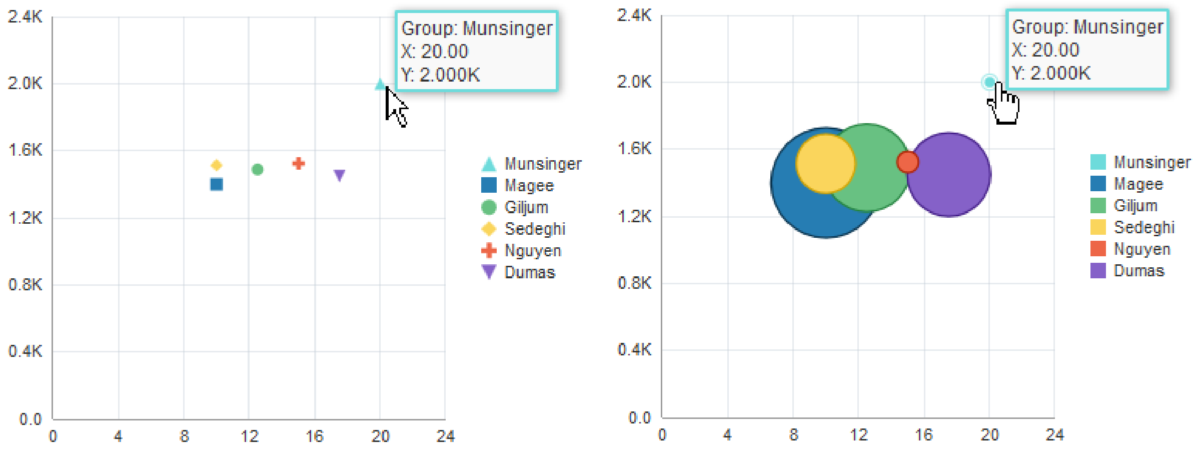
Scatter charts require at least two data values for each marker. Bubble charts require an additional data value for the marker size. Each data marker represents the following:

- The x value that determines the marker's location along the x-axis.
- The y value that determines the marker's location along the y-axis.
- The z value that determines the size of the marker on bubble charts

For more than one group of data, the data must be in multiples of two for scatter charts and three for bubble charts.

Figure 38-14 shows a scatter and a bubble chart displaying salary and commission for salesmen in the Summit organization. In this example, the x value is determined by the commission, and the y value is determined by the salary. The bubble chart is configured to use the salesmen's total sales for the z value. In this example, Magee has the lowest salary and commission but is the salesman with the highest sales total.

Figure 38-14 Databound Scatter and Bubble Chart Examples



Before you begin:

It may be helpful to have an understanding of databound data visualization charts. For more information, see Creating Databound Charts.

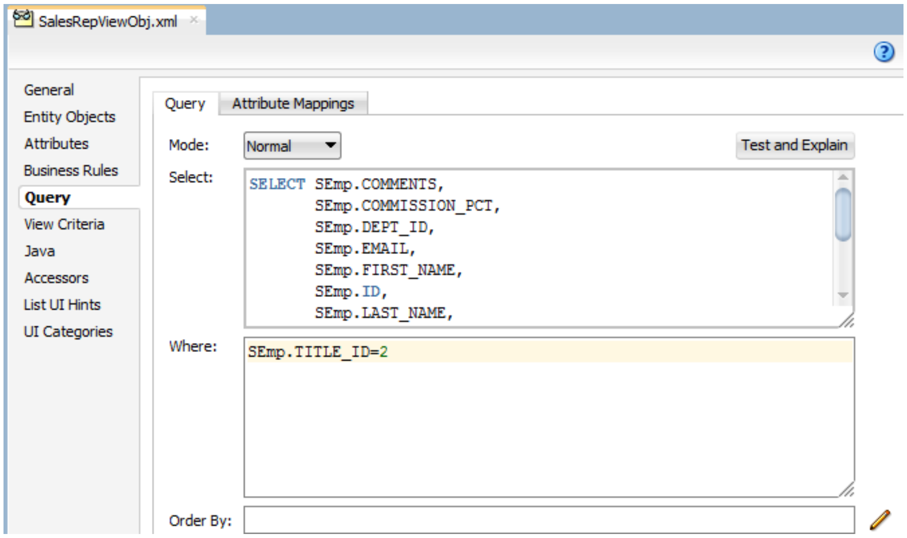
You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Data Visualization Components.

You will need to complete these tasks:

- Create an application module that contains instances of the view objects that you want in your data model, as described in Creating and Modifying an Application Module.

  For example, the bubble and scatter charts in Figure 38-14 in uses a top level view object based on the `SEmp` entity in the Summit ADF DVT sample application. In this example, the view object retrieves all sales representatives in the Summit ADF employee database by retrieving all employee records with a `TITLE_ID` of `2`. Figure 38-15 shows the `SalesRepViewObj` view object.

Figure 38-15 SalesRepViewObj View Object in Summit DVT Sample Application



Mode: Normal

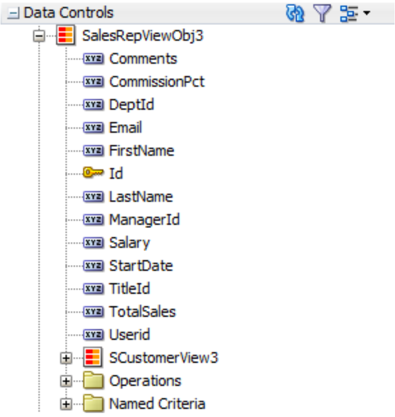
Select:
```
SELECT SEmp.COMMENTS,
       SEmp.COMMISSION_PCT,
       SEmp.DEPT_ID,
       SEmp.EMAIL,
       SEmp.FIRST_NAME,
       SEmp.ID,
       SEmp.LAST_NAME,
```

Where:
```
SEmp.TITLE_ID=2
```

Order By:

Figure 38-16 shows the `SalesRepViewObj3` data control.

Figure 38-16 SalesRepViewObj3 Data Control in Summit DVT Sample Application



```
Data Controls
  SalesRepViewObj3
    Comments
    CommissionPct
    DeptId
    Email
    FirstName
    Id
    LastName
    ManagerId
    Salary
    StartDate
    TitleId
    TotalSales
    Userid
    SCustomerView3
    Operations
    Named Criteria
```

The `TotalSales` attribute is a transient attribute that calculates the salesman's total sales from the `Total` attribute in the `SOrdView` view object: `SOrdView.sum`

('Total'). For information about adding transient attributes to view objects, see Adding Calculated and Transient Attributes to a View Object.

- Create a JSF page as described in the How to Create JSF Pages section of *Developing Web User Interfaces with Oracle ADF Faces*.

To create a databound bubble or scatter chart:

1. From the Data Controls panel, select a collection.

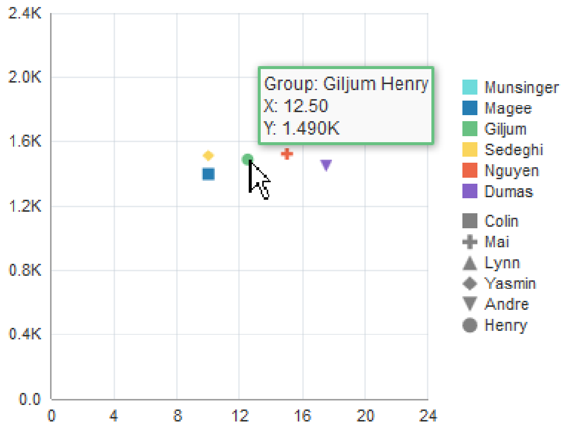
   For example, to create the bubble or scatter chart shown in Figure 38-14, select the SalesRepViewObj3 collection.
2. Drag the collection onto a JSF page and, from the context menu, choose **Charts.**
3. In the Component Gallery, select the bubble or scatter chart category and a chart type and click **OK**.
4. In the Create Bubble Chart or Create Scatter Chart dialog, do the following in the dialog to configure the bubble or scatter chart axis values:
   - In the **X Axis Data Point** field, use the dropdown list to specify the attribute in the data collection to use for the x-axis data point.

     For example, to configure the bubble or scatter chart to display the commission percent along the X Axis, select **CommissionPct** from the dropdown list.
   - In the **Y Axis Data Point** field, use the dropdown list to specify the attribute in the data collection to use for the y-axis data point.
   - For bubble charts, in the **Z Axis Data Point** field, use the dropdown list to specify the attribute in the data collection to use for the z-axis data point.
5. If you are configuring a scatter chart, in the Markers section, do the following in the dialog to configure the scatter chart's color, shape, and tooltip:
   - Select **Automatically** to use system-generated values for color/shape combinations. Select **Manually** to use distinct attributes to determine color and shape.

     If you select **Manually**, select the attributes in the dropdown list for **Marker Color** and **Marker Shape**.

     For example, you could select LastName and FirstName to be used for Marker Color and Marker Shape for the scatter chart displayed in Figure 38-14. When the scatter chart is rendered, the legend will contain entries for both the color and shape as shown in Figure 38-17. The scatter chart's tooltip displays the value associated with both the marker and shape attributes.

Figure 38-17 Scatter Chart Configured Manually for Color and Shape



- From the **Marker Type** field's dropdown list, specify one or more attributes in the data collection that determine the values that appear in the legend of the scatter chart. Each attribute will be represented by its own color/shape combination in the scatter chart's legend. You can drag and drop attributes from the **Available** list, or select from the **Add** dropdown list.

  After specifying attributes for the input field, you can right click on any attribute to display a context menu for actions such as **Move Right**, **Move Left**, **Delete**, or **Treat as Text**.

- From the **Marker Tooltip** field's dropdown list, select the attribute to use for the scatter chart's tooltip.

6. If you are configuring a bubble chart, in the Markers section, do the following in the dialog to configure the bubble colors and tooltip:

   - Select **Automatically** to use system-generated values for color. Select **Manually** to use distinct attributes to determine color and shape.

     If you select **Manually**, select the attribute in the dropdown list for **Marker Color** and **Marker Shape**.

   - If you selected **Automatically**, from the **Bubble Color** field's dropdown list, specify one or more attributes in the data collection that determine the values that appear in the legend of the bubble chart. Each attribute will be represented by its own color in the bubble chart's legend. You can drag and drop attributes from the **Available** list, or select from the **Add** dropdown list.

     After specifying attributes for the input field, you can right click on any attribute to display a context menu for actions such as **Move Right**, **Move Left**, **Delete**, or **Treat as Text**.

   - From the **Bubble Tooltip** field's dropdown list, select the attribute to use for the bubble chart's tooltip.

     By default, the bubble chart's tooltip will display the attribute or attributes associated with the bubble chart's legend. For example, the bubble chart's tooltip in Figure 38-14 shows the last name of the salesman for the Group. You can select a different attribute in the collection to be used for the Group item in the tooltip.

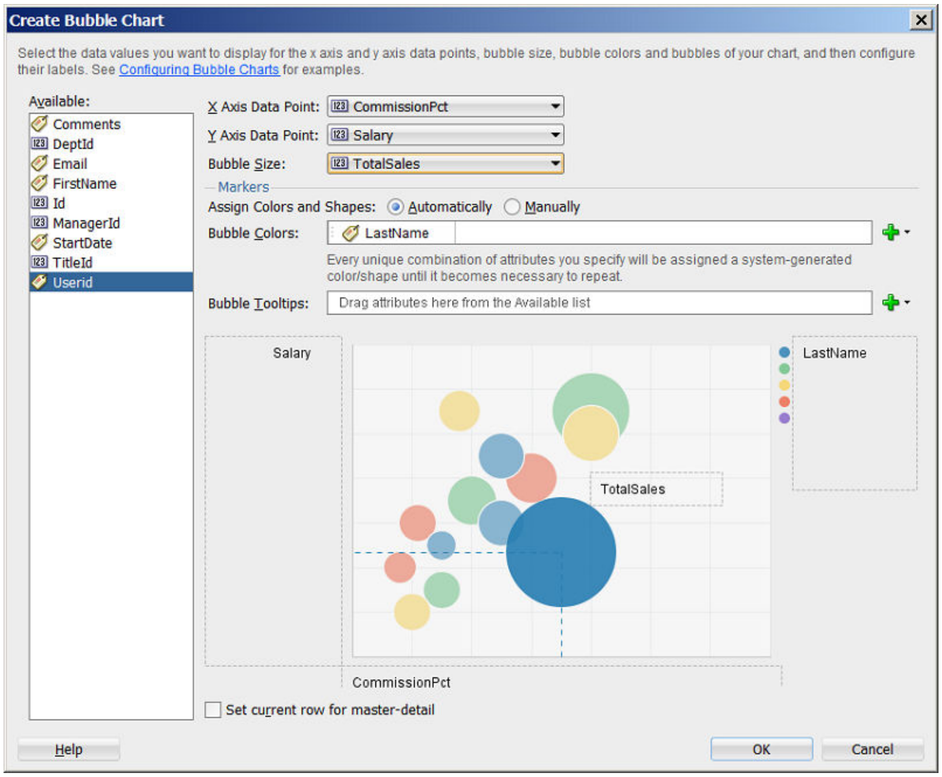
7. In the **Attribute Labels** table, accept the default value or select a value from the dropdown list in the **Label** field to specify the label that appears at runtime.

The underlying data type determines the choices available in the **Label** field. The choice you make determines how that attribute's label is rendered in the chart. For more information, see What You May Need to Know About Using Attribute Labels.

8. Click **OK**.

Figure 38-18 shows the Create Bubble Chart dialog that generates a bubble chart using data from the `CommissionPct`, `Salary`, and `Total Sales` attributes in the `SalesRepViewObj3` data collection.

Figure 38-18 Create Bubble Chart Dialog



Create Bubble Chart

Select the data values you want to display for the x axis and y axis data points, bubble size, bubble colors and bubbles of your chart, and then configure their labels. See Configuring Bubble Charts for examples.

Available:
- Comments
- DeptId
- Email
- FirstName
- Id
- ManagerId
- StartDate
- TitleId
- Userid

X Axis Data Point: CommissionPct

Y Axis Data Point: Salary

Bubble Size: TotalSales

Markers

Assign Colors and Shapes: ● Automatically ○ Manually

Bubble Colors: LastName

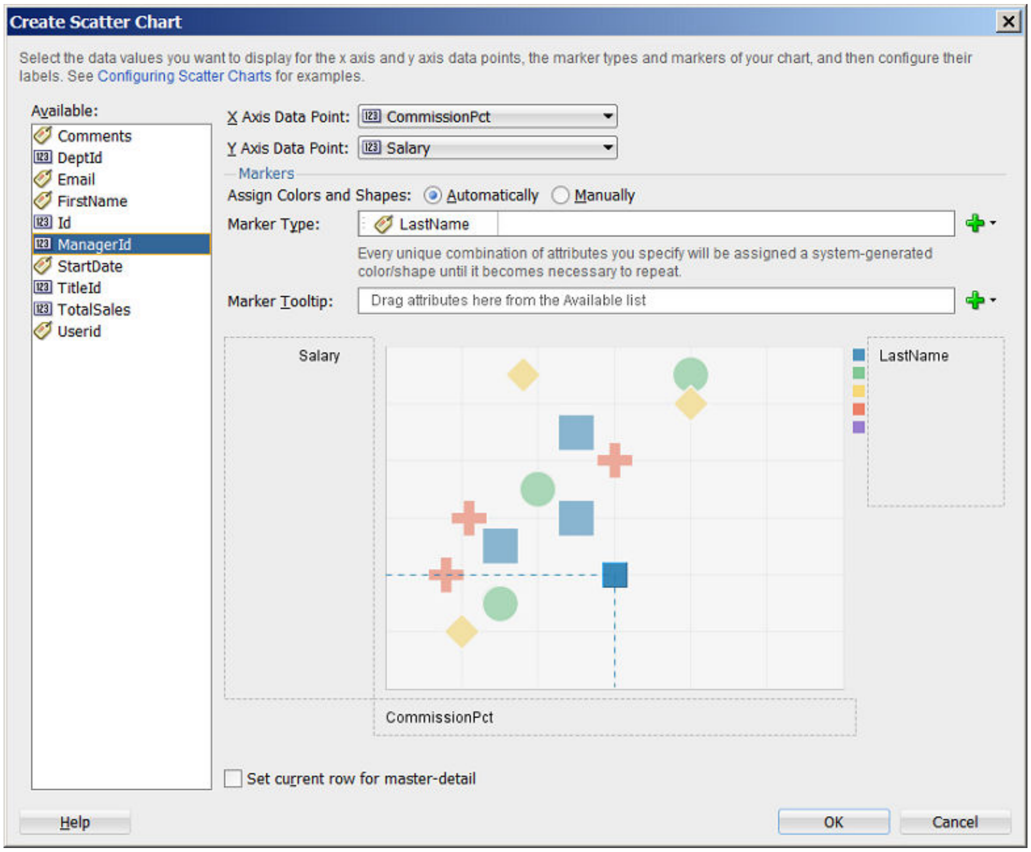
Every unique combination of attributes you specify will be assigned a system-generated color/shape until it becomes necessary to repeat.

Bubble Tooltips: Drag attributes here from the Available list

□ Set current row for master-detail

The entries for the Create Scatter Chart dialog are similar, but you would not need to select a bubble size. Figure 38-19 shows the Create Scatter Chart dialog that generates a scatter chart using data from the `CommissionPct` and `Salary` attributes in the `SalesRepViewObj3` data collection.

Figure 38-19 Create Scatter Chart Dialog



Create Scatter Chart

Select the data values you want to display for the x axis and y axis data points, the marker types and markers of your chart, and then configure their labels. See Configuring Scatter Charts for examples.

Available:
- Comments
- DeptId
- Email
- FirstName
- Id
- ManagerId
- StartDate
- TitleId
- TotalSales
- Userid

X Axis Data Point: CommissionPct

Y Axis Data Point: Salary

Markers

Assign Colors and Shapes: ● Automatically ○ Manually

Marker Type: LastName

Every unique combination of attributes you specify will be assigned a system-generated color/shape until it becomes necessary to repeat.

Marker Tooltip: Drag attributes here from the Available list

□ Set current row for master-detail

The example below shows the code generated for the bubble and scatter charts when you drag the SalesRepViewObj3 data collection onto a JSF page and specify CommissionPct for the **X Axis Data Point**, Salary for the **Y Axis Data Point**, and TotalSales for the bubble chart's **Z Axis Data Point**.

```
<dvt:scatterChart id="scatterChart1" var="row"
                  value="#{bindings.SalesRepViewObj3.collectionModel}">
  <dvt:chartLegend rendered="true" id="cl1">
    <dvt:legendSection source="ag1" id="ls1"/>
  </dvt:chartLegend>
  <f:facet name="dataStamp">
    <dvt:chartDataItem id="di1" group="#{row.LastName}"
x="#{row.CommissionPct}"
                       y="#{row.Salary}">
      <dvt:attributeGroups id="ag1" value="#{row.LastName}" type="color
shape"/>
    </dvt:chartDataItem>
  </f:facet>
</dvt:scatterChart>
...
<dvt:bubbleChart id="bubbleChart1" var="row"
                 value="#{bindings.SalesRepViewObj3.collectionModel}">
  <dvt:chartLegend rendered="true" id="cl1">
    <dvt:legendSection source="ag1" id="ls1"/>
  </dvt:chartLegend>
  <f:facet name="dataStamp">
    <dvt:chartDataItem id="di1" group="#{row.LastName}"
x="#{row.CommissionPct}"
                       y="#{row.Salary}" z="#{row.TotalSales}">
      <dvt:attributeGroups id="ag1" value="#{row.LastName}"
type="color"/>
    </dvt:chartDataItem>
  </f:facet>
</dvt:bubbleChart>
```

If you chose to configure the scatter chart manually using attributes from the data collection for marker and shape, JDeveloper adds an additional `dvt:legendSection` and `dvt:attributeGroups` tag to the JSF page. The example below shows the revised code, with the modified entries highlighted in bold.

```
<dvt:scatterChart id="scatterChart1" var="row"
                  value="#{bindings.SalesRepViewObj3.collectionModel}">
  <dvt:chartLegend rendered="true" id="cl1">
    <dvt:legendSection source="ag1" id="ls1"/>
    <dvt:legendSection source="ag2" id="ls2"/>
  </dvt:chartLegend>
  <f:facet name="dataStamp">
    <dvt:chartDataItem id="di1" group="#{row.LastName} #{row.FirstName}"
                       x="#{row.CommissionPct}"
                       y="#{row.Salary}">
      <dvt:attributeGroups id="ag1" value="#{row.LastName}"
type="color"/>
      <dvt:attributeGroups id="ag2" value="#{row.FirstName}"
type="shape"/>
    </dvt:chartDataItem>
  </f:facet>
</dvt:scatterChart>
```

After completing the data binding dialog, you can use the Properties window to specify settings for the chart attributes, and you can also use the child tags associated with the chart tag to customize the chart further. For example, you can specify where you want the chart's legend to appear. For more information, see Using Chart Components in *Developing Web User Interfaces with Oracle ADF Faces*.

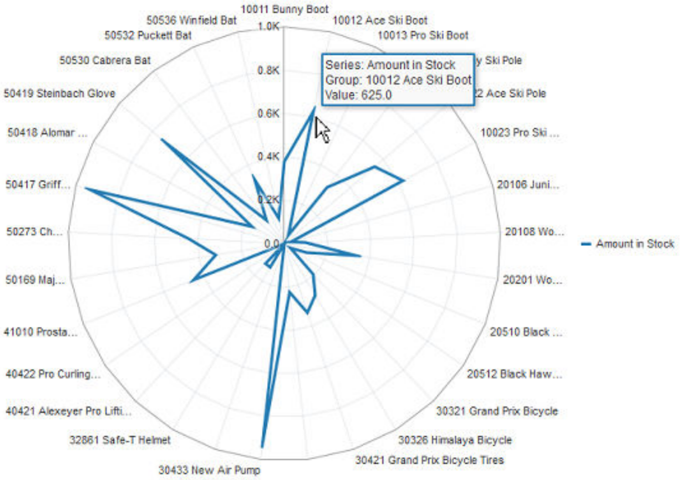
How to Create Databound Polar and Radar Charts

Polar and Radar charts are variations of the line, bar, area, bubble and scatter charts. You can create a polar or radar chart using the Data Controls panel.

The requirements of polar and radar charts vary based on the type of chart used. Radar line, radar area, and polar bar charts require at least two groups of data, with one or more series. Polar scatter charts require at least two data values to indicate position, while polar bubble charts require three values - two for position and one for bubble size.

Figure 38-20 shows a sample polar line chart, configured to show the available inventory levels of different products against the product ID and name.

Figure 38-20   Sample Polar Bar Chart



Before you begin:

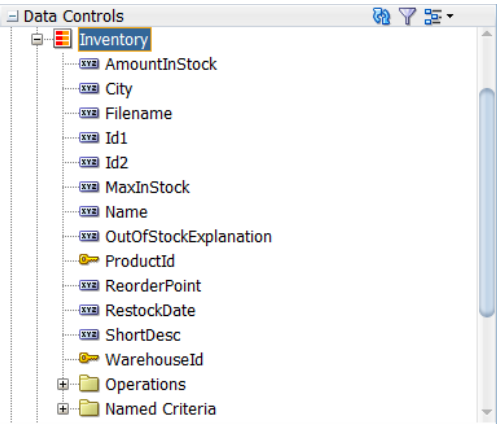
It may be helpful to have an understanding of databound data visualization charts. For more information, see Creating Databound Charts.

You will need to complete these tasks:

- Create an **application module** that contains instances of the **view objects** that you want in your data model, as described in Creating and Modifying an Application Module.

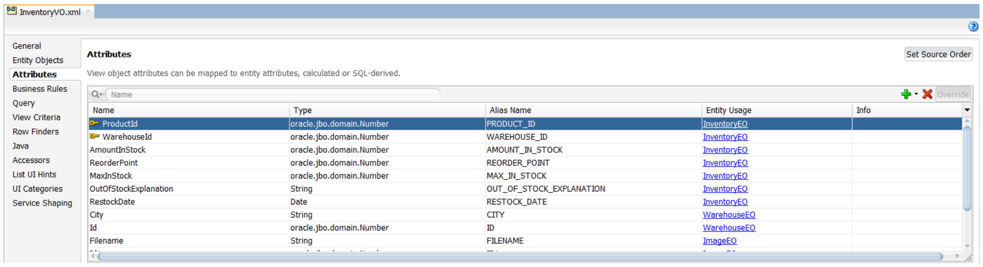
  For example, the sample polar bar chart is based on the `Inventory` data collection. Figure 38-21 shows the `Inventory` collection in the Data Controls panel.

Figure 38-21   Data Collection for Inventory



The Inventory data collection is based on the `InventoryVO` view object. Figure 38-22 shows the `InventoryVO` view object.

Figure 38-22 InventoryVO View Object in Summit Sample Application



Attributes

View object attributes can be mapped to entity attributes, calculated or SQL-derived.

Name	Type	Alias Name	Entity Usage	Info
ProductId	oracle.jbo.domain.Number	PRODUCT_ID	InventoryEO	
WarehouseId	oracle.jbo.domain.Number	WAREHOUSE_ID	InventoryEO	
AmountInStock	oracle.jbo.domain.Number	AMOUNT_IN_STOCK	InventoryEO	
ReorderPoint	oracle.jbo.domain.Number	REORDER_POINT	InventoryEO	
MaxInStock	oracle.jbo.domain.Number	MAX_IN_STOCK	InventoryEO	
OutOfStockExplanation	String	OUT_OF_STOCK_EXPLANATION	InventoryEO	
RestockDate	Date	RESTOCK_DATE	InventoryEO	
City	String	CITY	WarehouseEO	
Id	oracle.jbo.domain.Number	ID	WarehouseEO	
Filename	String	FILENAME	ImageEO	

- Create a JSF page as described in the How to Create JSF JSP Pages section of the *Developing Web User Interfaces with Oracle ADF Faces*.

To create a databound radar or polar chart:

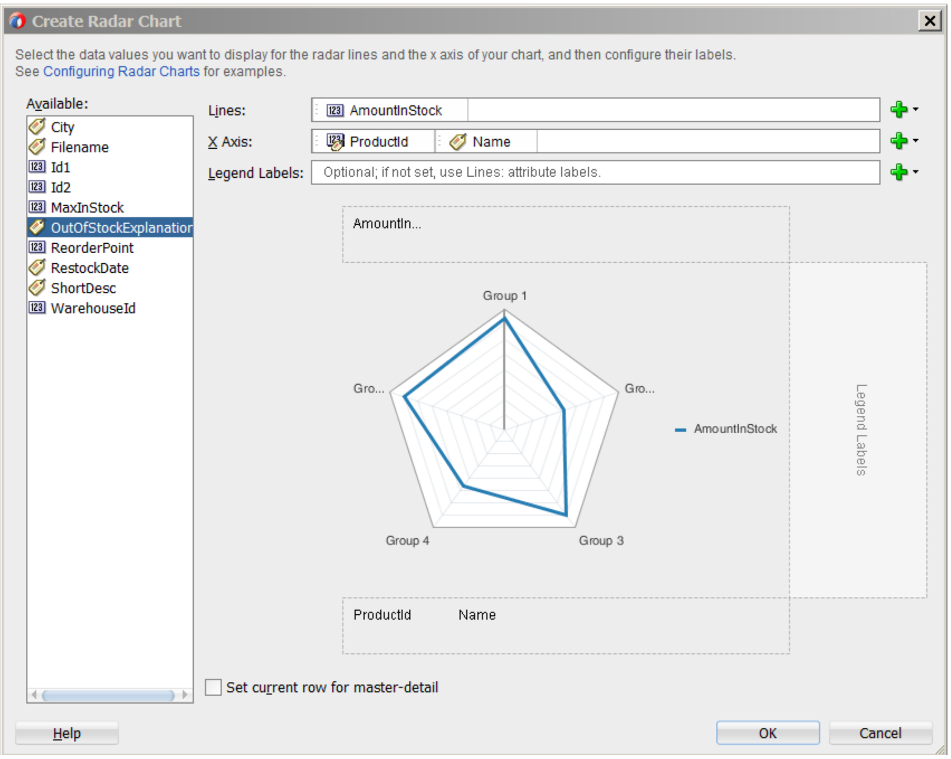
1. From the Data Controls panel, select a collection.
2. Drag the collection onto a JSF page and, from the context menu, choose **Charts.**
3. In the Component Gallery, select the Radar/Polar chart category and a chart type and click **OK**.

   For example, select the **Radar/Polar** chart category and the **Radar** chart type to create a radar line chart.
4. If configuring a polar line, area or bar chart, do the following:
   - Drag attributes from the **Available** list to the input fields, depending on how you want the values for the attributes to appear at runtime.

     For example, to configure the polar bar chart in the sample application to display the inventory levels, drag the `AmountInStock` attribute from the **Available** list to the **Bars** field in the Create Bar Chart dialog. To configure the chart to display the product ID and name, drag the `ProductId` and `Name` attributes from the **Available** list to the **X Axis** field in the Create Bar Chart dialog.
5. Click **OK**.

Figure 38-23 shows the Create Radar Chart dialog that generates a polar line chart using data from the `AmountInStock`, `ProductId`, and `Name` attributes in the `Inventory` data collection.

Figure 38-23   Create Radar Chart Dialog



Create Radar Chart

Select the data values you want to display for the radar lines and the x axis of your chart, and then configure their labels.
See Configuring Radar Charts for examples.

Available:
- City
- Filename
- Id1
- Id2
- MaxInStock
- OutOfStockExplanation
- ReorderPoint
- RestockDate
- ShortDesc
- WarehouseId

Lines: AmountInStock

X Axis: ProductId Name

Legend Labels: Optional; if not set, use Lines: attribute labels.

□ Set current row for master-detail

The example below shows the code generated for the bubble and scatter charts when you drag the `Inventory` data collection onto a JSF page and specify `AmountInStock` for the **Y Axis Data Point**, and `ProductId` and `Name` for the **Y Axis Data Point**. Notice that radar and polar charts do not have a separate tag, but are rather basic charts with the `coordinateSystem` attribute set to `polar`.

```
<dvt:lineChart coordinateSystem="polar" polarGridShape="polygon"
id="lineChart1" var="row" value="#{bindings.Inventory.collectionModel}">
 <f:facet name="dataStamp">
  <dvt:chartDataItem id="di1"
series="#{bindings.Inventory.hints.AmountInStock.label}"
value="#{row.AmountInStock}" group="#{row.ProductId} #{row.Name}"/>
 </f:facet>
</dvt:lineChart>
```

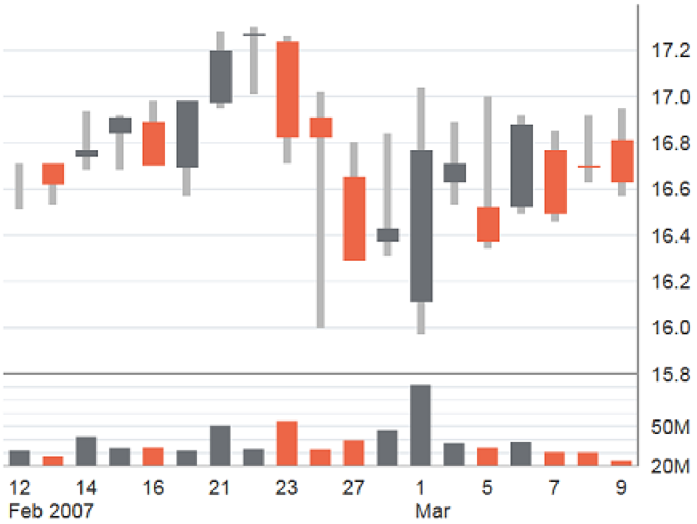
After completing the data binding dialog, you can use the Property Inspector to specify settings for the chart attributes and you can also use the child tags associated with the chart tag to customize the chart further. For more information about configuring chart components, see the Using Chart Components section in the *Developing Web User Interfaces with Oracle ADF Faces*.

How to Create Databound Stock Charts

Stock charts are used to track the period milestone values of trade stocks, such as the high, low, opening and closing prices of a stock. You can create a stock chart by using the Data Controls panel.

Depending on the type of chart chosen, stock charts require a minimum of four values to a maximum of seven values. Figure 38-24 shows a sample stock chart configured to show stocks and their opening value, closing value, high value, low value, time period, and trade volume. As the user moves over each stock, a tooltip provides the numerical quantity of all values for that stock.

Figure 38-24 Sample Stock Chart



Before you begin:

It may be helpful to have an understanding of databound data visualization charts. For more information, see Creating Databound Charts.

You will need to complete these tasks:

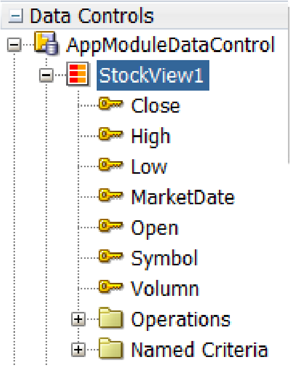
- Create an **application module** that contains instances of the **view objects** that you want in your data model, as described in Creating and Modifying an Application Module.
- Create a JSF page as described in the How to Create JSF Pages section of the *Developing Web User Interfaces with Oracle ADF Faces*.

To create a databound stock chart:

1. From the Data Controls panel, select a collection.

   For example, the stock chart displayed above is made using the `StockView1` collection. Figure 38-25 shows the `StockView1` collection in the Data Controls panel.

Figure 38-25 Data Control for StockView1 Collection



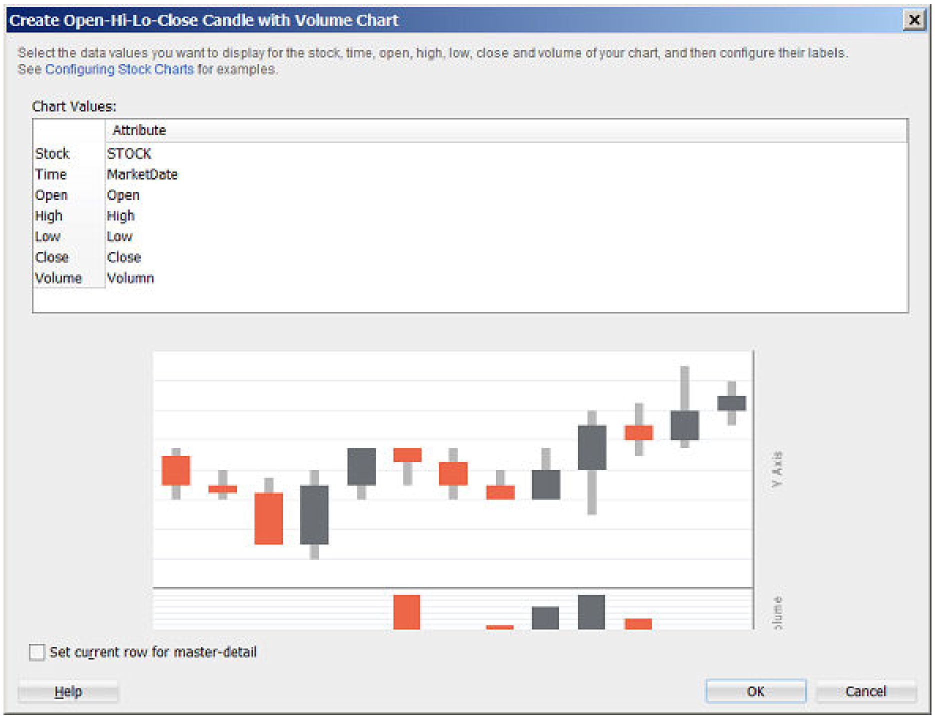
- Data Controls
  - AppModuleDataControl
    - StockView1
      - Close
      - High
      - Low
      - MarketDate
      - Open
      - Symbol
      - Volumn
      - Operations
      - Named Criteria

2. Drag the collection onto a JSF page and, from the context menu, choose **Charts**.
3. In the Component Gallery, select the Stock chart category and a chart type and click **OK**.

   For example, to create the chart configured in the sample, select the Stock chart category and the Open-Hi-Lo-Close Candle with Volume chart type.
4. Do the following in the dialog to configure the chart to display data:
   - From the **Stock** field's dropdown menu, select an attribute to represent the name of the stock. You may use the **Select Text Resource** option to enter a text value or use the **Expression Builder** to provide a JSF expression.
   - Depending on the type of stock chart chosen, a selection of labels will be shown. For each label, select an attribute from that label's dropdown menu. For example, to configure the chart from the example above, select the `Open`, `High`, `Low`, `Close`, `Volume`, and `MarketDate` attributes for the **Open**, **High**, **Low**, **Close**, **Volume**, and **Time** labels respectively.

     Figure 38-26 shows the Create Chart Dialog for the stock chart configured in the sample above.

Figure 38-26 Create Chart Dialog for StockView1 Collection



Create Open-Hi-Lo-Close Candle with Volume Chart

Select the data values you want to display for the stock, time, open, high, low, close and volume of your chart, and then configure their labels. See Configuring Stock Charts for examples.

Chart Values:

	Attribute
Stock	STOCK
Time	MarketDate
Open	Open
High	High
Low	Low
Close	Close
Volume	Volumn

☐ Set current row for master-detail

5. Click **OK**.

The example below shows the code generated for the stock chart for the sample above when you drag the Stock data collection onto a JSF page and specify the given values.

```
<dvt:stockChart id="stockChart1" var="row"
value="#{bindings.StockView1.collectionModel}">
    <dvt:chartY2Axis id="cya1"/>
    <dvt:stockDataItem id="di1" volume="#{row.Volumn}"
high="#{row.High}" low="#{row.Low}" series="Stock"
                              close="#{row.Close}" open="#{row.Open}"
group="#{row.MarketDate}"/>
</dvt:stockChart>
```

After completing the data binding dialog, you can use the Property Inspector to specify settings for the chart attributes and you can also use the child tags associated with the chart tag to customize the chart further. For more information about configuring chart components, see the Using Chart Components section in the *Developing Web User Interfaces with Oracle ADF Faces*.

What You May Need to Know About Using Attribute Labels

When you configure attribute labels in the Create Chart dialog, the underlying data type determines the choices available in the **Label** field. The choice you make determines how that attribute's label is rendered in the chart.

If an attribute represents data values, then the choices in the **Label** field are:

- **Use Attribute Name**: Select to render the value as a string using the label from the `UIHints` for that attribute in the underlying `ViewObject`. This is the default selection.
- **No Label**: Select to render no label. This choice is useful if there is a single metric and you want to provide your own descriptive text on the page to describe the resulting chart.
- **Select Text Resource**: Select to open a Select Text Resource dialog to select or add a text resource to use for the label. The text resource is a translatable string from an application resource bundle. If you need help, press F1 or click **Help.**
- **Expression Builder**: Select to open the Expression Builder dialog to create an expression to be executed at runtime for the label. If you need help, press F1 or click **Help**.

If the attribute represents a category of data values, then the choices are:

- **Use Attribute Value**: Select to render the attribute values as category labels. This is the default selection.
- From the dropdown list, choose an alternate attribute for the label. For example, use **Employee Names** for labels instead of **Employee IDs**.

Creating Databound Gauges

A gauge plots a single data value, such as a sales total, stock level, temperature, or speed. Using thresholds, gauges can show state information such as acceptable or unacceptable ranges using color.

For example, a gauge value axis might show ranges colored red, yellow, and green to represent low, medium, and high states. One databound gauge component can create a single gauge or a column of gauges, depending on the number of rows in the data collection used. In a data collection, each row contains the value for a single gauge.

The Component Gallery for gauges allows you to choose from four gauge categories.

Table 38-2 ADF Gauge Categories in the Component Gallery

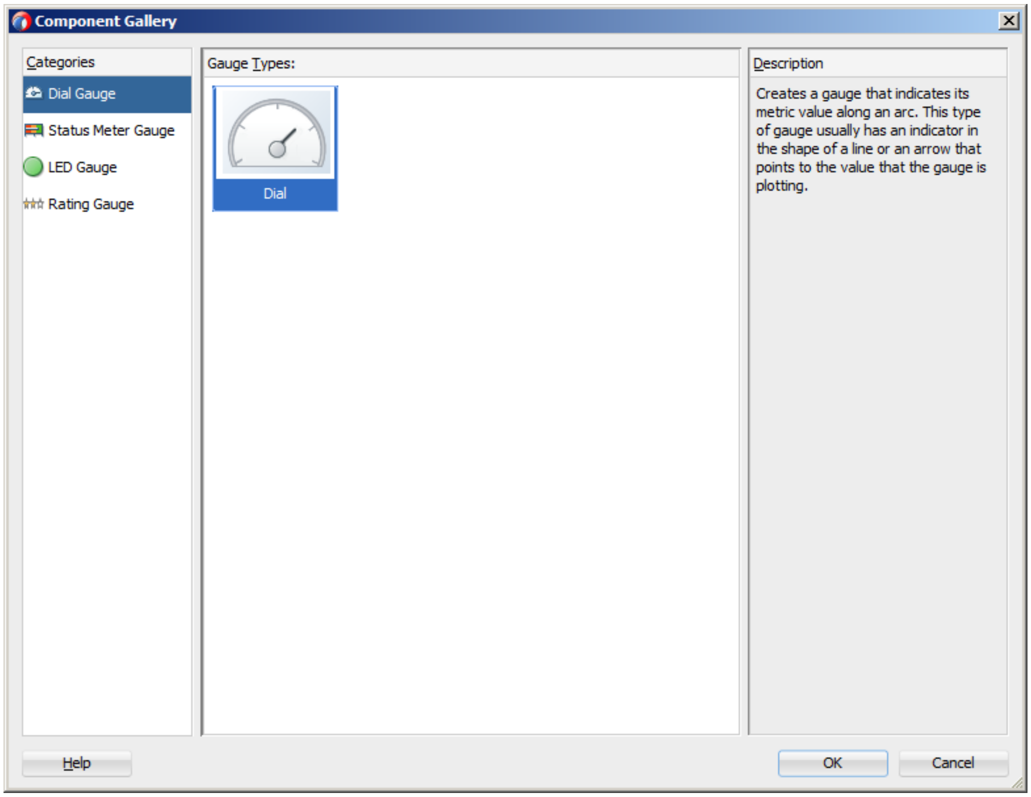
Image	Category	Description
	Dial	Displays a metric value plotted on a circular axis. The gauge's background attribute determines whether the gauge's background is displayed as a rectangle, circle, or semicircle. An indicator points to the dial gauge's metric value on the axis.
	LED	Graphically depicts a measurement, such as a key performance indicator (KPI). Several styles of shapes are available for LED gauges, including round or rectangular shapes that use color to indicate status, and triangles or arrows that point up, left, right, or down in addition to the color indicator.

Table 38-2 (Cont.) ADF Gauge Categories in the Component Gallery

Image	Category	Description
★★★ Rating Gauge	Rating	Displays and optionally accepts input for a metric value. This gauge is typically used to show ratings for products or services, such as the star rating for a movie.
Status Meter Gauge	Status Meter	Displays the metric value on a horizontal or circular axis. An inner rectangle shows the current level of a measurement against the ranges marked on an outer rectangle. Optionally, status meters can display colors to indicate where the metric value falls within predefined thresholds.

Each category contains one or more gauge types. Explore the Component Gallery that appears when you create a single gauge to view all available gauge and category types, and descriptions for each one. Figure 38-27 shows the Component Gallery that appears for ADF gauges.

Figure 38-27 ADF Gauges Component Gallery



Creates a gauge that indicates its metric value along an arc. This type of gauge usually has an indicator in the shape of a line or an arrow that points to the value that the gauge is plotting.

The data binding process is essentially the same regardless of which type of gauge you create. Only the metric value (that is, the measurement that the gauge is to indicate) is required. However, if a row in a data collection contains range information such as maximum, minimum, and thresholds, then these values can be bound to the gauge to provide dynamic settings. If information that you want to use in a gauge's upper or lower labels is available in the data collection, then you can bind these values to the gauge as well.

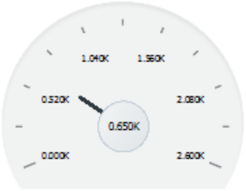
For information about customizing a gauge after the data binding is completed, see Using Gauge Components in *Developing Web User Interfaces with Oracle ADF Faces*.

How to Create a Databound Dial Gauge

You can use the ADF gauge component to create a dial gauge against a circle, dome, or rectangle background. The gauge's indicator specifies the current value of the metric.

Figure 38-28 shows a single dial gauge that appears if you create a gauge from the `AmountInStock` data for inventory items in the Summit ADF DVT sample application. The value of the `AmountInStock` metric, which is 0.650K, appears in a label in the center of the gauge

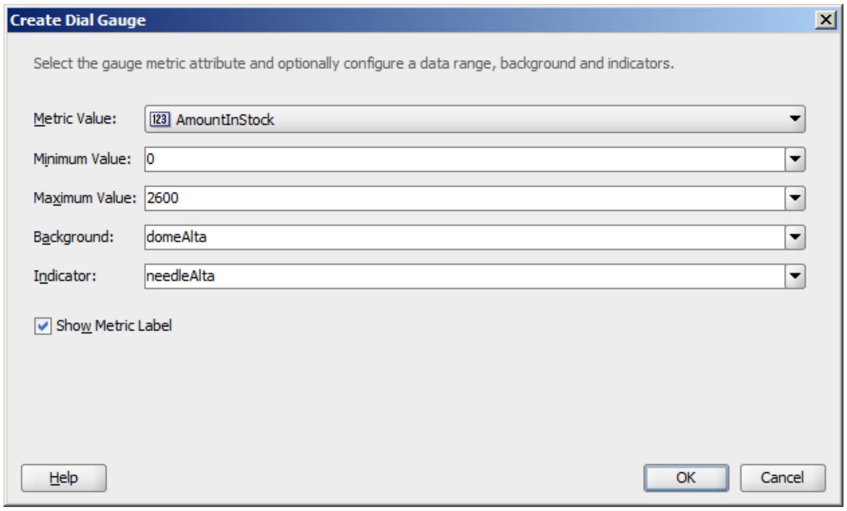
Figure 38-28 The Amount in Stock Dial Gauge



To create a dial gauge using a data control, you bind the gauge component to an attribute in a data collection. JDeveloper allows you to do this declaratively by dragging and dropping an attribute from the Data Controls panel. After you drag and drop the attribute, use the Create Dial Gauge dialog to configure the gauge.

Figure 38-29 shows the Create Dial Gauge dialog configured for the Amount in Stock dial gauge shown in Figure 38-28 and using the gauge type shown in Figure 38-27.

Figure 38-29   Create Dial Gauge Dialog for Amount in Stock Dial Gauge



Select the gauge metric attribute and optionally configure a data range, background and indicators.

Metric Value: AmountInStock
Minimum Value: 0
Maximum Value: 2600
Background: domeAlta
Indicator: needleAlta

☑ Show Metric Label

Before You Begin:

It may be helpful to have an understanding of databound data visualization gauges. For more information, see Creating Databound Gauges.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Data Visualization Components.

You will need to complete these tasks:

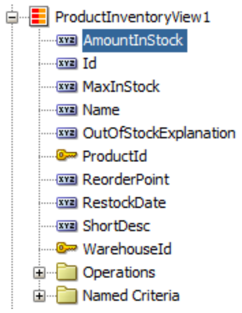
- Create an application module that contains instances of the view objects that you want in your data model, as described in Creating and Modifying an Application Module.
- Create a JSF page as described in the How to Create JSF Pages section of *Developing Web User Interfaces with Oracle ADF Faces*.

To create a databound dial gauge:

1. From the Data Controls panel, select an attribute from a collection.

   For example, to create a dial gauge in the Summit ADF DVT sample application to display the stock levels for an inventory item, you would select the `AmountInStock` attribute in the `ProductInventoryView1` collection. Figure 38-30 shows the `ProductInventoryView1` collection in the Data Controls panel with the `AmountInStock` attribute selected.

Figure 38-30 Data Collection with Amount in Stock for an Inventory Item



2. Drag the attribute onto a JSF page and, from the context menu, choose **Gauges**.
3. In the Component Gallery, choose the category and type of gauge, and then click **OK**.
4. In the Create Dial Gauge dialog, do the following:
   - In the **Metric** field, confirm the column in your data collection that contains the actual value that the gauge is to plot. This is the only required value in the dialog.
   - In the **Minimum** field, if your data collection stores a minimum value for the gauge range, select the column that contains this value from the dropdown list. Alternatively, specify a minimum number for the range. If you do not specify a value, the minimum defaults to `0`.
   - In the **Maximum** field, if your data collection stores a maximum value for the gauge range, select the column that contains this value from the dropdown list. Alternatively, specify a maximum value. If you do not specify a value, the maximum value defaults to `100`.
   - In the **Background** field, select a background from the dropdown list.

     Available shapes include circle, dome, and rectangle, with alta, light, antique, or dark shading. By default, the gauge's background is set to `auto` which will return the primary design for the current skin.
   - In the **Indicator** field, select an indicator style from the dropdown list.

     Available indicators include alta, antique, dark, and light. By default the indicator is set to auto which will select the indicator that matches the gauge's background.
   - In the **Show Metric Label** check box, select the **Show Metric Label** check box to show a metric label for the gauge. The metric label contains the actual metric value for the gauge. For example, if the amount in stock for a given item is 63, then the metric label will display 63.
5. Click **OK**.

In the Properties window, after you complete the binding of the gauge, you can set values for additional attributes in the gauge tag and its child tags to customize the component. For example, to configure the Amount In Stock gauge to show inventory amounts without scaling, set the `scaling` attribute of the metric and tick labels to `none`.

You can also examine and adjust the existing gauge bindings by clicking the **Edit** icon in the Properties window for the gauge component.

For additional information about customizing a gauge after the data binding is completed, see Using Gauge Components in *Developing Web User Interfaces with Oracle ADF Faces*.

What Happens When You Create a Dial Gauge from a Data Control

Dropping a gauge from the Data Controls panel has the following effect:

- Creates the bindings for the gauge and adds the bindings to the page definition file
- Adds the necessary code for the UI components to the JSF page

The example below shows the bindings that JDeveloper generated for the dial gauge that displays the inventory level for a product in a warehouse. This code example shows that the gauge metric receives its value dynamically from the `AmountInStock` attribute in the `ProductInventoryView1` data collection.

```
<bindings>
  <attributeValues IterBinding="ProductInventoryView1Iterator"
id="AmountInStock">
    <AttrNames>
      <Item Value="AmountInStock"/>
    </AttrNames>
  </attributeValues>
</bindings>
```

The example below shows the code that JDeveloper generated in the JSF page for a dial gauge.

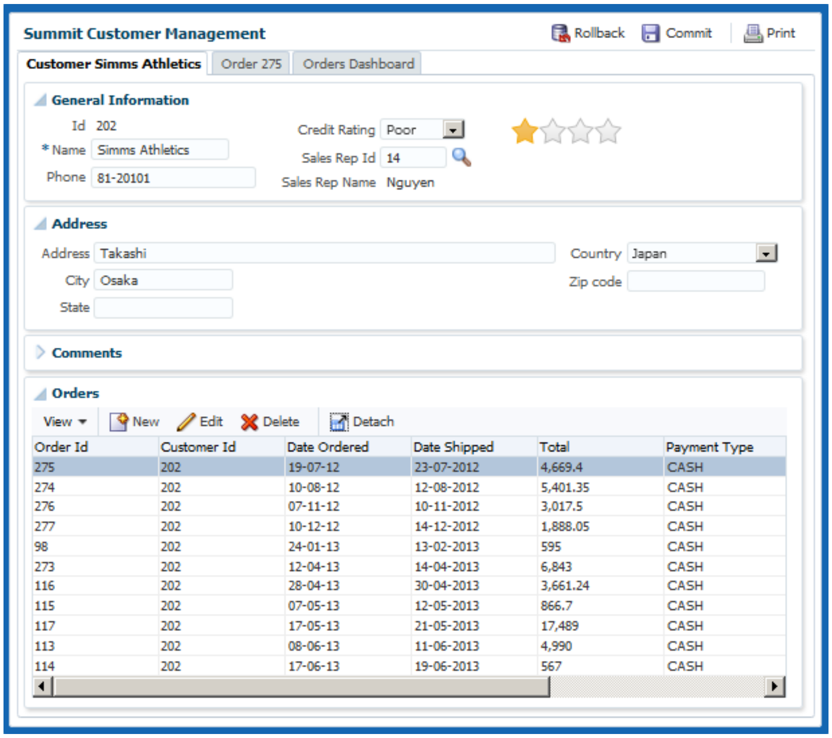
```
<dvt:dialGauge id="dialGauge1" indicator="needleAlta" minimum="0"
               maximum="2600" background="domeAlta"
               value="#{bindings.AmountInStock.inputValue}">
  <dvt:gaugeMetricLabel rendered="true" id="gml1"/>
</dvt:dialGauge>
```

How to Create a Databound Rating Gauge

The rating gauge displays and optionally accepts input for a metric value. This gauge is typically used to show ratings for products or services, such as the star rating for a movie.

Figure 38-31 shows an example of a rating gauge used to indicate a customer's credit rating in the Summit ADF sample application. In this example, the customer has a poor credit rating and is assigned one star. Other ratings include two stars for fair credit, three stars for good credit, and four stars for excellent.

Figure 38-31   Rating Gauge in Summit ADF Sample Application



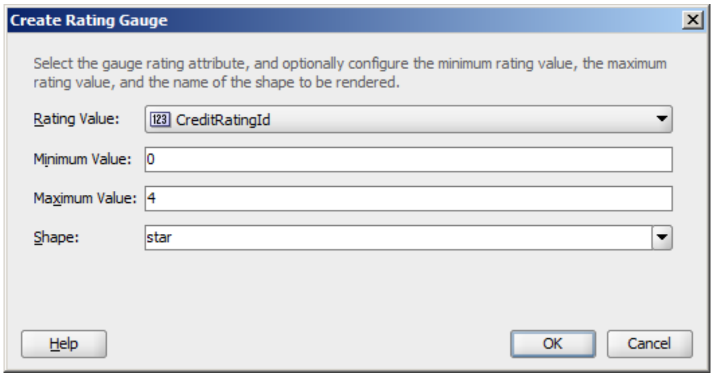
Id	202
* Name	Simms Athletics
Phone	81-20101
Credit Rating	Poor
Sales Rep Id	14
Sales Rep Name	Nguyen
Address	Takashi
City	Osaka
State	
Country	Japan
Zip code	

Order Id	Customer Id	Date Ordered	Date Shipped	Total	Payment Type
275	202	19-07-12	23-07-2012	4,669.4	CASH
274	202	10-08-12	12-08-2012	5,401.35	CASH
276	202	07-11-12	10-11-2012	3,017.5	CASH
277	202	10-12-12	14-12-2012	1,888.05	CASH
98	202	24-01-13	13-02-2013	595	CASH
273	202	12-04-13	14-04-2013	6,843	CASH
116	202	28-04-13	30-04-2013	3,661.24	CASH
115	202	07-05-13	12-05-2013	866.7	CASH
117	202	17-05-13	21-05-2013	17,489	CASH
113	202	08-06-13	11-06-2013	4,990	CASH
114	202	17-06-13	19-06-2013	567	CASH

To create a rating gauge using a data control, you bind the rating gauge to an attribute in a data collection. JDeveloper allows you to do this declaratively by dragging and dropping an attribute from the Data Controls panel. After you drag and drop the attribute, use the Create Rating Gauge dialog to configure the gauge.

Figure 38-32 shows the Create Rating Gauge dialog completed for the rating gauge in Figure 38-31. In this example, the rating gauge is set to a maximum value of four to reflect the four credit ratings.

Figure 38-32   Create Rating Gauge Dialog for Summit Credit Rating Gauge



Select the gauge rating attribute, and optionally configure the minimum rating value, the maximum rating value, and the name of the shape to be rendered.

Rating Value:	CreditRatingId
Minimum Value:	0
Maximum Value:	4
Shape:	star

Before You Begin:

It may be helpful to have an understanding of databound data visualization gauges. For more information, see Creating Databound Gauges.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Data Visualization Components.

You will need to complete these tasks:

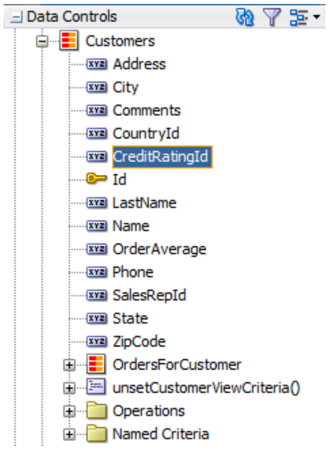
- Create an application module that contains instances of the view objects that you want in your data model, as described in Creating and Modifying an Application Module.
- Create a JSF page as described in the How to Create JSF Pages section of *Developing Web User Interfaces with Oracle ADF Faces*.

To create a databound rating gauge:

1. From the Data Controls panel, select an attribute from a collection.

   For example, to create the rating gauge in the Summit ADF sample application to display the credit ratings for a customer, you would select the `CreditRatingId` attribute in the `Customer` collection. Figure 38-30 shows the `Customer` collection in the Data Controls panel with the `CreditRatingId` attribute selected.

   Figure 38-33 Data Collection with Amount in Stock for an Inventory Item



   Data Controls
   Customers
   Address
   City
   Comments
   CountryId
   CreditRatingId
   Id
   LastName
   Name
   OrderAverage
   Phone
   SalesRepId
   State
   ZipCode
   OrdersForCustomer
   unsetCustomerViewCriteria()
   Operations
   Named Criteria

2. Drag the attribute onto a JSF page and, from the context menu, choose **Gauges**.
3. In the Component Gallery, choose the category and type of gauge, and then click **OK**.
4. In the Create Rating Gauge dialog, do the following:
   - In the **Rating Value** field, confirm the column in your data collection that contains the actual value that the gauge is to plot. This is the only required value in the dialog.
   - In the **Minimum** field, if your data collection stores a minimum value for the gauge range, select the column that contains this value from the dropdown list. Alternatively, specify a minimum number for the range. If you do not specify a value, the minimum defaults to `0`.
   - In the **Maximum** field, if your data collection stores a maximum value for the gauge range, select the column that contains this value from the dropdown

list. Alternatively, specify a maximum value. If you do not specify a value, the maximum value defaults to 5.

- In the **Shape** field, select a background from the dropdown list.

  Available shapes include circle, diamond, rectangle, and star.

5. Click **OK**.

In the Properties window, after you complete the binding of the gauge, you can set values for additional attributes in the gauge tag and its child tags to customize the component. For example, to configure the rating gauge to allow the customer's credit rating to be updated, set the `readOnly` attribute of the rating gauge to `false`.

In the Summit ADF example, the rating gauge's value was set to the `CreditRatingId` during creation. However, the `CreditRatingId` actually ranges between 1 and 4, with 1 being the highest rating. To convert the rating to stars for the rating gauge, edit the source code in the Code Editor after the gauge is credited and replace the rating gauge's value with: `#{5 - bindings.CreditRatingId.inputValue}`.

The example below shows the code on the JSF page for the Summit ADF rating gauge.

```
<dvt:ratingGauge id="ratingGauge1" minimum="0" maximum="4"
                 value="#{5 - bindings.CreditRatingId.inputValue}"
shape="star"/>
```

You can also examine and adjust the existing gauge bindings by clicking the **Edit** icon in the Properties window for the gauge component.

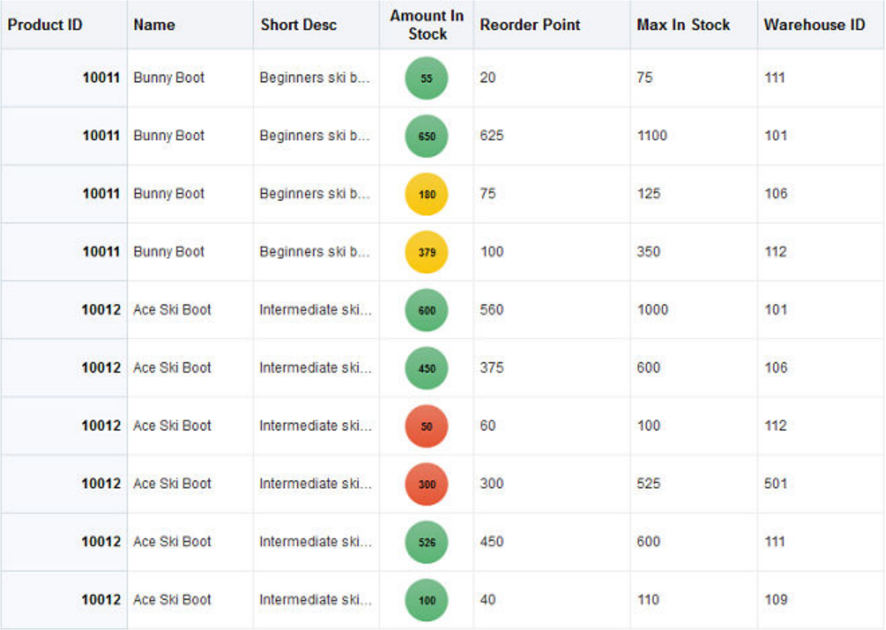
For additional information about customizing a gauge after the data binding is completed, see Using Gauge Components in *Developing Web User Interfaces with Oracle ADF Faces*.

Including Gauges in Databound ADF Tables

You can add databound gauges to a databound ADF table by choosing the `gauge` component when specifying column content during table editing or creation.

Figure 38-34 shows a portion of a table that displays all the products in a warehouse in the Summit ADF DVT sample application. In this example, the `AmountInStock` column includes a LED gauge that shows the amount in stock for each product and whether the value falls within an acceptable range.

Figure 38-34 LED Gauge in an ADF Table



Product ID	Name	Short Desc	Amount In Stock	Reorder Point	Max In Stock	Warehouse ID
10011	Bunny Boot	Beginners ski b...	55	20	75	111
10011	Bunny Boot	Beginners ski b...	650	625	1100	101
10011	Bunny Boot	Beginners ski b...	180	75	125	106
10011	Bunny Boot	Beginners ski b...	379	100	350	112
10012	Ace Ski Boot	Intermediate ski...	600	560	1000	101
10012	Ace Ski Boot	Intermediate ski...	450	375	600	106
10012	Ace Ski Boot	Intermediate ski...	50	60	100	112
10012	Ace Ski Boot	Intermediate ski...	300	300	525	501
10012	Ace Ski Boot	Intermediate ski...	526	450	600	111
10012	Ace Ski Boot	Intermediate ski...	100	40	110	109

How to Include a Gauge in a Databound ADF Table

To add a gauge to a databound ADF table, drag a data collection that includes the gauge metric from the Data Controls panel to the JSF page and specify the type of gauge to create.

Before you begin:

It may be helpful to have an understanding of databound data visualization gauges. For more information, see Creating Databound Gauges.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Data Visualization Components.

You will need to complete these tasks:

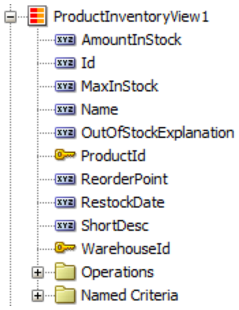
- Create an application module that contains instances of the view objects that you want in your data model, as described in Creating and Modifying an Application Module.
- Create a JSF page as described in the How to Create JSF Pages section of *Developing Web User Interfaces with Oracle ADF Faces*.

To include a gauge in a databound ADF table:

1. From the Data Controls panel, select the collection to use for the ADF table.

   For example, to create the table that displays the inventory levels for each product in a warehouse in the Summit ADF DVT sample application, select the `ProductInventoryView1` collection. Figure 38-35 shows the `ProductInventoryView1` collection in the Data Controls panel.

Figure 38-35 ProductInventoryView1 Data Collection



- ProductInventoryView1
  - AmountInStock
  - Id
  - MaxInStock
  - Name
  - OutOfStockExplanation
  - ProductId
  - ReorderPoint
  - RestockDate
  - ShortDesc
  - WarehouseId
  - Operations
  - Named Criteria

2. Drag the collection onto a JSF page and, from the context menu, choose **Table > ADF Table**.

3. In the Edit Table Columns dialog, select the column that represents the gauge metric.

   For example, select the `AmountInStock` column to use as the gauge metric.

4. From the Component to Use dropdown menu, choose the type of gauge to create.

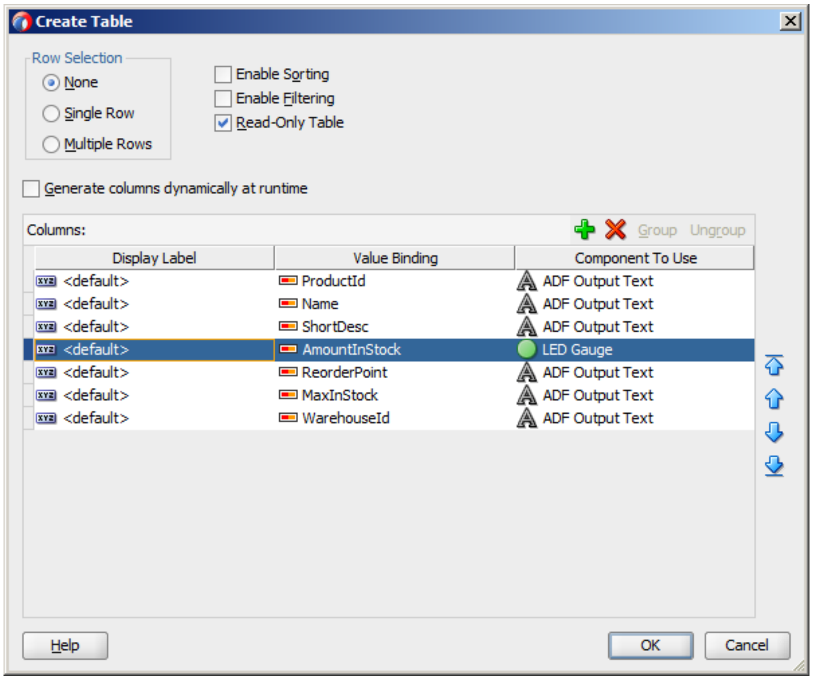
   For example, choose **Gauge Status Meter** to add a status meter to the table.

5. Complete the table configuration.

   If you need help, press F1 or click **Help**.

   Figure 38-36 shows the completed dialog for the table displayed in Figure 38-34, with the **LED Gauge** selected for the `AmountInStock` column.

Figure 38-36 Create Table Dialog Showing LED Gauge



Create Table

Row Selection
● None
○ Single Row
○ Multiple Rows

□ Enable Sorting
□ Enable Filtering
☑ Read-Only Table

□ Generate columns dynamically at runtime

Columns:

Display Label	Value Binding	Component To Use
<default>	ProductId	ADF Output Text
<default>	Name	ADF Output Text
<default>	ShortDesc	ADF Output Text
<default>	AmountInStock	LED Gauge
<default>	ReorderPoint	ADF Output Text
<default>	MaxInStock	ADF Output Text
<default>	WarehouseId	ADF Output Text

6. Click **OK** to add the table to the JSF page.

In the Properties window, after you complete the binding of the table, you can set values for additional attributes in the gauge tag and its child tags to customize the component. For example, the LED gauge in Figure 38-34 is configured to use thresholds to indicate whether the stock level is within an acceptable range. For additional information about configuring gauge thresholds, see How to Configure Gauge Thresholds in *Developing Web User Interfaces with Oracle ADF Faces*.

You can also examine and adjust the existing table bindings by clicking the **Edit** icon in the Properties window for the table component.

For additional information about creating and configuring databound ADF tables, see Creating ADF Databound Tables. For additional information about using and customizing gauge components, see Using Gauge Components in *Developing Web User Interfaces with Oracle ADF Faces*.

What Happens When You Include a Gauge in an ADF Table

When you include a gauge in a databound ADF table, the gauge's metric attribute is added to the page definition file, and the UI components are updated on the JSF page.

The example below shows the binding for the ADF table shown in Figure 38-34.

```
<bindings>
  <tree IterBinding="ProductInventoryView1Iterator"
id="ProductInventoryView1">
    <nodeDefinition DefName="model.ProductInventoryView"
                    Name="ProductInventoryView10">
      <AttrNames>
        <Item Value="ProductId"/>
        <Item Value="Name"/>
        <Item Value="ShortDesc"/>
        <Item Value="AmountInStock"/>
        <Item Value="ReorderPoint"/>
        <Item Value="MaxInStock"/>
        <Item Value="WarehouseId"/>
      </AttrNames>
    </nodeDefinition>
  </tree>
</bindings>
```

The example below shows the code added to the JSF page for the ADF table. The gauge elements are highlighted in bold. For the sake of brevity, only the first three table columns are displayed.

```
<af:table value="#{bindings.ProductInventoryView1.collectionModel}"
var="row"
          rows="#{bindings.ProductInventoryView1.rangeSize}"
          emptyText="#{bindings.ProductInventoryView1.viewable ?
                    'No data to display.' : 'Access Denied.'}"
          rowBandingInterval="0"
          fetchSize="#{bindings.ProductInventoryView1.rangeSize}"
id="t1"
          summary="Table Showing Product Inventory">
  <af:column
headerText="#{bindings.ProductInventoryView1.hints.ProductId.label}"
             id="c1" width="100"
             rowHeader="true">
```

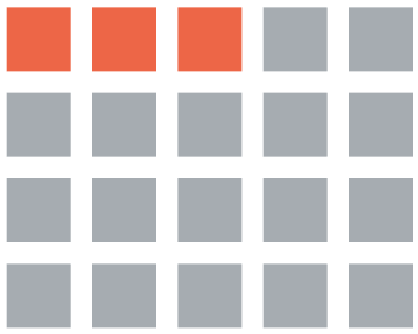
```
    <af:outputText value="#{row.ProductId}"

shortDesc="#{bindings.ProductInventoryView1.hints.ProductId.tooltip}"
             id="ot1">
      <af:convertNumber groupingUsed="false"

pattern="#{bindings.ProductInventoryView1.hints.ProductId.format}"/>
    </af:outputText>
  </af:column>
  <af:column
headerText="#{bindings.ProductInventoryView1.hints.Name.label}"
             id="c2" width="100">
    <af:outputText value="#{row.Name}"

shortDesc="#{bindings.ProductInventoryView1.hints.Name.tooltip}"
id="ot2"/>
  </af:column>
  <af:column
headerText="#{bindings.ProductInventoryView1.hints.ShortDesc.label}"
             id="c3" width="100">
    <af:outputText value="#{row.ShortDesc}"

shortDesc="#{bindings.ProductInventoryView1.hints.ShortDesc.tooltip}"
id="ot3"/>
  </af:column>
  <af:column
headerText="#{bindings.ProductInventoryView1.hints.AmountInStock.label}"
             id="c4" width="72" align="center">
    <dvt:ledGauge id="ledGauge1"
value="#{row.bindings.AmountInStock.inputValue}"

shortDesc="#{bindings.ProductInventoryView1.hints.AmountInStock.tooltip}"
                  maximum="3000">
      <dvt:gaugeMetricLabel rendered="true" scaling="none" id="gml1"/>
      <dvt:gaugeThreshold id="gt1"
maximum="#{row.bindings.ReorderPoint.inputValue}"
                          color="#d62800"/>
      <dvt:gaugeThreshold id="gt2"
maximum="#{row.bindings.MaxInStock.inputValue}"
                          color="#63a500"/>
      <dvt:gaugeThreshold id="gt3" color="#e7e700"/>
    </dvt:ledGauge>
  </af:column>

... remaining columns omitted
</af:table>
```

Creating Databound Picto Charts

The ADF DVT Picto Chart component uses icons to visualize an absolute number, or the relative sizes of the different parts of a population. Picto Charts are extensively used in infographics as a more interesting and effective way to present numerical information than traditional tables and lists.

Figure 38-37 shows a basic picto chart with two parts of a population. The data contains two rows of information, with each row having a name, count and color information.

Figure 38-37   Sample Picto Chart



For detailed information about picto chart end user and presentation features, use cases, tag structure, and adding special features to picto charts, see the Using Picto Chart Components in *Developing Web User Interfaces with Oracle ADF Faces*.

How to Create a Databound Picto Chart

A picto chart is used to visualize an absolute number or the relative sizes of the different parts of a population. You can create a picto chart by using the Data Controls panel.

Before you Begin:

It may be helpful to have an understanding of databound data visualization picto charts. See Creating Databound Picto Charts.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. See Additional Functionality for Data Visualization Components.

You will need to complete these tasks:

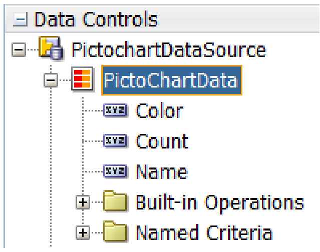
- Create an **application module** that contains instances of the **view objects** that you want in your data model, as described in Creating and Modifying an Application Module.
- Create a JSF page, as described in How to Create JSF Pages in *Developing Web User Interfaces with Oracle ADF Faces*.

To create a databound picto chart:

1. From the Data Controls panel, select a collection.

   For example, to use the data collection shown in Figure 38-37, select the `PictoChartData` collection. Figure 38-38 shows the collection in the Data Controls panel.

Figure 38-38 Data Control for Sample Picto Chart



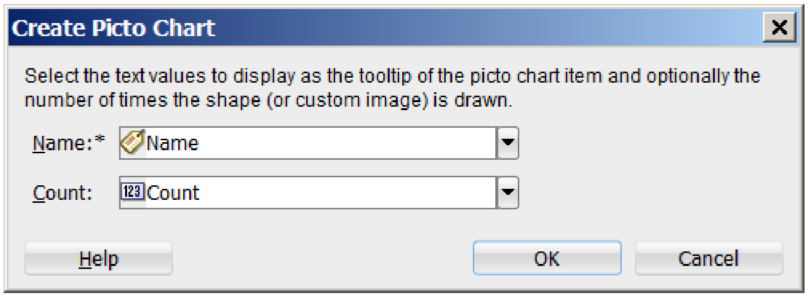
2. Drag the collection onto a JSF page, and from the menu, choose **Picto Chart**.
3. In the Create Picto Chart dialog, do the following:
   - In the **Name** field, specify a value for the tooltip to be displayed.

     For example, to configure the picto chart in the sample, select the `Name` attribute.
   - Optionally, in the **Count** field, specify a numeric value for the number of times the shape or image should be drawn. You can specify a data item or text resource or use the **Expression Builder** from the dropdown list. If no value is specified, it will use a default value of `1`.

     For example, to configure the picto chart in the sample, select the `Count` attribute.

     Figure 38-39 shows the Create Picto Chart dialog with the values filled in.

Figure 38-39 Create Picto Chart Dialog


Create Picto Chart

Select the text values to display as the tooltip of the picto chart item and optionally the number of times the shape (or custom image) is drawn.

Name:* Name

Count: Count

4. Click **OK**.

After you complete the binding of the picto chart, you can use the Properties window to set values for additional attributes in the picto chart tag and its child tags to customize the component. For more information about configuring picto chart components, see Using Picto Chart Components in *Developing Web User Interfaces with Oracle ADF Faces*

What Happens When You Create a Picto Chart from a Data Control

To create a picto chart from a Data control, drag the control from the Data Control panel and drop it on a relevant container on the JSF page.

Dropping a picto chart from the Data Controls panel has the following effect:

- Creates the bindings for the picto chart and adds the bindings to the page definition file

- Adds the necessary code for the UI components to the JSF page

The example below shows the bindings that JDeveloper generated for the sample picto chart.

```
<tree IterBinding="PictoChartDataIterator" id="PictoChartData"
ChangeEventPolicy="ppr">
    <nodeDefinition DefName="view.prototype.PictoChartData"
Name="PictoChartData">
        <AttrNames>
            <Item Value="Name"/>
            <Item Value="Count"/>
        </AttrNames>
    </nodeDefinition>
</tree>
```

The example below shows the code that JDeveloper generated in the JSF page for a picto chart.

```
<dvt:pictoChart id="pictoChart1" var="row"
value="#{bindings.PictoChartData.collectionModel}">
  <dvt:pictoChartItem name="#{row.Name}" count="#{row.Count}"
id="pci1"/>
</dvt:pictoChart>
```

After you create your picto chart, you can modify the binding or add additional rules using the Edit Picto Chart dialog. To open the dialog, click the **Edit** icon in the Properties window for the picto chart component. You can also customize the attributes of the picto chart directly in the code, in the visual editor, or by setting values in the Properties window. For example, you can use the Properties window or code editor to set the color for different parts of the population and to set a shape for the data items.. The example below highlights the relevant code.

```
<dvt:pictoChartItem name="#{row.Name}" count="#{row.Count}"
color="#{row.Color}"
            shape="human" id="pci1"/>
```

39 Creating Databound NBox Components

This chapter describes how to create NBox components from data modeled with ADF Business Components, using ADF data controls and ADF Faces components in a Fusion web application. Specifically, it describes how you can use ADF Data Visualization `nBox` components that visually represent business data. It describes how to use ADF data controls to create a NBox with data-first development.
If you are designing your page using simple UI-first development, then you can add the NBox to your page and configure the data bindings later. For information about the data requirements, tag structure, and options for customizing the look and behavior of the `nBox` components, see Using NBox Components in *Developing Web User Interfaces with Oracle ADF Faces*.

This chapter includes the following sections:

About ADF Data Visualization NBox Components

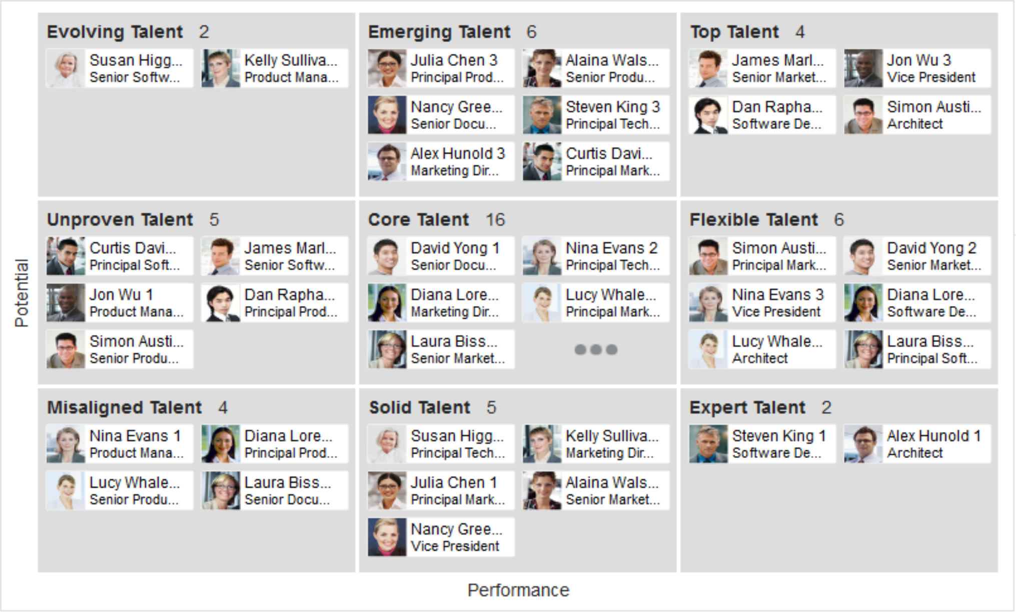
The NBox Component is a data grouping visualization that utilizes two ranges of data to form a grid of cells, and each cell contains customizable nodes that represent individual data items. NBox components are useful to group data with multiple factors of consideration, such as employee potential against employee performance.

ADF Data Visualization components provide extensive graphical and tabular capabilities for visually displaying and analyzing business data. You must bind each component to data before it can be rendered since the appearance of the components is dictated by the data that is displayed.

The ADF `nBox` component produces an interactive component that you can use to visualize and compare data across a two-dimensional grid, represented visually by rows and columns. The `nBox` component is comprised of two parts: the node that represents the data and the grid that comprises the cells into which the nodes are placed. If the number of nodes is greater than the space allocated for the cell, the NBox displays an indicator that users can click to access the additional nodes.

For example, you can use the `nBox` component to compare employee potential and performance data, where the row represents employee potential and the column represents employee performance. The node that represents the employee is stamped into the appropriate cell.

Figure 39-1 NBox Component Comparing Employee Potential and Performance



Evolving Talent 2 Susan Higg... Senior Softw... Kelly Sulliva... Product Mana...	Emerging Talent 6 Julia Chen 3 Principal Prod... Alaina Wals... Senior Produ... Nancy Gree... Senior Docu... Steven King 3 Principal Tech... Alex Hunold 3 Marketing Dir... Curtis Davi... Principal Mark...	Top Talent 4 James Marl... Senior Market... Jon Wu 3 Vice President Dan Rapha... Software De... Simon Austi... Architect
Unproven Talent 5 Curtis Davi... Principal Soft... James Marl... Senior Softw... Jon Wu 1 Product Mana... Dan Rapha... Principal Prod... Simon Austi... Senior Produ...	Core Talent 16 David Yong 1 Senior Docu... Nina Evans 2 Principal Tech... Diana Lore... Marketing Dir... Lucy Whale... Principal Mark... Laura Biss... Senior Market... ●●●	Flexible Talent 6 Simon Austi... Principal Mark... David Yong 2 Senior Market... Nina Evans 3 Vice President Diana Lore... Software De... Lucy Whale... Architect Laura Biss... Principal Soft...
Misaligned Talent 4 Nina Evans 1 Product Mana... Diana Lore... Principal Prod... Lucy Whale... Senior Produ... Laura Biss... Senior Docu...	Solid Talent 5 Susan Higg... Principal Tech... Kelly Sulliva... Marketing Dir... Julia Chen 1 Principal Mark... Alaina Wals... Senior Market... Nancy Gree... Vice President	Expert Talent 2 Steven King 1 Software De... Alex Hunold 1 Architect

The prefix `dvt:` occurs at the beginning of each data visualization component name indicating that the component belongs to the ADF Data Visualization Tools (DVT) tag library.

End User and Presentation Features

Visually compelling data visualization components enable end users to understand and analyze complex business data. The components are rich in features that provide out-of-the-box interactivity support. For detailed descriptions of the end user and presentation features for the NBox component, see the End User and Presentation Features section in *Developing Web User Interfaces with Oracle ADF Faces*.

Data Visualization Components Use Cases and Examples

For detailed descriptions of each data visualization use case and example, see the NBox Use Cases and Examples section in *Developing Web User Interfaces with Oracle ADF Faces*.

Additional Functionality for Data Visualization Components

You may find it helpful to understand other Oracle ADF features before you data bind your data visualization components. Additionally, once you have added a data visualization component to your page, you may find that you need to add functionality such as validation and accessibility. Following are links to other functionality that data visualization components use:

- Partial page rendering: You may want a data visualization component to refresh to show new data based on an action taken on another component on the page. For more information, see Rerendering Partial Page Content in *Developing Web User Interfaces with Oracle ADF Faces*.

- Personalization: Users can change the way the data visualization components display at runtime, those values will not be retained once the user leaves the page unless you configure your application to allow user customization. For more information, see Allowing User Customization on JSF Pages in *Developing Web User Interfaces with Oracle ADF Faces*.
- Accessibility: You can make your data visualization components accessible. For more information, see Developing Accessible ADF Faces Pages in *Developing Web User Interfaces with Oracle ADF Faces*.
- Skins and styles: You can customize the appearance of data visualization components using an ADF skin that you apply to the application or by applying CSS style properties directly using a style-related property (`styleClass` or `inlineStyle`). For more information, see Customizing the Appearance Using Styles and Skins in *Developing Web User Interfaces with Oracle ADF Faces*.
- Placeholder data controls: If you know the data visualization components on your page will eventually use ADF data binding, but you need to develop the pages before the data controls are ready, then you should consider using placeholder data controls, rather than manually binding the components. Using placeholder data controls will provide the same declarative development experience as using developed data controls. For more information, see Designing a Page Using Placeholder Data Controls.

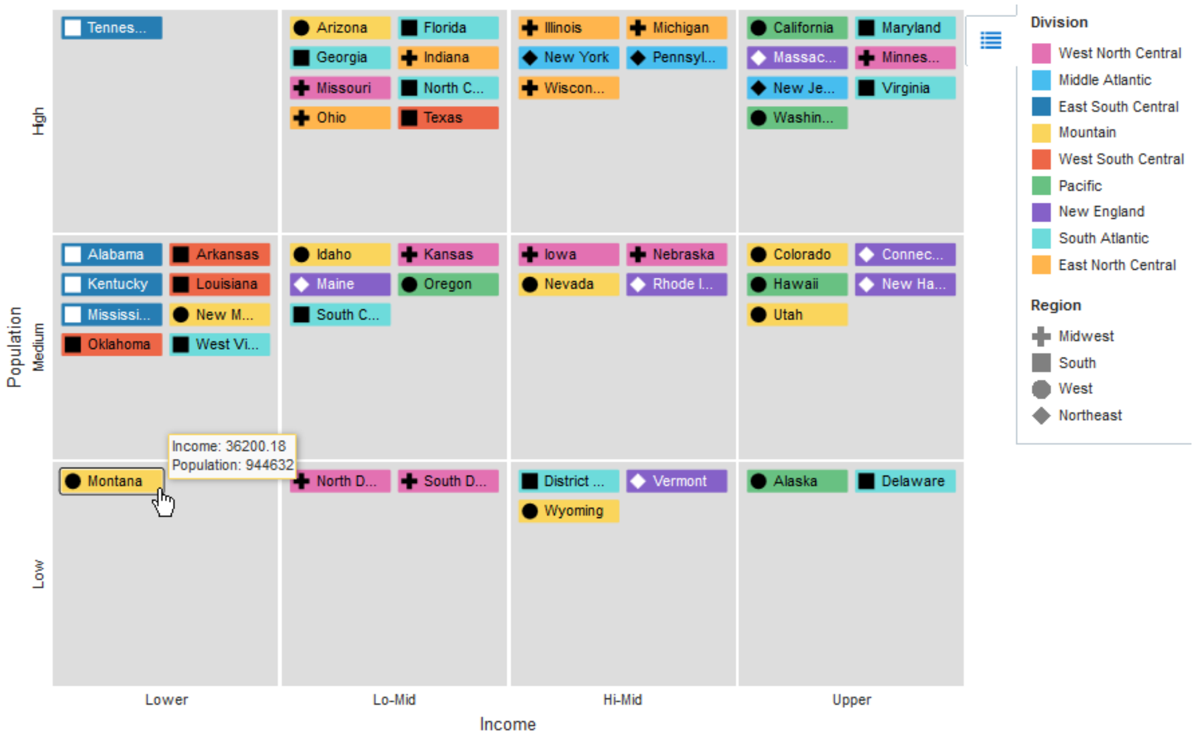
Creating Databound NBox Components

DVT NBox components can be created and bound to a data collection with a data-first development approach. JDeveloper allows you to do this declaratively by dragging and dropping a collection from the Data Controls panel.

The ADF NBox component displays data in a grid layout, with a configurable number of rows and columns used to represent two dimensions or measures of data. Nodes represent the actual data and are stamped inside the grid's cells according to where the node's value falls within the ranges or measures specified for the cells.

Figure 39-2 shows a sample NBox configured to display a grid with three rows and four columns. Each row represents a range of population data, labeled as Low, Medium, or High. Each column represents a range of income data, labeled as Lower, Lo-Mid, Hi-Mid, or Upper. The nodes represent the states, which are stamped into a cell depending on the value of the state's population and income level. The sample NBox also uses optional attribute groups to color each node depending upon its Division value and to vary the node's icon marker shape depending upon its Region value.

Figure 39-2 NBox Showing United States State Income and Population



The Create NBox wizard provides declarative support for creating the NBox and binding it to data. In the wizard pages you can:

- Specify the initial layout of the NBox, including:
  - Number of rows and columns (Required)
  - Values for each NBox row and column (Required)
  - Titles for the NBox rows and columns
  - Labels for each NBox row and column
- Configure the NBox nodes, including:
  - Attributes to use from the data collection to determine the node's row and column placement (Required)
  - Primary and secondary node labels
  - Attribute groups to group nodes by color or shape according to a specified attribute
  - Images to use for the node's icon or indicator

After you complete the NBox wizard and the NBox is added to your page, you can further customize the NBox using the Properties window. For additional information, see Using NBox Components in *Developing Web User Interfaces with Oracle ADF Faces*.

How to Create an NBox Component Using ADF Data Controls

To create a DVT NBox using a data control, bind the `dvt:nBox` component to a collection. JDeveloper allows you to do this declaratively by dragging and dropping a collection from the Data Controls panel.

> **Tip:**
>
> You can also create the NBox by dragging the NBox from the Components window. This approach allows you the option of designing the NBox user interface before binding the component to data.

Before you begin:

It may be helpful to have an understanding of databound NBoxes. For more information, see About ADF Data Visualization NBox Components.

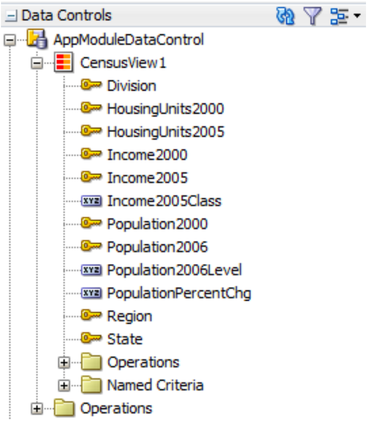
You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Data Visualization Components.

You will need to complete these tasks:

- Create an application module that contains instances of the view objects that you want in your data model for the NBox, as described in Creating and Modifying an Application Module.

  Figure 39-3 shows the data control for the NBox displayed in Figure 39-2. In this example, `CensusView1` contains the data collection for the NBox. NBox nodes are represented visually as rectangular boxes identified by their state names. Each node is stamped into one of the NBox cells according to the values in the node's `Population2006Level` and `Income2005Class` attributes.

Figure 39-3 Data Control for NBox Showing United States Income and Population



The `Population2006Level` and `Income2005Class` attributes are defined as transient attributes whose values are derived from the `Population2006` and `Income2005` attributes in the data collection. In this example, the rows are defined as ranges, with the lowest range displayed on the bottom row of the NBox. For example, a node whose `Population2006` value is less than or equal to 1,000,000 will be assigned a `Population2006Level` of `low`. Columns are also defined as ranges, with the lowest range displayed in the leftmost column for left-to-right locales and in the rightmost column for right-to-left locales.

```
package model

import oracle.jbo.script.annotation.TransientValueExpression;

@TransientValueExpression(attributeName="Population2006Level")
def Population2006Level_ExpressionScript_Expression()
{
Population2006 <= 1000000 ? 'low' : (Population2006 <= 5000000 ?
'medium' : 'high')
}

@TransientValueExpression(attributeName="Income2005Class")
def Income2005Class_ExpressionScript_Expression()
{
Income2005 <= 40000.0 ? 'lower' : (Income2005 <= 45000.0 ? 'lo-mid' :
(Income2005 <= 50000.0 ? 'hi-mid' : 'upper'))
}
```

For information about adding transient attributes to view objects, see Adding Calculated and Transient Attributes to a View Object.

- Create a JSF page as described in the How to Create JSF Pages section of *Developing Web User Interfaces with Oracle ADF Faces*.

To create a databound NBox:

1. From the Data Controls panel, select a collection.

   For example, to use the data control shown in this section, select `CensusView1`.

2. Drag the collection onto a JSF page and, from the menu, choose **NBox**.

3. In the Create NBox - Configure NBox Grid dialog, enter numeric values for the number of rows and number of columns in the NBox.

   For example, enter `3` for Rows and `4` for Columns to create the NBox with three rows and four columns.

4. Optionally, enter the text label to display for the row and column titles of the NBox. If you wish to associate a text resource from a resource bundle with the text label, select **Select Text Resource** to launch a dialog available for that purpose. If you wish to access data stored in objects, or reference and invoke methods using an EL Expression, select **Expression Builder** to launch the Expression Builder dialog. The rows title displays in the dynamic sample NBox.

5. For each cell in the NBox, enter values for the row and column.

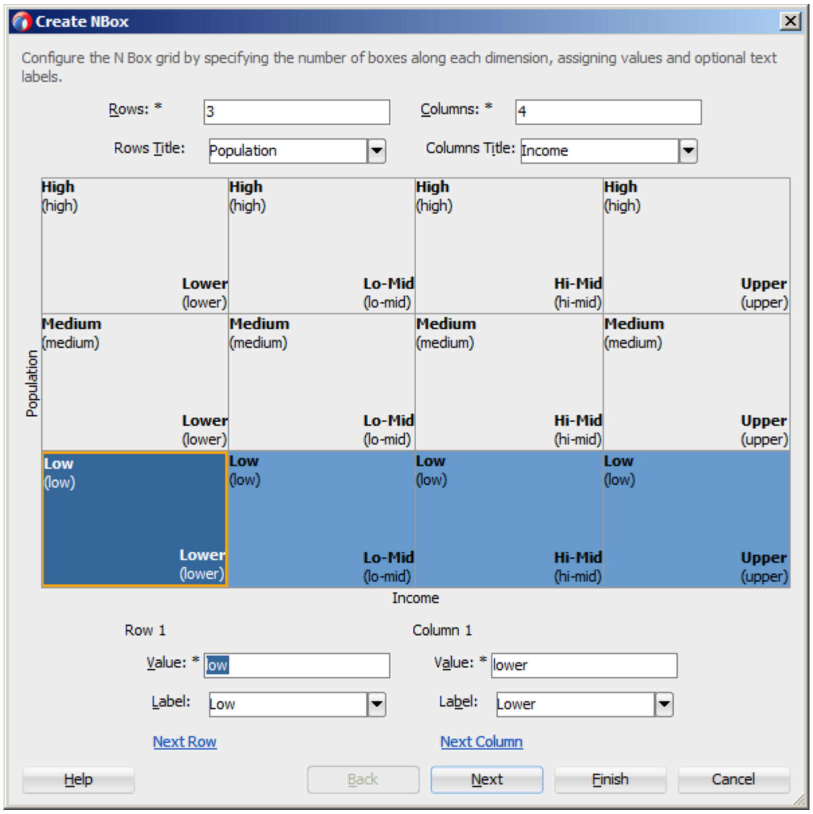
   For example, to use the data control shown in this section, in Row1 enter `low` for the row value and `lower` for Column 1's value.

   Optionally, to display a text label in each cell, select **Select Text Resource** to associate a text resource for the label. If you wish to access data stored in objects, or reference and invoke methods using an EL Expression, select **Expression Builder** to launch the Expression Builder dialog.

You can click **Next Row** or **Next Column** to move to the next row or column in the NBox. Alternatively, you can click in the desired cell in the NBox dynamic sample to display the row and column fields for that cell. When you have finished entering values, click **Next** to move to the next dialog.

Figure 39-4 shows the completed dialog for the NBox shown in this section.

Figure 39-4 Create NBox Dialog for NBox Showing United States Population and Income



Create NBox

Configure the N Box grid by specifying the number of boxes along each dimension, assigning values and optional text labels.

Rows: * 3
Columns: * 4
Rows Title: Population
Columns Title: Income

Population	Income			
	High (high) Lower (lower)	High (high) Lo-Mid (lo-mid)	High (high) Hi-Mid (hi-mid)	High (high) Upper (upper)
	Medium (medium) Lower (lower)	Medium (medium) Lo-Mid (lo-mid)	Medium (medium) Hi-Mid (hi-mid)	Medium (medium) Upper (upper)
	Low (low) Lower (lower)	Low (low) Lo-Mid (lo-mid)	Low (low) Hi-Mid (hi-mid)	Low (low) Upper (upper)

Row 1
Value: * low
Label: Low

Column 1
Value: * lower
Label: Lower

6. In the Create NBox - Configure Node dialog, in the Row and Column fields, select the attributes from the data collection that contains the row and column values. If you wish to access data stored in objects, or reference and invoke methods using an EL Expression, select **Expression Builder** to launch the Expression Builder dialog.

   For example, select **Population2006Level** and **Income2005Class** to use the transient attributes shown in this section.

7. Optionally, to display a text label in the node, from the Label and Secondary Label dropdown menus, select the attribute to use for the text label. To use a text resource for the label, select **Select Text Resource** to associate a text resource for the label. If you wish to access data stored in objects, or reference and invoke methods using an EL Expression, select **Expression Builder** to launch the Expression Builder dialog.

8. Optionally, to configure the NBox node's icon or indicator, in the Create NBox dialog, click **Icon** or **Indicator** and enter values for the following:

- **Shape**: Click and use the dropdown to select a shape to use for the icon or indicator. Valid values include: `circle`, `human`, `diamond`, `square`, `triangleDown`, `triangleUp`, and `plus`.

> **Note:**
>
> When you specify the icon or indicator shape explicitly, the same shape will be used for each node in the NBox. If you want to vary the shape according to some measure, configure an attribute group instead.

- **Image**: Click and use the dropdown to select an image to use for the icon or indicator.

9. Optionally, to configure an attribute group for the NBox node, icon, or indicator, click **New** in the **Grouping Rules** section under the Node, Icon, or Indicator tab. Use attribute groups if you want the NBox node, icon, or indicator display to vary based on color, shape, or pattern. Attribute groups are also required if you want to display a legend.

   To configure an attribute group, enter values for the following:

   - **Group by value**: From the dropdown list, select the attribute in the data collection to group by in the attribute group. You can select one of the attributes provided or select **Expression Builder** to enter a JSF EL expression.

     The sample application in this section defines attribute groups to group the nodes by Division and the icon by Region. For example, select **Division** as the **Group by value** for the `CensusView1` collection, and the colors displayed on the NBox nodes will vary according to color.

   - **Node**, **Icon**, or **Indicator**: From the dropdown list, select the option that you want to group by for display.

     For the NBox node, you can choose **Color** to group the nodes by color or select **Indicator Color** if you want the attribute group to vary by color and display the color as an indicator. To vary the attribute group by both color and indicator color, select **Select Multiple Attributes** and select both **Color** and **Indicator Color**. Click **OK**.

     For the NBox icon and indicator, you can choose **Color**, **Shape**, or **Pattern** to group the icon or indicator by color, shape, or pattern. To vary the attribute group by multiple attributes, select **Select Multiple Attributes** and select **Color**, **Shape**, or **Pattern** from the dialog. Click **OK**.

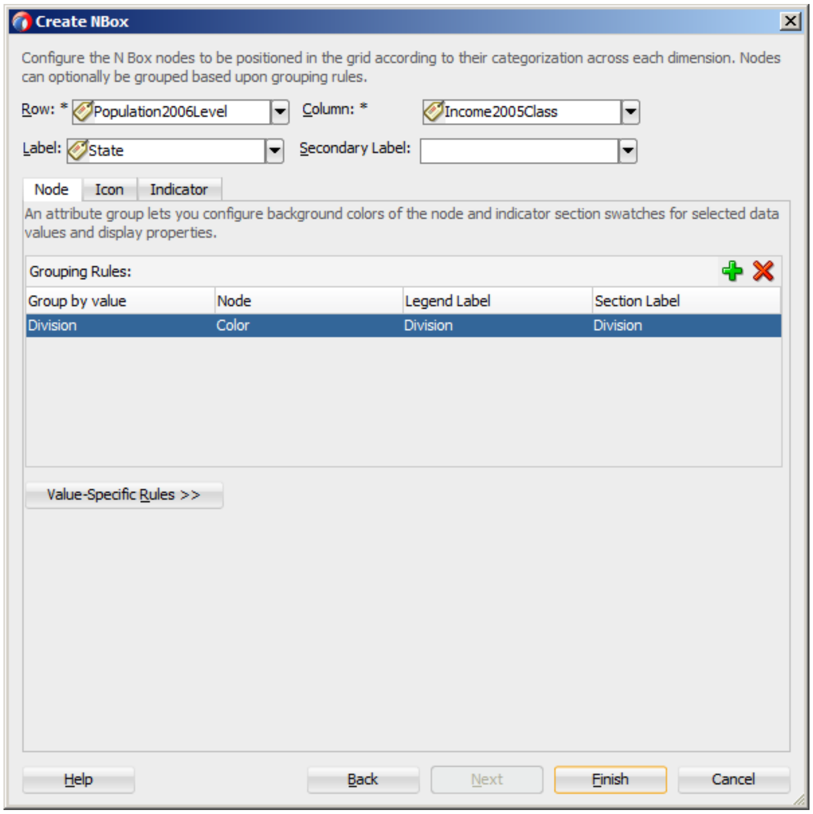
   - **Legend Label**: From the dropdown list, select the attribute in the data collection to display in the NBox legend. You can select one of the attributes provided or select **Expression Builder** to enter a JSF EL expression.

   - **Section Label**: To use a text resource for the label, select **Select Text Resource** to associate a text resource for the label. If you wish to access data stored in objects, or reference and invoke methods using an EL Expression, select **Expression Builder** to launch the Expression Builder dialog.

> **Note:**
>
> The section label describes the legend content of a sub-section of the legend and is rendered in the legend area. In Figure 39-2, for example, Division and Region are section labels.

Figure 39-5 shows the completed Create NBox dialog to configure the nodes.

Figure 39-5 Create NBox Dialog for Node Configuration



Create NBox

Configure the N Box nodes to be positioned in the grid according to their categorization across each dimension. Nodes can optionally be grouped based upon grouping rules.

Row: * Population2006Level Column: * Income2005Class

Label: State Secondary Label:

Node | Icon | Indicator

An attribute group lets you configure background colors of the node and indicator section swatches for selected data values and display properties.

Grouping Rules:

Group by value	Node	Legend Label	Section Label
Division	Color	Division	Division

- Optionally, click **Value-Specific Rules** to expand the attribute group dialog to specify a match or exception rule. Use match rules to specify colors or patterns for simple true or false conditions or when you want to match a specific value. Use exception rules when you want to specify a color or pattern when the grouped-by value meets a specific condition.

  To specify a match rule, in the **Match Rules** section, click **New** and enter values for the following:

  – **Group Value**: Enter the category value for the match. This can be a string that represents a category or you can set this to `true` or `false`. If you set this to true or false, the **Group by value** field must contain an EL expression that evaluates to true or false as in the following example:`#{row.AmountInStock gt row.ReorderPoint}`.

  – **Property**: From the dropdown list, select **Color** if you want the node to vary by color or select **Indicator Color** if you want the background of the indicator area on the node to vary by color.

- **Property Value**: From the dropdown list, select the color or pattern to display when the node's value matches the **Group Value**. For color, you can select one of the provided values or select **Custom Color** to enter a custom color in the Select Custom Color dialog. For pattern, you must select one of the provided values.

To specify an exception rule, in the **Exception Rules** section, click **New** and enter values for the following:

- **Condition**: Enter a JSF EL expression that evaluates to true or false. You can enter the expression directly in the **Condition** field or select **Expression Builder** to enter the JSF EL expression.
- **Property**: From the dropdown list, select **Color** if you want the node to vary by color. Select **Indicator Color** if you want the node's indicator area to vary by indicator color.
- **Property Value**: From the dropdown list, select the color or pattern to display when the node's value meets the condition you specified in the **Condition** field. For color, you can select one of the provided values or select **Custom Color** to enter a custom color in the Select Custom Color dialog. For pattern, you must select one of the provided values.
- **Legend Label**: From the dropdown list, select **Select Text Resource** to select a text resource to be used for the legend label. You can also enter text in this field or select **Expression Builder** to enter a JSF EL expression.

10. When you have completed the node configuration, click **Finish** to exit the dialog and add the NBox to the page.

After completing the Create NBox dialog, you can use the Properties window to specify settings for the NBox attributes, and you can also use the child tags associated with the NBox tag to customize the NBox further. For detailed information about NBox end user and presentation features, use cases, tag structure, and special features, see Using NBox Components in *Developing Web User Interfaces with Oracle ADF Faces*.

What Happens When You Create a Databound NBox

Creating an NBox from the Data Controls panel has the following effect:

- Creates the bindings for the NBox in the page definition file (*pageName*`PageDef.xml`) of the JSF page
- Adds the necessary tags to the JSF page for the `dvt:nBox` component

Bindings for NBox Components

The following code sample shows the bindings that JDeveloper generated for a `dvt:nBox` component using the data collection shown in Figure 39-3:

```
<executables>
  <variableIterator id="variables"/>
  <iterator Binds="CensusView1" RangeSize="-1" DataControl="AppModuleDataControl"
            id="CensusView1Iterator"/>
</executables>
<bindings>
  <tree IterBinding="CensusView1Iterator" id="CensusView1">
    <nodeDefinition DefName="model.CensusView" Name="CensusView10">
      <AttrNames>
```

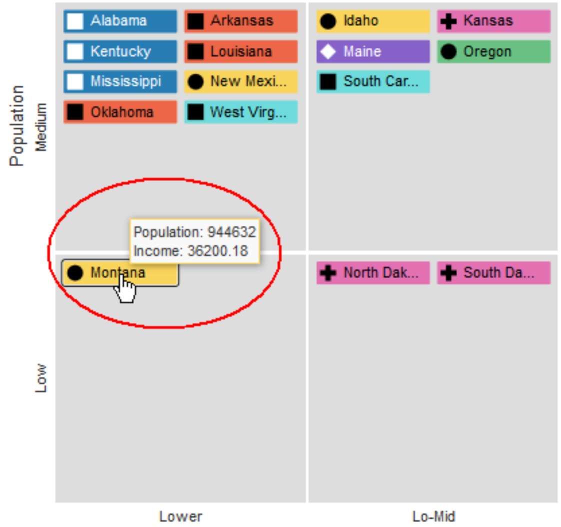
```
        <Item Value="Population2006Level"/>
        <Item Value="Income2005Class"/>
        <Item Value="State"/>
        <Item Value="Division"/>
        <Item Value="Region"/>
      </AttrNames>
    </nodeDefinition>
  </tree>
</bindings>
```

The rules for populating the NBox node are defined in a node definition. Each node definition references a view object and the attributes specified in the Create NBox dialog. For additional information about the *pageName*`PageDef.xml` file, see pageNamePageDef.xml.

Editing the NBox Binding

Attributes that were not specified in the Create NBox dialog during creation will not be included in the binding. If you need to reference another attribute in the data collection, you must add it manually to the binding. For example, the sample in this section displays a tooltip showing the state's income and population when the user hovers over a node.

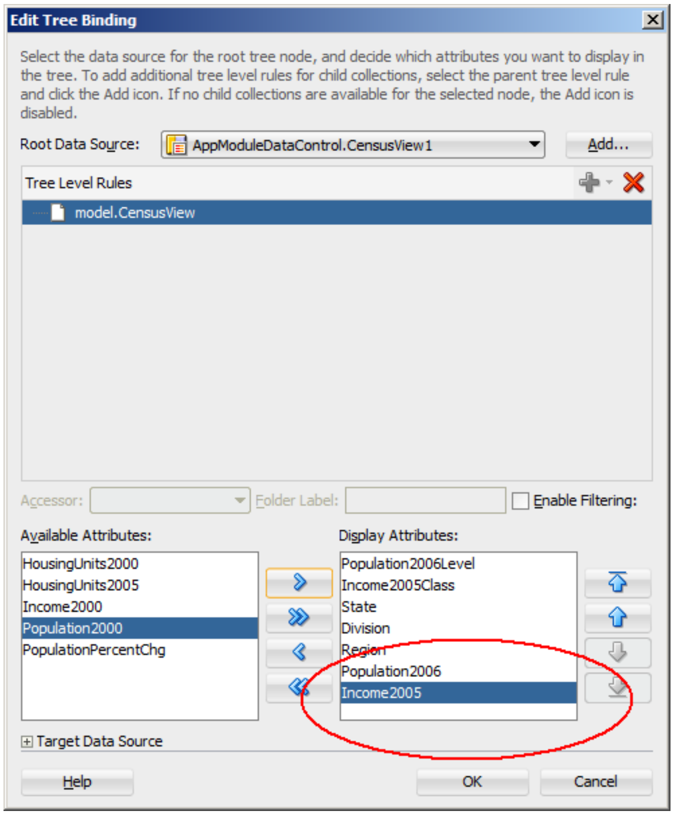
Figure 39-6 NBox Tooltip



For the tooltip to evaluate properly, you must add the `Population2006 and Income2005` attributes to the binding definition for the NBox. To add the binding, in the Structure window, right-click the **dvt:nBox** component and choose **Go to Binding**. Choose **Edit** in the **Binding** section to add the `Population2006` and `Income 2005` attributes.

Figure 39-7 shows the completed Edit Tree Binding with `Population2006 and Income2005` added to the display attributes.

Figure 39-7   Edit Tree Binding Dialog for NBox Sample


Edit Tree Binding
Select the data source for the root tree node, and decide which attributes you want to display in the tree. To add additional tree level rules for child collections, select the parent tree level rule and click the Add icon. If no child collections are available for the selected node, the Add icon is disabled.
Root Data Source:
AppModuleDataControl.CensusView1
Add...
Tree Level Rules
model.CensusView
Accessor:
Folder Label:
Enable Filtering:
Available Attributes:
HousingUnits2000
HousingUnits2005
Income2000
Population2000
PopulationPercentChg
Display Attributes:
Population2006Level
Income2005Class
State
Division
Region
Population2006
Income2005
Target Data Source
Help
OK
Cancel

After you click **OK**, the binding will be updated with the added attributes. The following code sample shows the revised binding for the `CensusView10` node definition.

```
<nodeDefinition DefName="model.CensusView" Name="CensusView10">
  <AttrNames>
    <Item Value="Population2006Level"/>
    <Item Value="Income2005Class"/>
    <Item Value="State"/>
    <Item Value="Division"/>
    <Item Value="Region"/>
    <Item Value="Population2006"/>
    <Item Value="Income2005"/>
  </AttrNames>
</nodeDefinition>
```

Code on the JSF page for an NBox Component

The following example shows the code that is generated on the JSF page for the NBox:

```
<dvt:nBox id="nb1" var="ent" rowsTitle="#{viewcontrollerBundle.POPULATION}"
          value="#{bindings.CensusView1.collectionModel}"
          columnsTitle="#{viewcontrollerBundle.INCOME}">
  <f:facet name="rows">
    <af:group id="g1">
      <dvt:nBoxRow label="#{viewcontrollerBundle.LOW}" id="nbr1" value="low"/>
      <dvt:nBoxRow label="#{viewcontrollerBundle.MEDIUM}" id="nbr2"
value="medium"/>
```

```
      <dvt:nBoxRow label="#{viewcontrollerBundle.HIGH}" id="nbr3" value="high"/>
    </af:group>
  </f:facet>
  <f:facet name="columns">
    <af:group id="g2">
      <dvt:nBoxColumn label="#{viewcontrollerBundle.LOWER}" id="nbc1"
value="lower"/>
      <dvt:nBoxColumn label="#{viewcontrollerBundle.LO_MID}" id="nbc2" value="lo-
mid"/>
      <dvt:nBoxColumn label="#{viewcontrollerBundle.HI_MID}" id="nbc3" value="hi-
mid"/>
      <dvt:nBoxColumn label="#{viewcontrollerBundle.UPPER}" id="nbc4"
value="upper"/>
    </af:group>
  </f:facet>
  <dvt:nBoxNode column="#{ent.Income2005Class}" row="#{ent.Population2006Level}"
                label="#{ent.State}" id="nbn1">
    <dvt:attributeGroups value="#{ent.Division}" type="color"
                         label="#{ent.Division}"
                         sectionLabel="#{viewcontrollerBundle.DIVISION}"
id="ag1"/>
    <f:facet name="icon">
      <dvt:marker id="m1">
        <dvt:attributeGroups value="#{ent.Region}" type="shape"
                             label="#{ent.Region}"
                             sectionLabel="#{viewcontrollerBundle.REGION}"
                             id="ag2"/>
      </dvt:marker>
    </f:facet>
  </dvt:nBoxNode>
</dvt:nBox>
```

Modifying NBox Properties and Layout

After you create your NBox, you can modify the layout of the component or add additional elements, such as a label, using the Create NBox dialog. To open the dialog, use the **Edit** icon in the Properties window for the nBox component. You can also customize the layout of the Nbox directly in the code, in the visual editor, or by setting values in the Properties window.

For example, to add the tooltip shown in Figure 39-6 to your NBox, you can enter a value for the NBox node's shortDesc attribute in the Properties window or in the code editor.

```
<dvt:nBoxNode column="#{ent.Income2005Class}" row="#{ent.Population2006Level}"
              label="#{ent.State}" id="nbn1"
              shortDesc="Population: #{ent.Population2006}&lt;br/>Income:
#{ent.Income2005}">
    ... contents omitted
</dvt:nBoxNode>
```

For additional information and examples for customizing your NBox, see Using NBox Componentsin *Developing Web User Interfaces with Oracle ADF Faces*.

40

Creating Databound Pivot Table and Pivot Filter Bar Components

This chapter describes how to create pivot tables from data modeled with ADF Business Components, using ADF data controls and ADF Faces components in a Fusion web application. Specifically, it describes how you can use ADF Data Visualization `pivotTable` and `pivotFilterBar` components to create pivot tables that visually represent business data. It describes how to use ADF data controls to create a pivot table with data-first development.
If you are designing your page using simple UI-first development, then you can add the pivot table to your page and configure the data bindings later. For information about the data requirements, tag structure, and options for customizing the look and behavior of the pivot table components, see the Using Pivot Table Components chapter in *Developing Web User Interfaces with Oracle ADF Faces*.

This chapter includes the following sections:

About ADF Data Visualization Pivot Table and Pivot Filter Bar Components

Pivot tables display data in a grid layout with unlimited layers of hierarchically nested row header cells and column header cells. Similar to spreadsheets, pivot tables provide the option of automatically generating subtotals and totals for grid data. Pivot filter bars enhance the Pivot table by selectively displaying data according to the user's requirements.

ADF Data Visualization components provide extensive graphical and tabular capabilities for visually displaying and analyzing business data. Each component needs to be bound to data before it can be rendered since the appearance of the components is dictated by the data that is displayed.

The pivot table component produces a grid that supports multiple layers of data labels on the row edge or the column edge of the grid. An optional pivot filter bar represents a page edge that filters the available pivot table data. This component also provides the option of automatically generating subtotals and totals for grid data. Pivot tables let you pivot data layers from one edge to another to obtain different views of your data. For example, a pivot table might initially display total sales data for products within regions on the row edge, broken out by years on the column edge. If you pivot region and year at runtime, then you end up with total sales data for products within years, broken out by region. At runtime, end users can click buttons that appear in the inner column labels to sort rows in ascending or descending order.

The prefix `dvt:` occurs at the beginning of each data visualization component name indicating that the component belongs to the ADF Data Visualization Tools (DVT) tag library.

Data Visualization Components Use Cases and Examples

For detailed descriptions of each data visualization use cases and examples, see the Pivot Table and Pivot Filter Bar Component Use Cases and Examples section in *Developing Web User Interfaces with Oracle ADF Faces*.

End User and Presentation Features

Visually compelling data visualization components enable end users to understand and analyze complex business data. The components are rich in features that provide out-of-the-box interactivity support. For detailed descriptions of the end user and presentation features for each component, see the End User and Presentation Features of Pivot Table Components section in *Developing Web User Interfaces with Oracle ADF Faces*.

Additional Functionality for Data Visualization Components

You may find it helpful to understand other **Oracle ADF** features before you data bind your data visualization components. Additionally, once you have added a data visualization component to your page, you may find that you need to add functionality such as validation and accessibility. Following are links to other functionality that data visualization components use:

- Partial page rendering: You may want a data visualization component to refresh to show new data based on an action taken on another component on the page. For more information, see the Rerendering Partial Page Content chapter in *Developing Web User Interfaces with Oracle ADF Faces*.
- Personalization: Users can change the way the data visualization components display at runtime, those values will not be retained once the user leaves the page unless you configure your application to allow user customization. For more information, see the Allowing User Customization on JSF Pages chapter in *Developing Web User Interfaces with Oracle ADF Faces*.
- Accessibility: By default, data visualization components are accessible. You can make your application pages accessible for screen readers. For more information, see the Developing Accessible ADF Faces Pages chapter in *Developing Web User Interfaces with Oracle ADF Faces*.
- Skins and styles: You can customize the appearance of data visualization components using an ADF skin that you apply to the application or by applying CSS style properties directly using a style-related property (`styleClass` or `inlineStyle`). For more information, see the Customizing the Appearance Using Styles and Skins chapter in *Developing Web User Interfaces with Oracle ADF Faces*.
- Placeholder data controls: If you know the data visualization components on your page will eventually use ADF data binding, but you need to develop the pages before the data controls are ready, then you should consider using placeholder data controls, rather than manually binding the components. Using placeholder data controls will provide the same declarative development experience as using developed data controls. For more information, see Designing a Page Using Placeholder Data Controls.

Creating Databound Pivot Tables

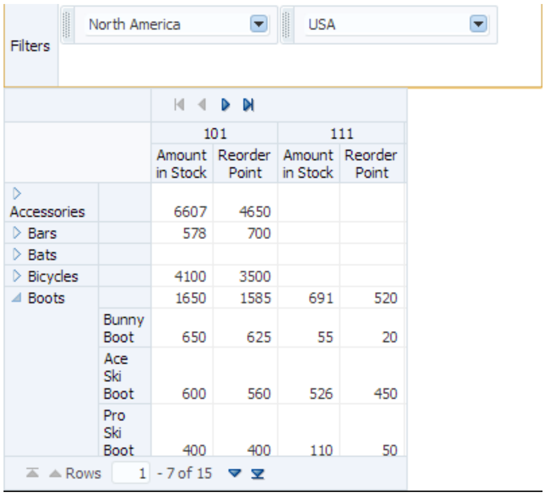
DVT Pivot Table components can be created and bound to a data collection with a data-first development approach. JDeveloper allows you to do this declaratively by dragging and dropping a collection from the Data Controls panel.

The ADF pivot table displays data in a grid layout with unlimited layers of hierarchically nested row header cells and column header cells. The pivot table supports an optional pivot filter bar, representing a page edge that filters the available pivot table data. The pivot table has the following structure:

- Column edge: The horizontal axis above the pivot table containing one or more layers of information in the pivot table.
- Row edge: The vertical axis to the side of the pivot table containing one or more layers of information in the pivot table.
- Page edge: The optional pivot filter bar containing zero or more layers of information for filtering the display of data in the pivot table.
- Data body: One or more measures, or data values, displayed in the cells of the pivot table.

Figure 40-1 shows a Product Inventory pivot table that displays data values for the amount in stock and reorder point in the data body, a warehouse ID data layer on the column edge, and product category and product data layers on the row edge. A pivot filter bar displays a world region and region filter on the page edge.

Figure 40-1 Product Inventory Pivot Table



Filters: North America | USA

		101		111	
		Amount in Stock	Reorder Point	Amount in Stock	Reorder Point
Accessories		6607	4650		
Bars		578	700		
Bats					
Bicycles		4100	3500		
Boots		1650	1585	691	520
	Bunny Boot	650	625	55	20
	Ace Ski Boot	600	560	526	450
	Pro Ski Boot	400	400	110	50

Rows 1 - 7 of 15

A Create Pivot Table wizard provides declarative support for data-binding and configuring the pivot table. In the wizard pages you can:

- Specify the initial layout of the pivot table
- Associate and configure a pivot filter bar
- Specify alternative labels for the data layers

- Configure insert or filter drilling
- Define aggregation of data values
- Configure category and data sorting

As you lay out the pivot table in the first page of the wizard, corresponding entries are initialized in the following wizard pages. You can use the **Back** and **Next** buttons to adjust the pivot table as you go through the wizard pages. You can also skip configuration options in later wizard pages by clicking **Finish**.

For information about customizing a pivot table after data binding is completed, see the Using Pivot Table Components chapter in *Developing Web User Interfaces with Oracle ADF Faces*.

How to Create a Pivot Table Using ADF Data Controls

To create a pivot table using a data control, you bind the pivot table component to a collection. JDeveloper allows you to do this declaratively by dragging and dropping a collection from the Data Controls panel.

> **Tip:**
>
> You can also create a pivot table by dragging a pivot table component from the Components window. This approach allows you the option of designing the pivot table user interface before binding the component to data.

Before you begin:

It may be helpful to have an understanding of databound pivot tables and pivot filter bars. For more information, see Creating Databound Pivot Tables.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Data Visualization Components.

You will need to complete these tasks:

- Create an **application module** that contains instances of the **view objects** that you want in your data model for the pivot table, as described in Creating and Modifying an Application Module.

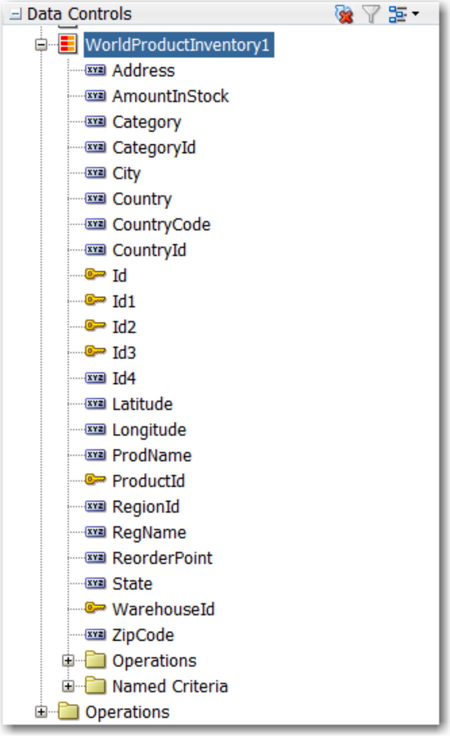
  For example, the data source for Product Inventory pivot table shown in Figure 40-1 comes from a view object created for the Summit sample application for ADF DVT components.
- Create a JSF page as described in the How to Create JSF Pages section of *Developing Web User Interfaces with Oracle ADF Faces*.

To create a databound pivot table:

1. From the Data Controls panel, select a collection.

   For example, to create a pivot table and pivot filter bar that displays product inventory levels in warehouses throughout the World, you could select the `WorldProductInventory1` collection in the Data Controls panel, as shown in Figure 40-2.

Figure 40-2 Data Collection for Product Inventory Levels



2. Drag the data collection onto a JSF page and, from the context menu, choose **Tables/List View** > **ADF Pivot Table**.

3. In the Select Display Attributes page of the Create Pivot Table wizard, specify the initial layout of the pivot table by doing the following:

   a. If you want to associate a pivot filter bar with your pivot table, select **Create Pivot Filter Bar**. Optionally, you can drag attributes from the **Available Attributes** list to the page edge to configure the initial display of filters; otherwise, an empty pivot filter bar is created.

   > **Note:**
   >
   > You can add a pivot filter bar after completing the wizard by right-clicking the `pivotTable` node in the Structure window, and choosing **Insert Before Pivot Table** > **ADF Data Visualizations** > **Pivot Filter Bar**.
   >
   > To remove a pivot filter bar, in the Structure window, right-click the `pivotFilterBar` node and choose **Delete**.

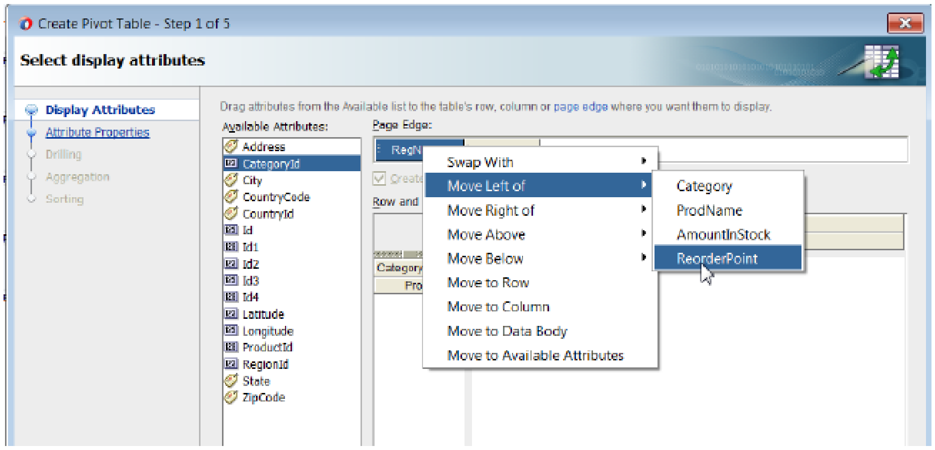
   b. For the initial layout, select the attributes for the pivot table's columns, rows, page edge, and data body by dragging the attributes from the **Available Attributes** list to the pivot table layout.

      In the pivot table layout, **Data Labels** refers to a layer of the pivot table that identifies the data in the cells (data values), and also appears as header labels in the row, column, or page edge. Labels for attributes that you drag to the data body of the pivot table appear in the data labels layer.

You can drag data labels to any location on the row, column, or page edge. You can also drag attributes to different locations on the same edge or on another edge.

As an alternative to using a drag operation to place or move attributes in the layout, you can right-click the attribute or use Shift+F10 to display a context menu of options. Figure 40-3 shows the context menu options for the RegName attribute.

Figure 40-3 Display Attributes Context Menu



Create Pivot Table - Step 1 of 5

Select display attributes

Drag attributes from the Available list to the table's row, column or page edge where you want them to display.

> **Note:**
>
> Potential drill paths between attributes are defined as you lay out multiple attributes on the row, column, and page edges. These drill paths can later be enabled to support pivot table drilling at runtime.

c. If you want to change from the default selection of **Typed Attributes** to **Name-Value Pairs** to configure how data points are stored in a collection, then click the **Change Data Shape** button. A dialog appears that presents you with the following options:

- **Typed Attributes**

  Each kind of data point in the collection is represented by a different attribute. This option is also valid when there is only a single kind of data point in the pivot table.

  For example, if you have data points for **Estimated Value** and **Actual Value**, then select **Typed Attributes** only if you have one attribute for the estimated value and a second attribute for the actual value.

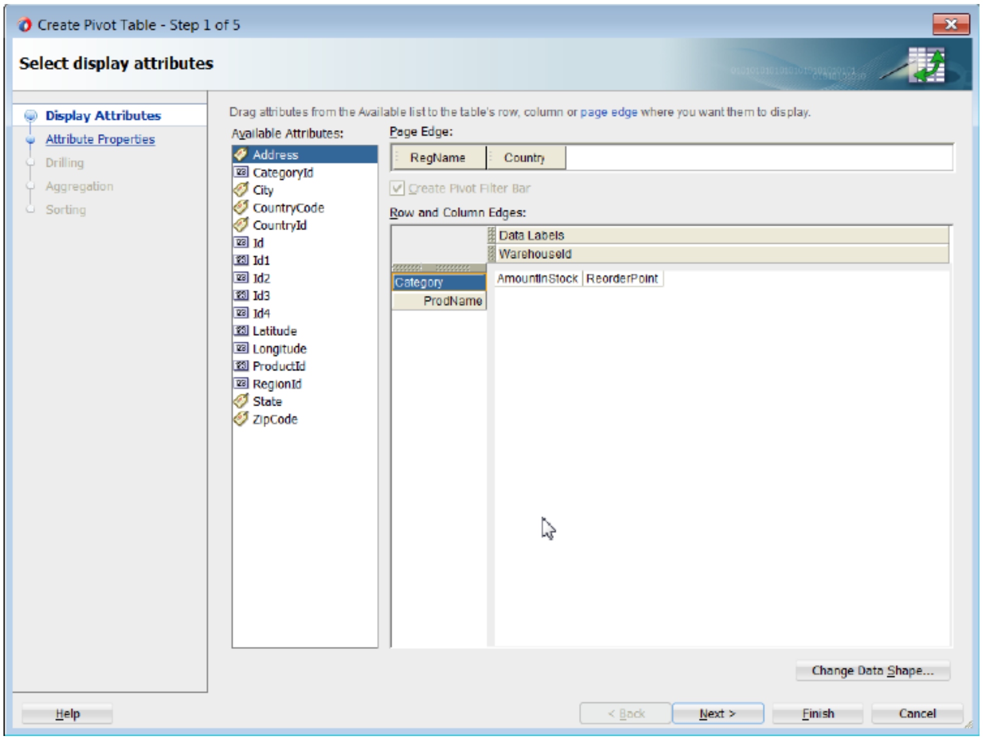
- **Name-Value Pairs**

  Indicates that there are two or more kinds of data points represented by exactly two attributes; a `Name` attribute that specifies the kind of data point, and a `Value` attribute that specifies the data value.

For example, the `Name` attribute might have the value `EST` for a `Value` attribute that represents an estimated value, or the `Name` attribute might have a value `ACT` for a `Value` attribute that represents an actual value.

For example, to specify the initial layout of the Product Inventory pivot table shown in Figure 40-1, you would drag the RegName and Country attributes to the page edge, Category and ProdName attributes to the row edge, AmountinStock and ReorderPoint attributes to the data body (Data Labels), WarehouseID to the column edge, and select **Create Pivot Filter Bar**, as shown in Figure 40-4.

Figure 40-4 Select Display Attributes Page of Create Pivot Table Wizard



Drag attributes from the Available list to the table's row, column or page edge where you want them to display.

Page Edge: RegName | Country

☑ Create Pivot Filter Bar

4. If you want to specify alternative values or labels for the attributes laid out in the Select Display Attributes page of the wizard, click **Next**, and use the Specify Attribute Properties page to do the following:

   a. By default, the data cells in a pivot table are not editable. An `af:outputText` component is automatically stamped for each data cell. If you wish to create a pivot table with editable data cells, deselect **Read-Only Pivot Table** and an `af:inputText` component option will be available for each data cell attribute in the **Component** column. You can then specify zero or more data values as editable by selecting the `af:inputText` component, and the tag and its corresponding `f:validator` tag will be stamped for that pivot table data cell. For more information, see What You May Need to Know About Configuring Editable Data Cells.

   b. To specify alternative labels for data values in the **Data Values** area, change the default `Use Data Attribute Name` text label stamped in the header cell for the attribute at runtime. You can enter the text directly, select `No Label` to suppress the header cell as in the case of using a single data value for the pivot table, specify a text resource from a resource bundle, or use the EL Expression builder to evaluate the label text at runtime.

c. To specify alternative labels for attribute categories in the **Categories** area, change the default `Use Attribute Name` text label stamped in the header cell for the attribute at runtime in the **Attribute Display Name** column. You can enter the text directly, specify a text resource from a resource bundle, or use the EL Expression builder to evaluate the label text at runtime. The label displays in the pivot handle at runtime.

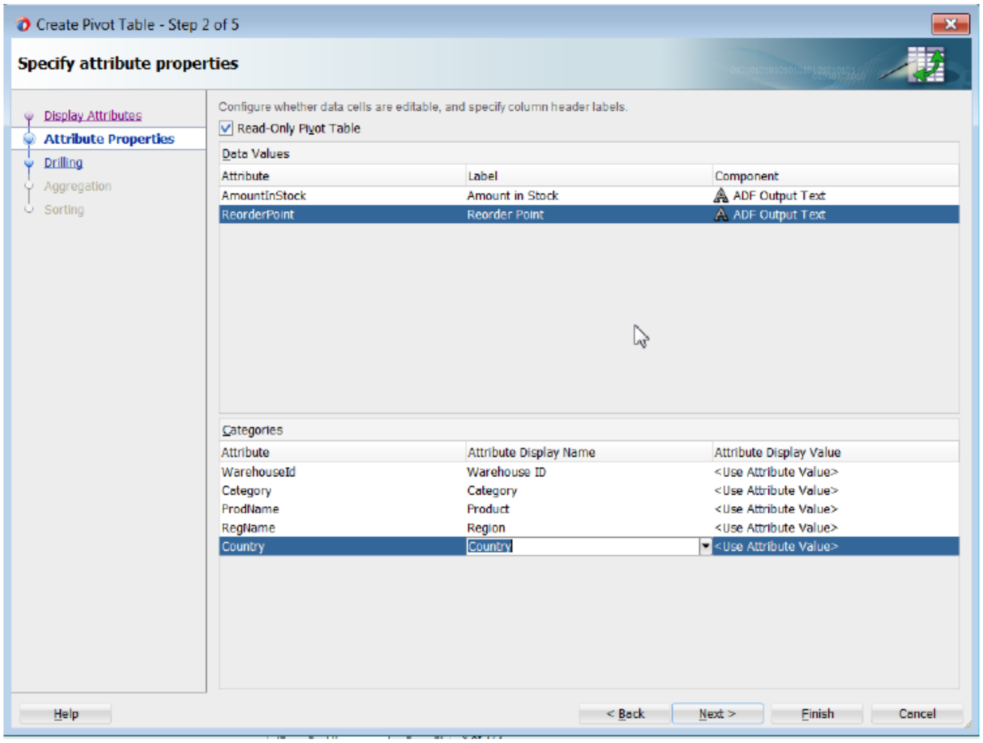
   You can also specify an alternative value for an attribute category by selecting a different attribute in the **Attribute Display Value** column. For example, you might use a `RegionId` attribute in the data collection to lay out the pivot table, but you want the `RegionName` attribute values to appear in the pivot table header at runtime to make the information more readable.

Setting a text resource enables your JSF page or application to display the correct language for the language setting of a user's browser. Choose the **Select Text Resource** option for the **Data Values** labels and the **Categories** attribute display name fields to set a translatable text resource.

> **Note:**
>
> If you configured **Project Properties** > **Resource Bundle** page to **Automatically Synchronize Bundle**, then you can type an alternate label string, and the design time code will create a translatable text resource for you.

For example, to set the Product Inventory pivot table shown in Figure 40-1 to use text resources for labels and display names, complete the Specify Attributes Properties page of the wizard as shown in Figure 40-5.

Figure 40-5 Specify Attribute Properties Page of Create Pivot Table Wizard



Create Pivot Table - Step 2 of 5

Specify attribute properties

Configure whether data cells are editable, and specify column header labels.

☑ Read-Only Pivot Table

Data Values

Attribute	Label	Component
AmountInStock	Amount in Stock	ADF Output Text
ReorderPoint	Reorder Point	ADF Output Text

Categories

Attribute	Attribute Display Name	Attribute Display Value
WarehouseId	Warehouse ID	<Use Attribute Value>
Category	Category	<Use Attribute Value>
ProdName	Product	<Use Attribute Value>
RegName	Region	<Use Attribute Value>
Country	Country	<Use Attribute Value>

5. If you want to expose drill operations in the pivot table at runtime, click **Next**, and use the Configure Drilling page of the Create Pivot Table wizard to enable one of the following options:

   - Select **Insert Drilling** to provide a collapsed or expanded view of the detail data while preserving the sibling and aggregate data. At runtime, a drill icon is displayed in the parent attribute display label.

     Use **Insert Parent Row** to specify whether the aggregate total for the parent attribute will be displayed before or after the child attributes in the expanded view.

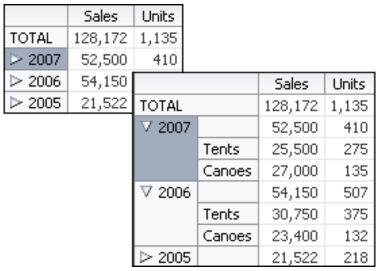
     To enable insert drilling you must also:

     – Select the drill paths to enable. Drill paths are configured based upon the layout of the attributes in the Select Display Attributes page of the wizard.

     – Configure aggregation in the Configure Aggregation page of the wizard.

     For example, Figure 40-6 shows a pivot table using insert drilling to expand the view for the Year data layer. The aggregated value of Sales (52,500 in 2007, 544,150 in 2006) and Units (410 in 2007, 507 in 2006) for each year is displayed in the row above the products.

Figure 40-6 Pivot Table with Insert Drilling Enabled



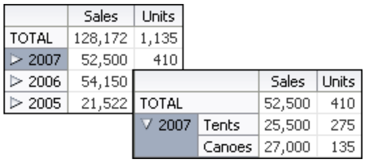
	Sales	Units
TOTAL	128,172	1,135
▷ 2007	52,500	410
▷ 2006	54,150	
▷ 2005	21,522	

		Sales	Units
TOTAL		128,172	1,135
▽ 2007		52,500	410
	Tents	25,500	275
	Canoes	27,000	135
▽ 2006		54,150	507
	Tents	30,750	375
	Canoes	23,400	132
▷ 2005		21,522	218

- Select **Filter Drilling** to provide a collapsed or expanded view of the detail data without preserving the sibling or aggregate data. At runtime, a drill icon is enabled in the parent attribute display label.

  Filter drilling focuses the view on the details of the data layer attribute. For example, Figure 40-7 shows a pivot table using filter drilling to expand the view of the Year (2007) data layer, displaying the total Sales (52,500) and Units (410), while filtering out both the data for the other years and the aggregated total for all the years.

Figure 40-7 Pivot Table with Filter Drilling Enabled



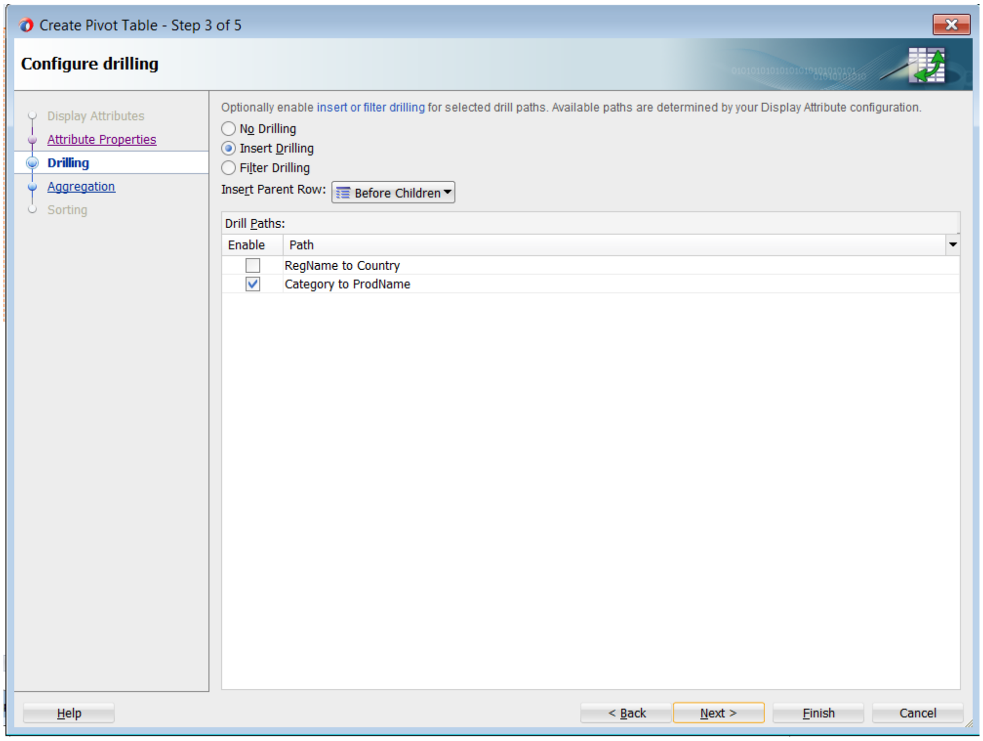
	Sales	Units
TOTAL	128,172	1,135
▷ 2007	52,500	410
▷ 2006	54,150	
▷ 2005	21,522	

		Sales	Units
TOTAL		52,500	410
▽ 2007	Tents	25,500	275
	Canoes	27,000	135

  To enable filter drilling you must select the drill paths to enable. Drill paths are configured based upon the layout of the attributes in the Select Display Attributes page of the wizard.

For example, to enable the insert drilling for the Product Inventory pivot table shown in Figure 40-1, complete the Configure Drilling page of the wizard, as shown in Figure 40-8.

Figure 40-8 Configure Drilling Page of Create Pivot Table Wizard



Create Pivot Table - Step 3 of 5

Configure drilling

Optionally enable insert or filter drilling for selected drill paths. Available paths are determined by your Display Attribute configuration.

○ No Drilling
● Insert Drilling
○ Filter Drilling

Insert Parent Row: Before Children

Drill Paths:

Enable	Path
□	RegName to Country
☑	Category to ProdName

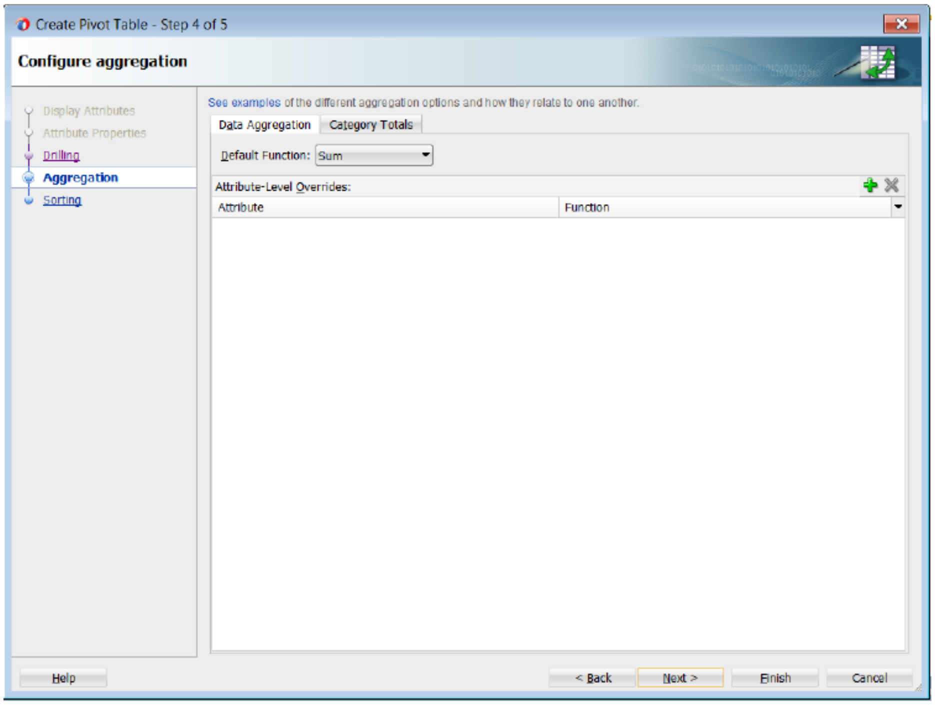
6. If you want to define how data is aggregated in totals and subtotals for the pivot table, click **Next**, and use one or both of the Configure Aggregation pages of the Create Pivot Table wizard.

   By default, if the attributes displayed in the pivot table do not uniquely identify each row in the data collection, the data from duplicate rows is aggregated to collapse that data into a single pivot table cell. You can also override the default aggregate type for a particular data item.

   - If you want to specify how data is aggregated in the pivot table, in the Data Aggregation page, do the following:
     - If you want to change the default aggregation method for handling duplicate rows, use the **Default Function** dropdown list to specify the value. Valid values are `Sum`, `Average`, `Count`, `Maximum`, `Minimum`, `Standard Deviation`, `Median`, and `Variance`.
     - If you want to override the default aggregate type for a specific data value, click the **Add** icon to insert a row for the available attributes. Then, in the **Function** column for each attribute, select the mathematical operation that you want to use for the aggregation. Available options are `Sum`, `Average`, `Count`, `Maximum`, `Minimum`, `Standard Deviation`, `Median`, and `Variance`. This attribute is useful only when you have multiple data values (such as Sales and Units) bound to your pivot table.

     For example, to override the default aggregation type for the Units data value in the Sales pivot table shown in Figure 40-1, use the **Add** icon to add the Units attribute and select Average in the **Function** column in the Data Aggregation page, as shown in Figure 40-9.

Figure 40-9   Data Aggregation Page of Create Pivot Table Wizard



Create Pivot Table - Step 4 of 5

Configure aggregation

See examples of the different aggregation options and how they relate to one another.

Default Function: Sum

- You can also define totals and subtotals for attribute categories added to the column, row, or page edges in the pivot table. In the Categories Totals page, use the **Add** icon to insert each attribute or select **Aggregate All** to add all available attributes, and do the following:
  - In the **Attribute** column, select the attribute that you want to total.
  - In the **Function** column, select the mathematical operation that you want to use for the aggregation. Available options are `Sum`, `Average`, `Count`, `Maximum`, `Minimum`, `Standard Deviation`, `Median`, and `Variance`.
  - In the **Insert Total** column, select the value that indicates where you want the aggregate display to appear relative to the item referenced in the **Attribute** column. Valid values are: `Before`, `After`, or `Replace`.
  - In the **Total Label** column, enter the text that you want to use as a label for the aggregation. You can enter the text directly, specify a text resource from a resource bundle, or use the EL Expression builder to evaluate the label text at runtime.

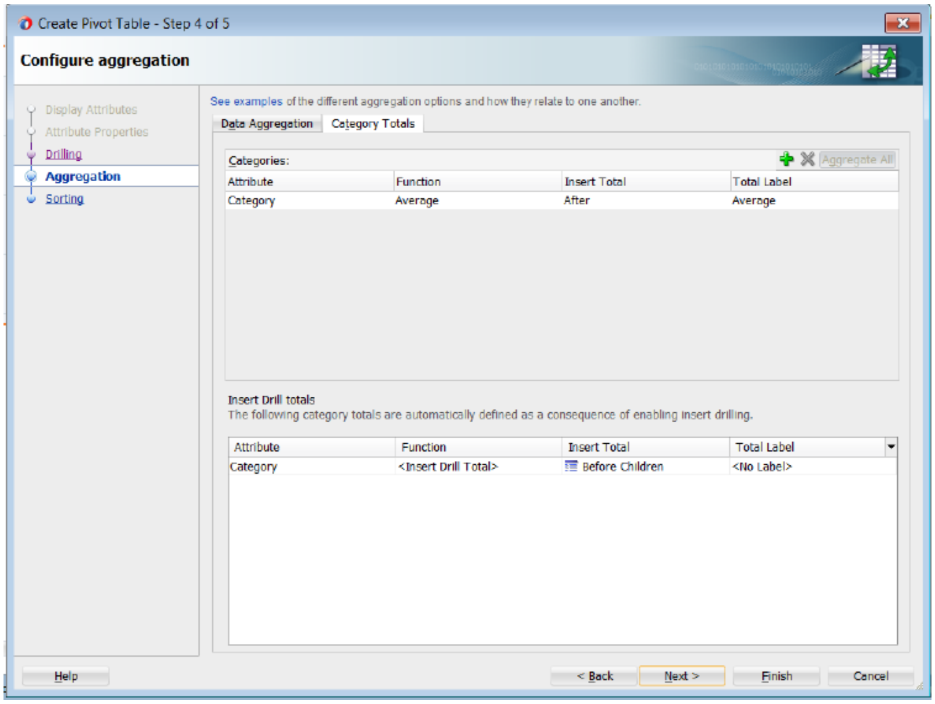
> **Note:**
>
> The read-only **Insert Drill Totals** table displays the category totals automatically defined as a consequence of enabling insert drilling on the pivot table.

For example, to define totals for the Geography and Year data layers in the Sales pivot table shown in Figure 40-1, select `Sum` in the **Function** column and `After` in the **Insert Total** column, and enter text (Total Geography and

Total Year) in the **Total Labels** column respectively for each attribute in the Categories Totals page, as shown in Figure 40-10.

In the resulting pivot table at runtime, expanding a particular Year value will automatically preserve the aggregate total computed from its child value based on the layout and configuration of the insert drill option in the previous wizard page.

Figure 40-10 Categories Totals Page of the Create Pivot Table Wizard



Create Pivot Table - Step 4 of 5

Configure aggregation

See examples of the different aggregation options and how they relate to one another.

Categories:

Attribute	Function	Insert Total	Total Label
Category	Average	After	Average

Insert Drill totals
The following category totals are automatically defined as a consequence of enabling insert drilling.

Attribute	Function	Insert Total	Total Label
Category	<Insert Drill Total>	Before Children	<No Label>

7. If you want to configure sorting in the pivot table, click **Next**, and use one or both of the Configure Sorting pages in the Create Pivot Table wizard.

   By default, a pivot table initially sorts data based on values in the outer row data layer. You can specify sort order on the data layer of any row, column, or page edge, called a category sort. At runtime, when the data layer is pivoted to a different edge, the specified category sort order is honored.

   You cannot specify a category sort of data labels (data values), although you can order the attributes mapped to the data body in the Select Display Attributes page of the wizard. For example, Figure 40-4 shows a pivot table layout with data values for Sales and Units. While you cannot specify a category sort of these measures, you can specify the order in which the values will appear in the data body of the pivot table at runtime, shown in Figure 40-1.

   You can also specify an initial sort order of the data values in the data body when the pivot table is rendered, called a data sort.

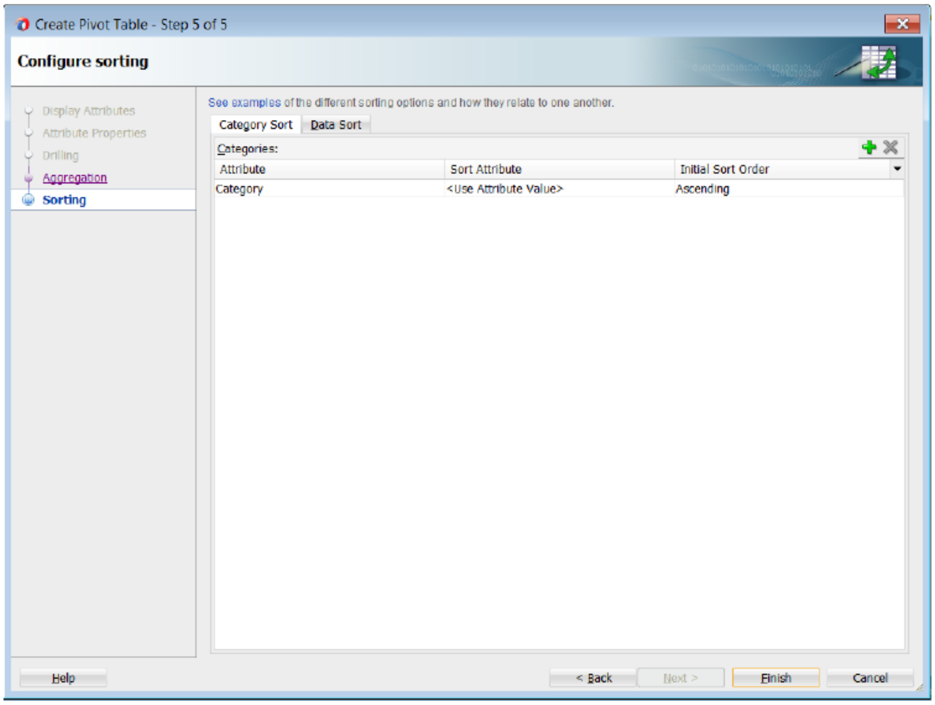
   - To configure sorting by category, in the Category Sort page, use the **Add** icon to add the attribute for each row, column, or page edge you wish to configure, and do the following:
     – In the **Sort Attribute** column, accept the default `Use Attribute Value` to specify an alphabetical sort based on the actual values in the pivot table header, or customize the sort order by specifying an alternate sort

order attribute from the dropdown list. For example, if the underlying query included a rank calculation for ranking products by profitability, you could choose to see products ordered by (ProductRank, Descending).

- In the **Initial Sort Order** column, select the initial direction of the sort. Valid values are `ASCENDING` or `DESCENDING`.

For example, Figure 40-11 shows the Category Sort page of the wizard configured to display the Category data layer ascending on the column edge.

Figure 40-11 Category Sort Page of Create Pivot Table Wizard



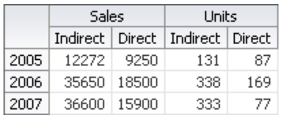
See examples of the different sorting options and how they relate to one another.

Categories:

Attribute	Sort Attribute	Initial Sort Order
Category	<Use Attribute Value>	Ascending

At runtime, the pivot table displays as shown in Figure 40-12.

Figure 40-12 Category Sort Example



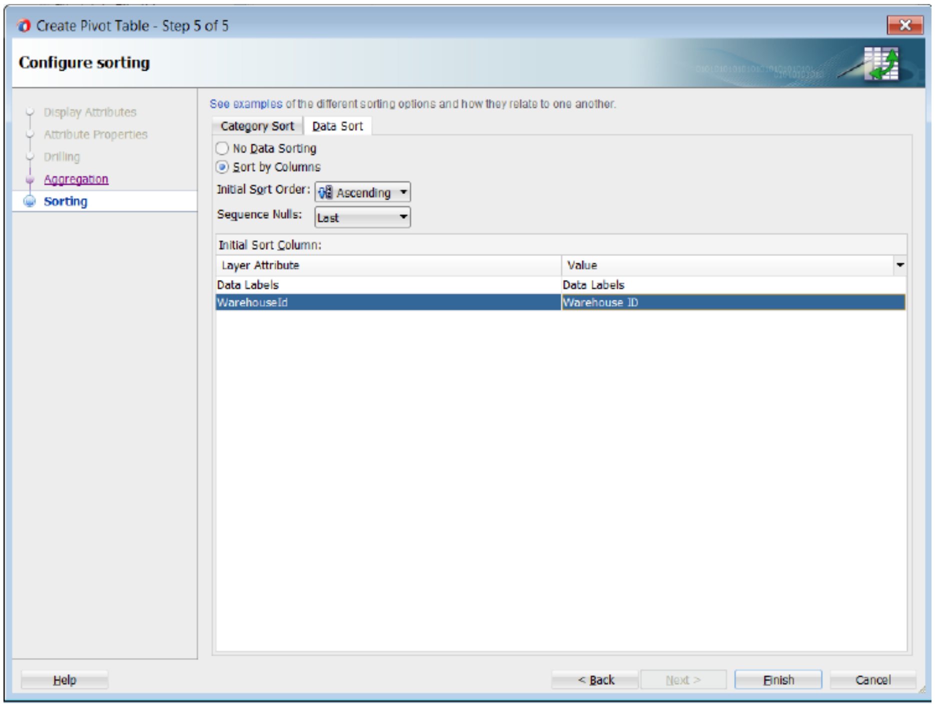
	Sales		Units	
	Indirect	Direct	Indirect	Direct
2005	12272	9250	131	87
2006	35650	18500	338	169
2007	36600	15900	333	77

- To configure data sorting, in the Data Sort page, do the following:
  - Select **Sort by Columns** to specify an initial sort order of the data when the pivot table is rendered.
  - In the **Initial Sort Order** dropdown list select the initial direction of the sort. Valid values are `ASCENDING` and `DESCENDING`.
  - In the **Sequence Nulls** dropdown list, select `First` if you want the null values to appear at the beginning of a sort and select `Last` if you want the null values to appear at the end of the sort.

– In the **Initial Sort Column** table, specify a data value in the **Value** column for each data layer displayed in the **Layer Attribute** column.

For example, Figure 40-13 shows the Data Sort page configured to sort the Channel data layer grouped by Year, based upon Units/World/Canoes data values.

Figure 40-13 Data Sort Page of the Create Pivot Table Wizard



Create Pivot Table - Step 5 of 5

Configure sorting

See examples of the different sorting options and how they relate to one another.

Category Sort | Data Sort

○ No Data Sorting
● Sort by Columns

Initial Sort Order: Ascending

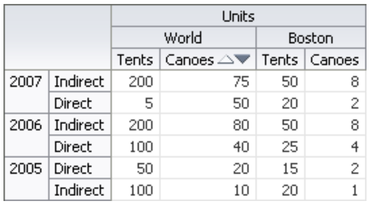
Sequence Nulls: Last

Initial Sort Column:

Layer Attribute	Value
Data Labels	Data Labels
WarehouseId	Warehouse ID

At runtime, the pivot table initially renders as shown in Figure 40-14.

Figure 40-14 Data Sort Example



		Units			
		World		Boston	
		Tents	Canoes △▼	Tents	Canoes
2007	Indirect	200	75	50	8
	Direct	5	50	20	2
2006	Indirect	200	80	50	8
	Direct	100	40	25	4
2005	Direct	50	20	15	2
	Indirect	100	10	20	1

8. Click **Finish** to complete the creation of your databound pivot table.

   After completing the wizard to create the pivot table, use the tools in JDeveloper to customize the look and feel of the pivot table. For example, you can configure word wrapping for labels that do not fit into the default size of the header cell and add a page control as an alternative to scrollbars for the data set in the pivot table in Figure 40-1.

9. To customize pivot table display elements, perform the following tasks.

   a. In the Structure window, right-click the `dvt:pivotTable` node and choose **Go to Properties**.

b. In the Properties window, expand the **Common** section and set the **ScrollPolicy** property to `page` to configure a page control as an alternative to the default scrollbars.

c. In the Structure window, right-click the `dvt:headerCell` node and choose **Go to Properties**.

d. In the Properties window, expand the **Behavior** section and set the **WhiteSpace** attribute to `normal` to configure word wrapping in the pivot table headers.

For additional information about customizing a pivot table after data binding is completed, see the Using Pivot Table Components chapter in *Developing Web User Interfaces with Oracle ADF Faces*.

What Happens When You Use the Data Controls Panel to Create a Pivot Table

Dropping a pivot table from the Data Controls panel has the following effect:

- Creates the bindings for the pivot table and adds the bindings to the page definition file
- Adds the necessary code for the UI components to the JSF page

When you create a pivot table from the Data Controls panel, the page definition file is updated with the bindings. The code sample below shows the row set bindings that were generated for the pivot table that displays product sales and units sold within geography by year. The pivot table data map contains the following elements:

- `<columns>`: Defines each column item in the appropriate sequence
- `<rows>`: Defines each row item in the appropriate sequence
- `<pages>`: Defines the items to be included in the pivot filter bar
- `<aggregatedItems>`: Defines the totals and subtotals of items
- `<hierarchies>`: Defines the potential drill paths between two items
- `<sorts>`: Defines category sorts and the initial sort order of pivot table data

The default data aggregation method for duplicate rows is specified in the `<data>` element. For more information about aggregating duplicates, see What You May Need to Know About Aggregating Attributes in the Pivot Table.

For more information about sorting operations, see What You May Need to Know About Specifying an Initial Sort for a Pivot Table.

```
<pivotTable IterBinding="WorldProductInventory1Iterator"
            id="WorldProductInventory1"
            xmlns="http://xmlns.oracle.com/adfm/dvt" ChangeEventPolicy="ppr">
  <pivotTableDataMap convert="false">
    <columns>
      <data aggregateDuplicates="true" defaultAggregateType="SUM">
        <item label="${adfBundle['view.ViewControllerBundle'].AMOUNT_IN_STOCK}"
              value="AmountInStock"/>
        <item label="${adfBundle['view.ViewControllerBundle'].REORDER_POINT}"
              value="ReorderPoint"/>
      </data>
        <item value="WarehouseId"
              itemLabel="$
```

```
{adfBundle['view.ViewControllerBundle'].WAREHOUSE_ID}"/>
    </columns>
    <rows>
      <item value="Category"
            itemLabel="${adfBundle['view.ViewControllerBundle'].CATEGORY}"/>
    </rows>
    <pages>
      <item value="RegName"
            itemLabel="${adfBundle['view.ViewControllerBundle'].REGION}"/>
      <item value="Country"
            itemLabel="${adfBundle['view.ViewControllerBundle'].COUNTRY}"/>
    </pages>
    <aggregatedItems>
      <item aggregateLocation="AFTER" aggregateType="AVERAGE" value="Category"
            aggregateLabel="${adfBundle['view.ViewControllerBundle'].AVERAGE}"/>
    </aggregatedItems>
    <drills type="INSERT"/>
    <hierarchies>
      <item value="Category" location="BEFORE">
        <child value="ProdName"
               itemLabel="${adfBundle['view.ViewControllerBundle'].PRODUCT}"/>
      </item>
    </hierarchies>
    <sorts>
      <categorySort item="Category" direction="ASCENDING"/>
    </sorts>
  </pivotTableDataMap>
</pivotTable>
```

When the pivot table is created using the Data Controls panel, the necessary code is added to the page. The example below shows the code generated on the JSF page for the sales pivot table and associated pivot filter bar.

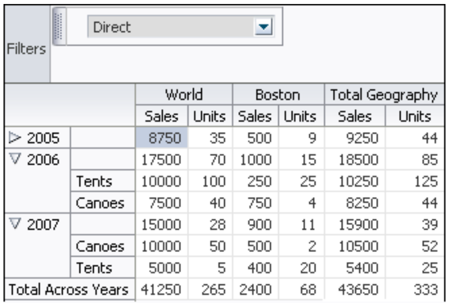
```
<dvt:pivotFilterBar id="pfb1"

value="#{bindings.WorldProductInventory1.pivotFilterBarModel}"
                    modelName="pt1Model"/>
<dvt:pivotTable id="pt1"
                value="#{bindings.WorldProductInventory1.pivotTableModel}"
                modelName="pt1Model"
                var="cellData" varStatus="cellStatus"
                summary="#{viewcontrollerBundle.WorldProductInventoryPivotTable}"
                scrollPolicy="page">
  <dvt:headerCell whiteSpace="normal">
    <af:switcher facetName="#{cellData.layerName}" defaultFacet="Default"
id="s1">
      <f:facet name="DataLayer">
        <af:outputText value="#{cellData.label}" id="ot1"/>
      </f:facet>
      <f:facet name="WarehouseId">
        <af:outputText value="#{cellData.dataValue}" id="ot2">
          <af:convertNumber groupingUsed="false"
                            pattern="#{bindings.WorldProductInventory1.
                                     hints.WarehouseId.format}"/>
        </af:outputText>
      </f:facet>
      <f:facet name="Category">
        <af:outputText value="#{cellData.dataValue}" id="ot3"/>
      </f:facet>
      <f:facet name="ProdName">
        <af:outputText value="#{cellData.dataValue}" id="ot4"/>
      </f:facet>
      <f:facet name="RegName">
        <af:outputText value="#{cellData.dataValue}" id="ot5"/>
      </f:facet>
      <f:facet name="Country">
        <af:outputText value="#{cellData.dataValue}" id="ot6"/>
      </f:facet>
      <f:facet name="Default">
        <af:outputText value="#{cellData.dataValue}" id="ot7"/>
      </f:facet>
    </af:switcher>
  </dvt:headerCell>
  <dvt:dataCell>
    <af:switcher facetName="#{cellStatus.members.DataLayer.value}"
                 defaultFacet="Default" id="s2">
      <f:facet name="AmountInStock">
        <af:outputText value="#{cellData.dataValue}" id="ot8">
          <af:convertNumber groupingUsed="false"
                            pattern="#{bindings.WorldProductInventory1.
                                     hints.AmountInStock.format}"/>
        </af:outputText>
      </f:facet>
      <f:facet name="ReorderPoint">
        <af:outputText value="#{cellData.dataValue}" id="ot9">
          <af:convertNumber groupingUsed="false"
                            pattern="#{bindings.WorldProductInventory1.
                                    hints.ReorderPoint.format}"/>
        </af:outputText>
      </f:facet>
      <f:facet name="Default">
        <af:outputText value="#{cellData.dataValue}" id="ot10"/>
      </f:facet>
    </af:switcher>
  </dvt:dataCell>
</dvt:pivotTable
```

What You May Need to Know About Aggregating Attributes in the Pivot Table

If the attributes that you choose to display in your pivot table do not uniquely identify each row in your data collection, then you can aggregate the data from duplicate rows to collapse that data into a single pivot table cell.

For example, if the rows in the data collection shown in Figure 40-15 also contained a store identification, then the data rows from all stores in a given combination of Product, Channel, and Geography would have to be collapsed into a single cell in the pivot table.

Figure 40-15   Sales Pivot Table



Filters: Direct ▾

		World		Boston		Total Geography	
		Sales	Units	Sales	Units	Sales	Units
▷ 2005		8750	35	500	9	9250	44
▽ 2006		17500	70	1000	15	18500	85
	Tents	10000	100	250	25	10250	125
	Canoes	7500	40	750	4	8250	44
▽ 2007		15000	28	900	11	15900	39
	Canoes	10000	50	500	2	10500	52
	Tents	5000	5	400	20	5400	25
Total Across Years		41250	265	2400	68	43650	333

The pivot table has the following optional data binding attributes available for controlling the calculation of duplicate data rows:

- `aggregateDuplicates`: Boolean property of the `<data>` element that determines whether special processing is enabled at binding runtime to aggregate data values in duplicate rows. If this attribute is not specified, then `false` is assumed.
- `defaultAggregateType`: String property of the `<data>` element that specifies a default aggregation method for handling duplicates. Valid values are `SUM`, `AVERAGE`, `COUNT`, `MIN`, `MAX`, `STDDEV`, `MEDIAN`, `VARIANCE`. If `aggregateDuplicates` is `true` and `defaultAggregateType` is unspecified, then `SUM` is assumed.
- `aggregateType`: String property of an `<item>` element that enables you to override the default aggregate type for a particular data item. This attribute is useful only when you have multiple data values (such as Sales and Units) bound to your pivot table.

Default Aggregation of Duplicate Data Rows

By default, the pivot table uses the SUM operation to aggregate the data values of duplicate data rows in a data collection to produce a single cell value in the pivot table. This means that the `aggregateDuplicates` attribute is set to `true` and the `defaultAggregateType` is assumed to be `SUM`.

The `<data>` element shown in the example below is an example of such default aggregation.

```
<pivotTable IterBinding="ptExampleDataIterator" id="ptExampleData"
            xmlns="http://xmlns.oracle.com/adfm/dvt"
            ChangeEventPolicy="ppr">
  <pivotTableDataMap>
    <columns>
      <item value="Geography" itemLabel="Location"/>
        <data aggregateDuplicates="true" defaultAggregateType="SUM">
          <item value="Sales"/>
          <item value="Units" aggregateType="AVERAGE"/>
        </data>
    </columns>
    <rows>
      <item value="Year"/>
    </rows>
    <pages>
      <item value="Channel"/>
    </pages>
    <aggregatedItems>
```

```
      <item aggregateLocation="AFTER" aggregateType="SUM" value="Geography"
            aggregateLabel="Total Geography"/>
      <item aggregateLocation="AFTER" aggregateType="SUM" value="Year"
            aggregateLabel="Total Across Years"/>
    </aggregatedItems>
    <drills type="INSERT"/>
    <hierarchies>
      <item value="Year" location="BEFORE">
        <child value="Product" label="Product"/>
      </item>
    </hierarchies>
    <sorts>
      <categorySort item="Channel" direction="DESCENDING"/>
      <categorySort item="Year" direction="ASCENDING"/>
      <qdrSliceSort direction="DESCENDING" edge="rows" grouped="true"
                    nullsFirst="true">
        <item name="Geography" value="World"/>
      </qdrSliceSort>
    </sorts>
  </pivotTableDataMap>
</pivotTable>
```

Custom Aggregation of Duplicate Rows

If you want the pivot table to use a different mathematical operation to aggregate the data values of duplicate rows, then you set the defaultAggregateType to the desired operation.

The example below shows a data element with the defaultAggregateType set to SUM. This operation would be appropriate if you want to see the total of sales from all stores for each unique combination of Product, Channel, and State.

```
<pivotTable IterBinding="SalesPivotTable1Iterator" id="SalesPivotTable11"
            xmlns="http://xmlns.oracle.com/adfm/dvt">
  <pivotTableDataMap>
    <columns>
      <data aggregateDuplicates="true" defaultAggregateType="SUM">
        <item value="Sales"/>
      </data>
      <item value="Geography"/>
    </columns>
    <rows>
      <item value="Channel"/>
      <item value="Product"/>
    </rows>
    <aggregatedItems>
      <item aggregateLocation="After" aggregateType="AVERAGE"
            value="Product" aggregateLabel="Average"/>
    </aggregatedItems>
  </pivotTableDataMap>
</pivotTable>
```

If you have a pivot table with multiple data values (such as sales and the average size of a store in square feet) and you want to sum the sales data values in duplicate rows, but you want to average the square feet data values, then do the following:

- On the <data> element, set the defaultAggregateType to SUM.
- On the <item> element for the square feet attribute, set the aggregateType to AVERAGE.

The example below shows the <columns> elements wrapped by a PivotTableDataMap element. The <data> element contains the default attributes for aggregation. These apply to all data items that do not have a specific custom aggregateType attribute specified.

```
<columns>
   <data aggregateDuplicates="true" defaultAggregateType="SUM">
     <item value="Sales" label="Total Sales"/>
     <item value="StoreSqFeet" label="Avg Sq Feet" aggregateType="AVERAGE"/>
   </data>
   <item value="State"/>
</columns>
```

What You May Need to Know About Specifying an Initial Sort for a Pivot Table

By default, a pivot table initially sorts data based on values in the outer row data layer. You can specify sort order on the data layer of any row, column, or page item, called a category sort. At runtime, when the data layer is pivoted to a different edge, the specified category sort order is honored. Insert a categorySort element inside the sorts element and set values for the attributes as described in Table 40-1.

Table 40-1 Attribute Values for categorySort Element

Attribute	Description
item	Specify the column, row, or page item for which you are setting the category sort. A value for this attribute is required.
direction	Specify the initial direction of the sort. Valid values are ASCENDING and DESCENDING. A value for this attribute is required.

You can also specify the initial sort order of the data values in the data body when the pivot table is rendered, called a data sort. You can change the default behavior by inserting a sorts element inside the pivotTableDataMap element of a pivot table binding in the page definition file. Insert a qdrSliceSort element inside the sorts element and set values for the attributes as described in Table 40-2.

Table 40-2 Attribute Values for qdrSliceSort Element

Attribute	Description
direction	Specify the initial direction of the sort. Valid values are ASCENDING and DESCENDING. A value for this attribute is required.
edge	Specify columns or rows to determine which edge sorts data. A value for this attribute is required.
grouped	Specify true if you want to sort slices within their parent or false if you want to sort across the entire edge. A value for this attribute is optional. The default value is false.
nullsFirst	Specify true if you want null values to appear at the beginning of a sort and false if you want null values to appear at the end of a sort. The default value is false. A value for this attribute is optional.

Insert one or more `item` tags inside the `qdrSliceSort` tag. An item tag specifies the slice on the opposite edge from which the values to be sorted should be obtained. For example, if sorting rows based upon the data, then you must specify an item tag for each layer on the column edge. Set values for the attributes as described in Table 40-3.

Table 40-3   Attribute Values for item Tag

Attribute	Description
`name`	Specify the name of the layer to sort on. Typically, this is the column name in the row set. Specify `DataLayer` to identify the layer that contains the data columns in a row set (for example, Sales, Costs, and so on).
`value`	Specify the value of the specified layer on the desired slice.

What You May Need to Know About Configuring Editable Data Cells

By default, the data cells in a pivot table are not editable. When you create a pivot table using ADF data controls, an `af:outputText` component is automatically stamped for each data cell. You can also use the Create Pivot Table wizard to configure a data cell as editable by specifying an `af:inputText` component and the tag and its corresponding `f:validator` tag will be stamped in the data cell as illustrated in the example below.

```
<dvt:dataCell>
  <af:switcher facetName="#{cellStatus.members.DataLayer.value}"
           defaultFacet="Default" id="s2">
    <f:facet name="AmountInStock">
      <af:inputText value="#{cellData.dataValue}"
                    label="#{bindings.WorldProductInventory1.hints.
                          AmountInStock.label}"
                    required="#{bindings.WorldProductInventory1.hints.
                          AmountInStock.mandatory}"
                    columns="#{bindings.WorldProductInventory1.hints.
                          AmountInStock.displayWidth}"
                    maximumLength="#{bindings.WorldProductInventory1.hints.
                          AmountInStock.precision}"
                    shortDesc="#{bindings.WorldProductInventory1.hints.
                          AmountInStock.tooltip}"
                    id="it1">
        <f:validator binding="#{cellData.bindings.AmountInStock.validator}"/>
      </af:inputText>
    </f:facet>
    <f:facet name="ReorderPoint">
      <af:outputText value="#{cellData.dataValue}" id="ot8">
        <af:convertNumber groupingUsed="false"
                          pattern="#{bindings.WorldProductInventory1.hints.
                                   ReorderPoint.format}"/>
      </af:outputText>
    </f:facet>
    <f:facet name="Default">
      <af:outputText value="#{cellData.dataValue}" id="ot10"/>
    </f:facet>
  </af:switcher>
</dvt:dataCell>

<dvt:dataCell>
  <af:switcher facetName="#{cellStatus.members.DataLayer.value}"
```

```
               defaultFacet="Default" id="s2">
    <f:facet name="Sal">
      <af:inputText value="#{cellData.dataValue}" id="ot5"
                    label="#{bindings.EmpView1.hints.Sal.label}"
                    required="#{bindings.EmpView1.hints.Sal.mandatory}"
                    columns="#{bindings.EmpView1.hints.Sal.displayWidth}"
                    maximumLength="#{bindings.EmpView1.hints.Sal.precision}"
                    shortDesc="#{bindings.EmpView1.hints.Sal.tooltip}">
      <f:validator binding="#{cellData.bindings.Sal.validator}"/>
    </af:inputText>
  </f:facet>
  <f:facet name="Default">
    <af:inputText value="#{cellData.dataValue}" id="ot6"/>
  </f:facet>
    </af:switcher>
</dvt:dataCell>
```

You can also configure pivot table data cells to use most components that implement the `EditableValueHolder` or `ActionSource` interfaces as a child of the `dataCell` component, for example, `af:selectBooleanCheckbox` or `af:inputComboboxListofValues`. For more information, see the How to Configure Header and Data Cell Stamps section in the *Developing Web User Interfaces with Oracle ADF Faces*.

41
Creating Databound Geographic and Thematic Map Components

This chapter describes how to create geographic or thematic maps from data modeled with ADF Business Components, using ADF data controls and ADF Faces components in a Fusion web application. Specifically, it describes how you can use ADF Data Visualization `map` and `thematicMap` components to create geographic or thematic maps that visually represent business data. It describes how to use ADF data controls to create these components with data-first development.
If you are designing your page using simple UI-first development, then you can add the geographic or thematic map to your page and configure the data bindings later. For information about the data requirements, tag structure, and options for customizing the look and behavior of the map components, see the Using Map Components chapter in *Developing Web User Interfaces with Oracle ADF Faces*.

This chapter includes the following sections:

- About ADF Data Visualization Map Components
- Creating Databound Geographic Maps
- Creating Databound Thematic Maps

About ADF Data Visualization Map Components

The DVT Geographic Map and Thematic Map components allow users to present business data on a map. This is useful to represent patterns, trends, and comparative data within a region of interest.

ADF Data Visualization components provide extensive graphical and tabular capabilities for visually displaying and analyzing business data. Each component needs to be bound to data before it can be rendered since the appearance of the components is dictated by the data that is displayed.

The geographic map component represents business data spatially, enabling you to superimpose multiple layers, also referred to as themes, of information on a single map. For example, a map of the United States might use a color theme that provides varying color intensity to indicate the popularity of a product within each state, a pie chart theme that shows sales within product category, and a point theme that identifies the exact location of each warehouse. When all three themes are superimposed on the United States map, you can easily evaluate whether there is sufficient inventory to support the popularity level of a product in specific locations. Geographic maps require a connection to an Oracle MapViewer service, and optionally, a geocoder service to display geographical and political detail.

Thematic map components represents business data as patterns in stylized areas or associated markers and does not require a connection to an Oracle MapViewer service. Thematic maps focus on data without the geographic details in a geographic map. The thematic map is packaged with prebuilt base maps including a USA base map, a world base map, and base maps for continents and regions of the world including EMEA and APAC. Each base map includes several sets of regions and one fixed set of cities. A set of regions or cities is referred to as a layer. Each layer can be bound to a data collection and stylized to represent the data with color and pattern fills, or a data marker, or both. At runtime, only one map layer and its associated data can be displayed at a time, unless the thematic map has been enabled for drilling.

The prefix `dvt:` occurs at the beginning of each data visualization component name indicating that the component belongs to the ADF Data Visualization Tools (DVT) tag library.

Use Cases and Examples

For detailed descriptions of each data visualization use cases and examples, see the Map Component Use Cases and Examples section in *Developing Web User Interfaces with Oracle ADF Faces*.

End User and Presentation Features

Visually compelling data visualization components enable end users to understand and analyze complex business data. The components are rich in features that provide out-of-the-box interactivity support. For detailed descriptions of the end user and presentation features for each component, see the End User and Presentation Features of Maps section in *Developing Web User Interfaces with Oracle ADF Faces*.

Additional Functionality for Data Visualization Components

You may find it helpful to understand other **Oracle ADF** features before you data bind your data visualization components. Additionally, once you have added a data visualization component to your page, you may find that you need to add functionality such as validation and accessibility. Following are links to other functionality that data visualization components use:

- Partial page rendering: You may want a data visualization component to refresh to show new data based on an action taken on another component on the page. For more information, see the Rerendering Partial Page Content chapter in *Developing Web User Interfaces with Oracle ADF Faces*.
- Personalization: Users can change the way the data visualization components display at runtime, those values will not be retained once the user leaves the page unless you configure your application to allow user customization. For more information, see the Allowing User Customization on JSF Pages chapter in *Developing Web User Interfaces with Oracle ADF Faces*.
- Accessibility: By default, data visualization components are accessible. You con configure your application pages for accessibility to screen readers. For more information, see the Developing Accessible ADF Faces Pages chapter in *Developing Web User Interfaces with Oracle ADF Faces*.
- Skins and styles: You can customize the appearance of data visualization components using an ADF skin that you apply to the application or by applying CSS style properties directly using a style-related property (`styleClass` or `inlineStyle`). For more information, see the Customizing the Appearance Using Styles and Skins chapter in *Developing Web User Interfaces with Oracle ADF Faces*.

- Placeholder data controls: If you know the data visualization components on your page will eventually use ADF data binding, but you need to develop the pages before the data controls are ready, then you should consider using placeholder data controls, rather than manually binding the components. Using placeholder data controls will provide the same declarative development experience as using developed data controls. For more information, see Designing a Page Using Placeholder Data Controls.

Creating Databound Geographic Maps

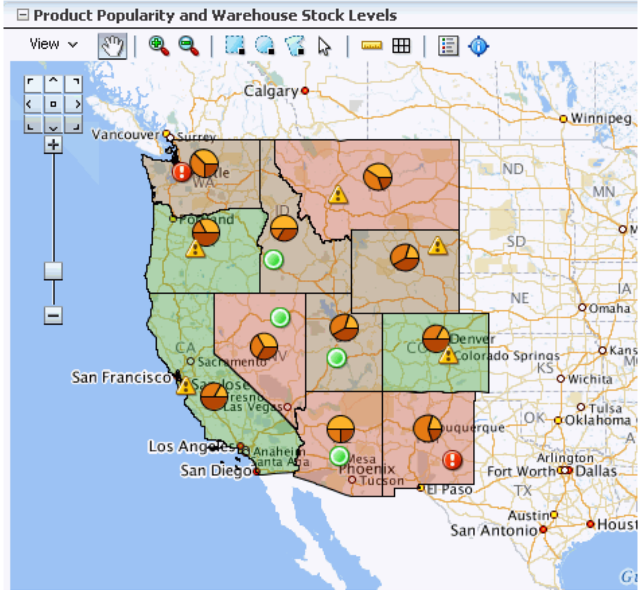
A geographic map is an ADF Data Visualization component that provides the functionality of Oracle Spatial within Oracle ADF. This component allows users to represent business data on a geographic map and to superimpose multiple layers of information (known as themes) on a single map.

These layers can be represented as any of the following themes: bar graph, pie graph, color, point, and predefined theme.

Figure 41-1 shows a geographic map component that uses a base map for a region in the United States with the following themes:

- Color theme: For the selected product, this theme colors states based on product popularity. The colors range from green, representing the highest popularity for that product, to red, representing the lowest popularity for that product.
- Pie graph theme: This theme displays a pie graph in each state to indicate the popular product categories in that state. In this example, the pie graph shows the product categories as pie slices for Media, Office, and Electronics.
- Point theme: This theme identifies warehouses as points. For each point, it displays an icon to indicate the inventory level at that warehouse for the selected product. A separate icon is displayed for each of the following ranges of inventory: low inventory, medium inventory, and high inventory.

Figure 41-1   Geographic Map with Color Theme, Pie Graph Theme, and Point Theme



A geographic map component differs from other ADF Data Visualization components as you do not need to put multiple maps on a page to display multiple sets of data. This contrasts to components such as graphs where you can put multiple graphs on a page. Instead, you show how multiple sets of data relate to each other spatially or, for a specific point, you display different attributes layered in separate themes.

The geographic map component itself is not bound to data. However, each map theme has its own data bindings.

A base map forms the background on which the ADF geographic map component layers the themes that developers create.

In Oracle Spatial, administrators create base maps that consist of one or more themes. The administrator controls the visibility of the base map themes. When you zoom in and out on a base map, various base map themes are hidden or displayed. At the ADF geographic map component level, you cannot use zoom factor to control the display of the themes created by the administrator on the base map.

When you overlay themes on the ADF geographic map, you can control the visibility of your themes by setting the `maxZoom` and `minZoom` properties of the components related to these themes. At runtime, you can also hide or display your custom themes by using the View menu of the Map toolbar or by using other ADF components that you create on the page.

For information about customizing a geographic map after data-binding is completed, see the Using Map Components chapter in *Developing Web User Interfaces with Oracle ADF Faces*.

How to Configure a Geographic Base Map

To create a geographic map, you first configure the map before you bind a point, pie or bar graph, or color theme of the map to a data collection. JDeveloper allows you to do this declaratively by dragging and dropping a collection from the Data Controls panel for the theme you want to create.

Before you begin:

It may be helpful to have an understanding of databound geographic maps. For more information, see Creating Databound Geographic Maps.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Data Visualization Components.

You will need to complete these tasks:

- Create an **application module** that contains instances of the **view objects** that you want in your data model for the pivot table, as described in Creating and Modifying an Application Module.

  For example, the data source for Product Inventory pivot table shown in Figure 41-1 comes from a view object created for the Summit sample application for ADF DVT components.

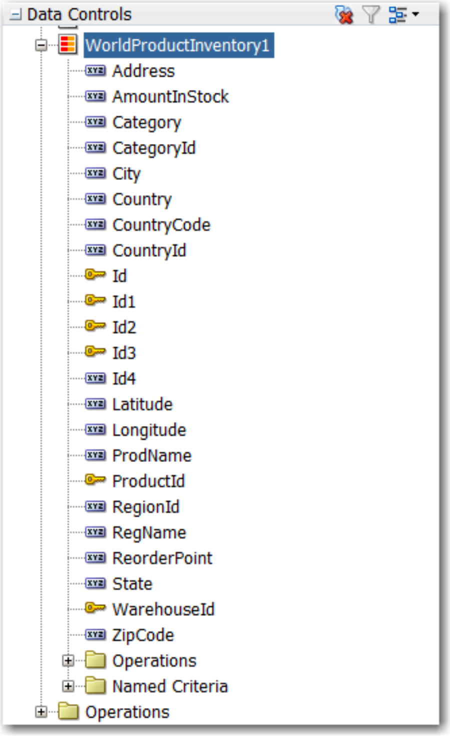
- Create a JSF page as described in the How to Create JSF Pages section of *Developing Web User Interfaces with Oracle ADF Faces*.

To configure a geographic base map:

1. From the Data Controls panel, select a collection.

For example, to create a geographic map that displays product inventory levels in warehouses throughout the World, you could select the `WorldProductInventory1` collection in the Data Controls panel, as shown in Figure 41-2.

Figure 41-2 Data Collection for Product Inventory Levels



Data Controls

- WorldProductInventory1
  - Address
  - AmountInStock
  - Category
  - CategoryId
  - City
  - Country
  - CountryCode
  - CountryId
  - Id
  - Id1
  - Id2
  - Id3
  - Id4
  - Latitude
  - Longitude
  - ProdName
  - ProductId
  - RegionId
  - RegName
  - ReorderPoint
  - State
  - WarehouseId
  - ZipCode
  - Operations
  - Named Criteria
- Operations

2. Drag the data collection onto a JSF page and, from the context menu, choose **Geographic Map** > **Map and Point, Pie Graph, Color, or Bar Graph Theme**.

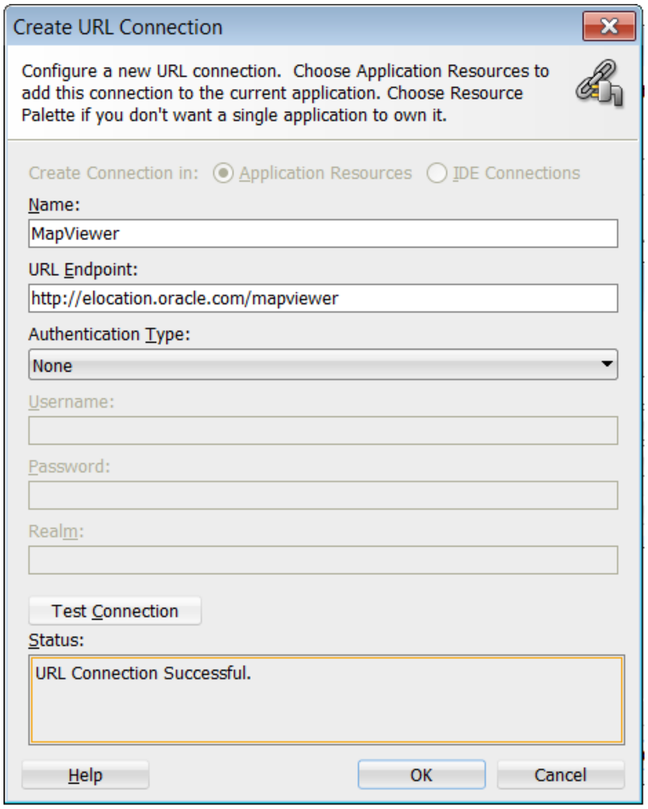
3. If you have not yet configured a map on the page, in the Create Geographic Map dialog, click the **New** icon to display the Create Geographic Map Configuration dialog and do the following:

   a. In the **Id** field enter the unique identifier for the map configuration. For example, `mapConfig1`.

   b. In the **MapViewer URL** field enter the URL for the Oracle MapViewer service.

      Use the dropdown list to select an existing connection, or click the Add icon to configure a new connection. In the Create URL Connection dialog use the address `http://elocation.oracle.com/mapviewer` for the URL endpoint and click **Test Connection** to confirm the connection. Figure 41-3 shows the completed Create URL dialog.

Figure 41-3 Create Map Viewer URL Connection



Create URL Connection

Configure a new URL connection. Choose Application Resources to add this connection to the current application. Choose Resource Palette if you don't want a single application to own it.

Create Connection in: ● Application Resources ○ IDE Connections

Name: MapViewer

URL Endpoint: http://elocation.oracle.com/mapviewer

Authentication Type: None

Username:

Password:

Realm:

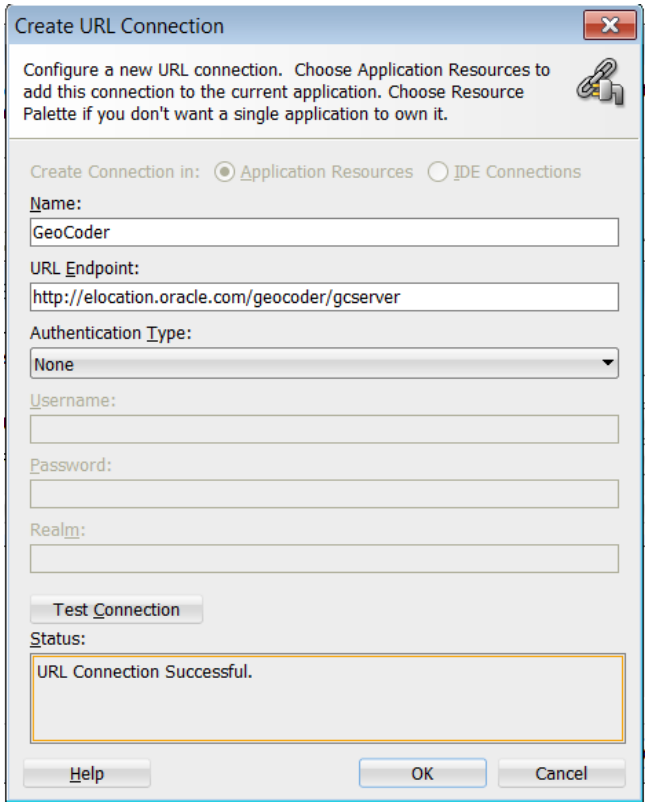
Status: URL Connection Successful.

c. In the **Geocoder URL** field enter the URL for the Geocoder web service that converts street addresses into latitude and longitude coordinates for mapping.

> **Note:**
>
> The Geocoder URL is needed only if you do *not* already have longitude and latitude information for addresses.

Use the dropdown list to select an existing connection, or click the Add icon to configure a new connection. In the Create URL Connection dialog use the address `http://elocation.oracle.com/geocoder/gcserver` for the URL endpoint and click **Test Connection** to confirm the connection. Figure 41-4 shows the completed Create URL dialog.

Figure 41-4 Create Geocoder URL Connection



Create URL Connection

Configure a new URL connection. Choose Application Resources to add this connection to the current application. Choose Resource Palette if you don't want a single application to own it.

Create Connection in: ● Application Resources ○ IDE Connections

Name: GeoCoder

URL Endpoint: http://elocation.oracle.com/geocoder/gcserver

Authentication Type: None

Username:

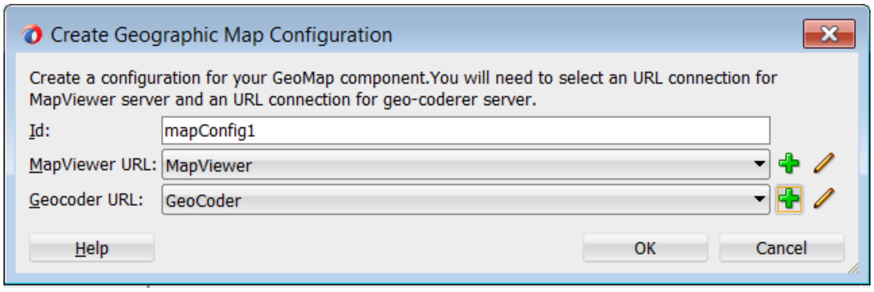
Password:

Realm:

Status: URL Connection Successful.

d. Click **OK**. Figure 41-5 shows the completed Create Geographic Map Configuration dialog.

Figure 41-5 Create Geographic Map Configuration Dialog



Create Geographic Map Configuration

Create a configuration for your GeoMap component.You will need to select an URL connection for MapViewer server and an URL connection for geo-coderer server.

Id: mapConfig1

MapViewer URL: MapViewer

Geocoder URL: GeoCoder

4. Click **OK**.

5. In the Create Geographic Map dialog, select the base map for the geographic map component and provide other settings to use with the map by doing the following:

   a. From the **Data Source** list select the collection of maps from which you will choose a base map.

   b. From the **Base Map** list select the map that will serve as the background for the geographic map component.

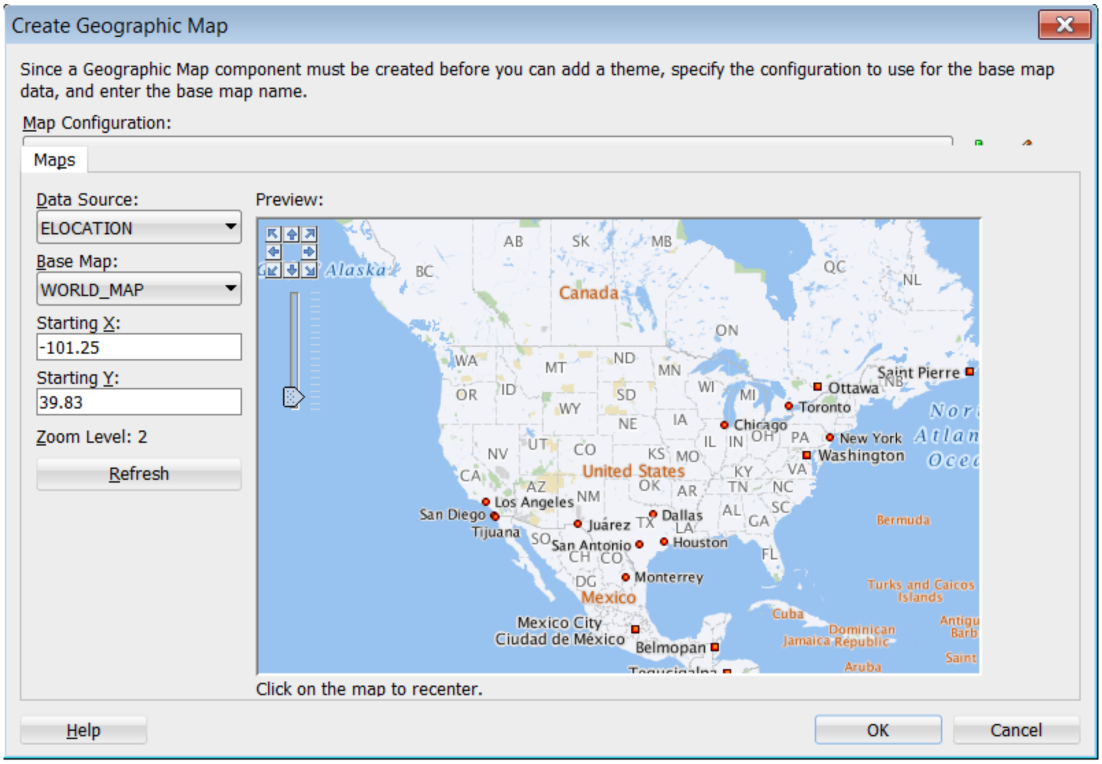
   c. To specify values for the **StartingX** field and the **StartingY** field click on the image of the map to center it within the Preview window.

You can use the arrows in the map navigator in the upper left-hand corner to move the map in the appropriate direction.

d. Optionally use the sliding arrow in the Preview window to adjust the zoom factor of the map.

e. Click **OK.** Figure 41-5 shows the completed Create Geographic Map dialog.

Figure 41-6 Create Geographic Map Dialog



Create Geographic Map

Since a Geographic Map component must be created before you can add a theme, specify the configuration to use for the base map data, and enter the base map name.

Map Configuration:

Maps

Data Source: ELOCATION

Base Map: WORLD_MAP

Starting X: -101.25

Starting Y: 39.83

Zoom Level: 2

Preview:

Click on the map to recenter.

6. Use the Create Theme dialog to data bind a map theme as follows:

   - Create a point theme to represent data on the base map. For more information, see How to Create a Geographic Map with a Point Theme.
   - Create a color theme to represent data on the base map. For more information, see How to Create a Geographic Map with a Color Theme.
   - Create a pie or bar graph theme to represent data on the base map. For more information, see How to Create a Geographic Map with a Pie or Bar Graph Theme.

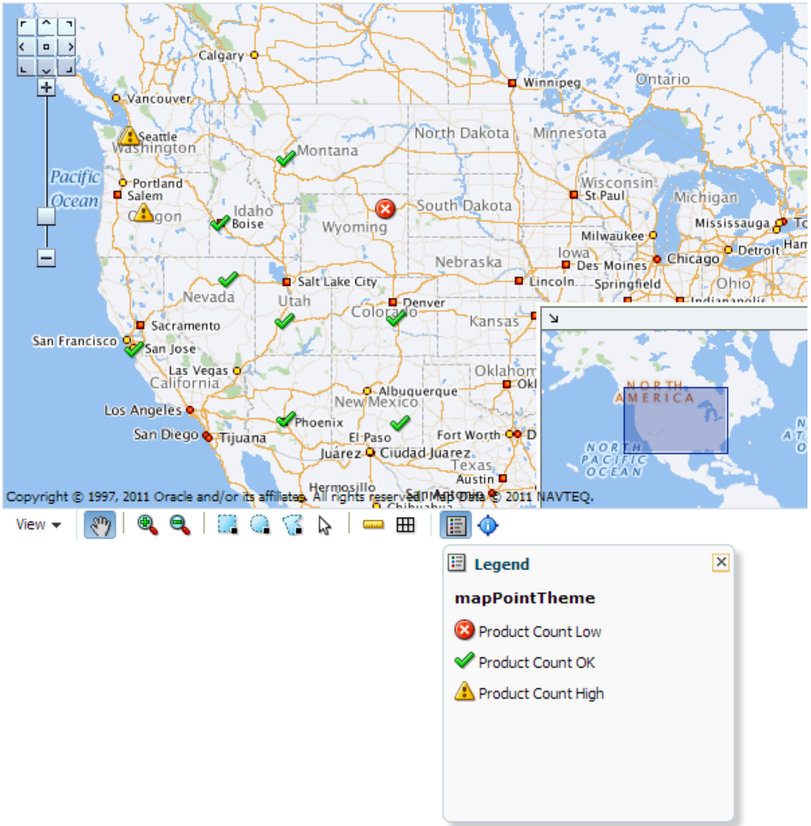
How to Create a Geographic Map with a Point Theme

You can create a geographic map with a point theme to display specific locations in a map, identified by latitude and longitude or address. For example, a point theme might identify the locations of warehouses in a map. If you customize the style of the point that is displayed, you could choose to use a different image for a warehouse product count (high, acceptable, low) in a set of warehouses to differentiate them from each other.

To create a geographic map with a point theme, you first configure the base map and then bind a point theme of the map to a data collection. JDeveloper allows you to do this declaratively by dragging and dropping a collection from the Data Controls panel. You can also layer a point theme on a base map that has already been configured and bound to another theme.

For example, the geographic map in Figure 41-7 has a point theme that identifies the location of warehouses and their product count levels in the United States.

Figure 41-7 Geographic Map with Point Theme



Before you begin:

It may be helpful to have an understanding of databound geographic maps. For more information, see Creating Databound Geographic Maps.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Data Visualization Components.

You will need to complete these tasks:

- Create an application module that contains instances of the view objects that you want in your data model for the geographic map with point theme, as described in Creating and Modifying an Application Module.

  For example, the data source for the geographic map point theme shown in Figure 41-7 comes from a view object created for the Summit ADF DVT sample application.

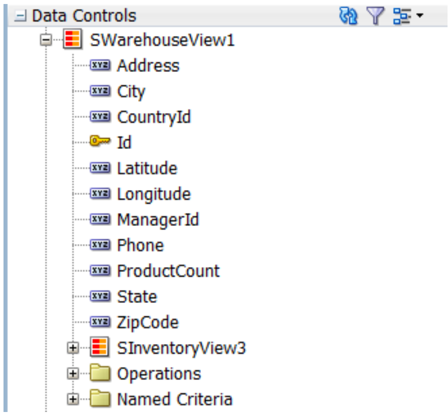
- Create a JSF page as described in the How to Create JSF Pages section of *Developing Web User Interfaces with Oracle ADF Faces*.

To create a geographic map with a databound point theme:

1. From the Data Controls panel, select a collection.

   Figure 41-8 shows an example where you could select the `SWarehouseView1` collection in the Data Controls panel to create a geographic map with a point theme that displays an image to identify the location of each warehouse, styled to represent the product count level.

   Figure 41-8 Data Collection for Warehouse Location and Product Count



Data Controls
SWarehouseView1
Address
City
CountryId
Id
Latitude
Longitude
ManagerId
Phone
ProductCount
State
ZipCode
SInventoryView3
Operations
Named Criteria

2. Drag the collection onto a JSF page and, from the context menu, choose **Geographic Map** > **Map and Point Theme**.
3. If you have not yet configured a map on the page, complete the Create Geographic Map dialog. For more information, see How to Configure a Geographic Base Map. Otherwise, do one of the following:
   - Use the dropdown list to select the map configuration you wish to use
   - Click **OK** to accept the available map configuration
   - Click **New** to configure a new base map
   - Click **Edit** to modify the available map configuration
4. If you are layering a point theme on a base map that has already been configured and bound to another theme, in the Create dialog select Point Theme.
5. In the Create Point Map Theme dialog, in the **Theme Id** field, enter the unique identifier for the point map theme.
6. In the **Location** section, choose whether the point location is to be specified by **Address** or by a pair of *x* and *y* coordinates (Longitude and Latitude).

   The choice you select for location will determine which controls appear in the Location section.

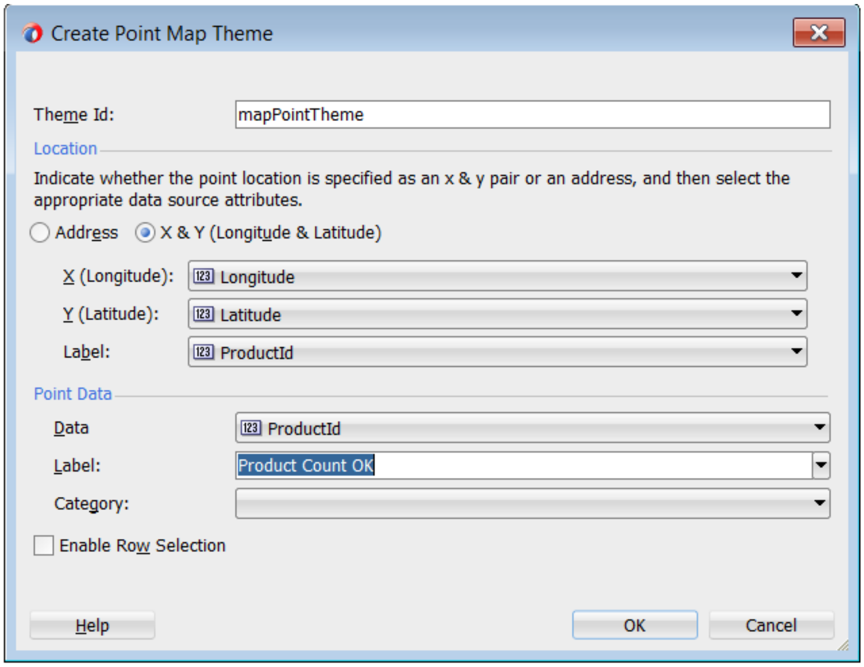
> **Tip:**
>
> Using *x* and *y* coordinates is a more efficient way to present data on the map rather than using the Address controls, which must be converted by a Geocoder to *x* and *y* coordinates. If the data collection has more than 100 rows, use *x* and *y* coordinates for better performance.

7. For the *x* and *y* point location, select attributes from the data collection that represents the following items:
   - X (Longitude): The horizontal location of the point on the map.
   - Y (Latitude): The vertical location of the point on the map.
   - Label: The labels for the points in the top section of the information window, which is displayed when you click a point.
8. In the Point Data section, provide the following information that identifies the data associated with the point, its label, and optionally the style for the point:
   - In the **Data** field, select the data column that is associated with the point, such as `ProductCount`.
   - In the **Label** field, enter the text that will appear in the information window before the data value when you click a point. You can enter a text resource to use for the label. The text resource can be a translatable string from a resource bundle or an EL expression executed at runtime. Use the dropdown list to open a Select Text Resource or Expression Builder dialog. If you need help, press F1 or click **Help**.
   - Optionally, in the **Category** field, select a data column to use for finding the appropriate style for a point. If you select a value for Category, that value is stored in the binding for this point theme and then matched against the `itemValue` attribute of the `mapPointStyleItem` tags that you create for this point theme.

     > **Note:**
     >
     > If your data does not have a column that you want to use as a category for finding the style of a point, you can also use `mapPointStyleItem` tags to define styles related to data ranges (such as high, acceptable, and low) that are matched to the values in the column that you select in the Data field. For more information, see How to Create Point Style Items for a Point Theme.
9. Select the **Enable Row Selection** Select if you want to enable master-detail relationships. This is useful when the data collection for the geographic map is a master in a master-detail relationship with a detail view that is displayed in another UI component on the page. Selecting this options enables both `selectionListenter` and `clickListener` attributes.
10. Click **OK**.

Figure 41-9 shows the completed Create Point Map Theme dialog for a geographic map with a point theme that displays an image representing the location and product count levels for each warehouse point.

Figure 41-9 Create Point Map Theme Dialog for Warehouse Product Counts



Create Point Map Theme

Theme Id: mapPointTheme

Location

Indicate whether the point location is specified as an x & y pair or an address, and then select the appropriate data source attributes.

○ Address ● X & Y (Longitude & Latitude)

X (Longitude): Longitude

Y (Latitude): Latitude

Label: ProductId

Point Data

Data: ProductId

Label: Product Count OK

Category:

□ Enable Row Selection

After you create a map point theme, you can further customize the style of the points that appear in the map to represent more detailed information. For example, the map point theme in Figure 41-7 uses a different image to represent the levels (high, acceptable, low) for the product count in each warehouse. For each different point style, use a `mapPointStyleItem` tag. For more information, see How to Create Point Style Items for a Point Theme.

How to Create Point Style Items for a Point Theme

There are a variety of options available for creating point style items for use in a given map point theme. These are:

- A single image for all data points
- Separate images for each data point category
- Images that represent low, medium, and high data value ranges

After you create the data binding for a map point theme, you have the option of selecting a single built-in image that should be used for all points in that map theme. In the Properties window, you can make this selection in the `builtInImage` attribute of the `mapPointTheme` tag. The default value for this attribute is an orange ball.

Alternatively, if you specify a value for Category in the Create Point Map Theme dialog, then you should also create a set of point style items to determine a separate image that represents data points in each category. In this case, you do not use the minimum and maximum values in the point style item tags. Instead, you set the `itemValue` attribute of point style item tags to a value that matches entries in the data column that you specified for Category.

In a point theme for a geographic map, if you do not specify a value for Category, you can still use the `mapPointStyleItem` child tags of the `mapPointTheme` tag to specify ranges of values (such as low, medium, and high) and the images that are to represent these ranges. If you do this, then each point will be represented by an image that identifies the range in which the data value for that point falls.

Before you begin:

It may be helpful to have an understanding of databound geographic maps. For more information, see Creating Databound Geographic Maps.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Data Visualization Components.

You will need to complete these tasks:

- Create an application module that contains instances of the view objects that you want in your data model for the geographic map with point theme, as described in Creating and Modifying an Application Module.

  For example, the data source for the geographic map point theme shown in Figure 41-7 comes from a view object created for the Summit ADF DVT sample application.
- Create a JSF page as described in the How to Create JSF Pages section of *Developing Web User Interfaces with Oracle ADF Faces*.
- Add a geographic map with a point theme to your page. For more information, see How to Create a Geographic Map with a Point Theme.

To add point style items to a map point theme to represent data value ranges:

1. In the Structure window, right-click the **dvt:mapPointTheme** tag and choose **Insert inside the Point Theme** > **Point Style Item**.
2. In the Point Style Item Properties window, set values as described Table 41-1.

Table 41-1 Properties for Point Style Item

For this property	Set this value
`Id`	Specify a unique ID for the point style item.
`MinValue`	Specify the minimum value in a data range that you define.
`MaxValue`	Specify the maximum value in a data range that you define.
`ShortLabel`	Specify text to appear when a user hovers over the point item. For example, if you define a point item for low inventory, then enter `Low Inventory` as the value for this property.
`ImageURL`	Specify the URL to the image file or select it from the dropdown list. At runtime, the image you specify appears on the map to represent the data range identified by the **MinValue** and **MaxValue** properties. Alternatively, you can select one of a number of predefined images referenced by the **BuiltInImage** dropdown list that appears in the **Other** section.
`HoverImageURL`	Specify the URL to the image file or select it from the dropdown list. At runtime, the image you specify appears when a user hovers over the point item.

Table 41-1 (Cont.) Properties for Point Style Item

For this property	Set this value
`SelectedImageURL`	Specify the URL to the image file or select it from the dropdown list. At runtime, the image you specify appears when a user selects the point item.

3. If you defined a data value range for a low data value range in Steps 1 and 2, then repeat Steps 1 and 2 to define medium and high data value ranges with appropriate values.

> **Note:**
>
> The use of `mapPointStyleItem` child tags to customize the style of points is a declarative approach that lets you provide custom point images. For information about using a callback to provide not only custom images but also custom HTML, see What You May Need to Know About Adding Custom Point Style Items to a Map Point Theme.

What Happens When You Create a Geographic Map with a Point Theme

Creating a geographic map with a point theme from the Data Controls panel has the following effect:

- Creates the bindings for the point theme and adds the bindings to the page definition file
- Adds the necessary tags to the JSF page for the geographic map component
- Adds the necessary point theme child tags within the geographic map tag to the JSF page

The example below shows the row set bindings that were generated for the geographic map with a point theme.

```
<bindings>
    <mapTheme xmlns="http://xmlns.oracle.com/adfm/dvt"></mapTheme>
    <mapTheme IterBinding="WorldProductInventory1Iterator"
              id="WorldProductInventory1"
              xmlns="http://xmlns.oracle.com/adfm/dvt">
      <mapThemeDataMap convert="false" mapThemeType="point">
        <data>
          <item value="AmountInStock"
                label="${adfBundle['view.ViewControllerBundle'].AMOUNT_IN_
                       STOCK}"/>
        </data>
        <item type="us_form_2" street="Address" city="City" state="State"
                   zipCode="ZipCode" label="WarehouseId"/>
      </mapThemeDataMap>
    </mapTheme>
    <mapTheme IterBinding="SWarehouseView1Iterator" id="SWarehouseView1"
              xmlns="http://xmlns.oracle.com/adfm/dvt">
      <mapThemeDataMap convert="false" mapThemeType="point">
        <data>
```

```
          <item value="ProductCount"
                label="${adfBundle['view.ViewControllerBundle'].PRODUCT_COUNT}"/>
        </data>
        <item type="lat_long" longitude="Longitude" latitude="Latitude"
              label="ProductCount"/>
      </mapThemeDataMap>
    </mapTheme>
  </bindings>
```

The example below shows the XML code generated on the JSF page for the geographic map and its point theme.

```
<dvt:map id="map" startingX="-104.15" mapServerConfigId="mapConfig1"
            baseMapName="ELOCATION.WORLD_MAP" mapZoom="3"
         inlineStyle="width:600px; height:375px;" startingY="42.09">
  <dvt:mapPointTheme id="mapPointTheme"
                     value="#{bindings.SWarehouseView1.geoMapModel}">
    <dvt:mapPointStyleItem maxValue="5.0"
                           shortLabel="#{viewcontrollerBundle.PRODUCT_COUNT_LOW}"
                           imageURL="/images/error.png" minValue="0.0"/>
    <dvt:mapPointStyleItem minValue="6.0" maxValue="7.0"
                           shortLabel="#{viewcontrollerBundle.PRODUCT_COUNT_OK}"
                           imageURL="/images/checkmark.png"/>
    <dvt:mapPointStyleItem minValue="8.0"

shortLabel="#{viewcontrollerBundle.PRODUCT_COUNT_HIGH}"
                           imageURL="/images/warning.png"/>
  </dvt:mapPointTheme>
</dvt:map>
<dvt:mapToolbar mapId="map" id="mt1"/>
```

What You May Need to Know About Adding Custom Point Style Items to a Map Point Theme

If you want to provide custom HTML as well as custom images for map points, then you can use the `customPointCallback` attribute of the `mapPointTheme` tag to accomplish this customization.

> **Note:**
>
> If you set the `customPointCallback` attribute for a map point theme, the map ignores any `dvt:mapPointStyleItem` child tags because the callback overrides these tags.

To use a callback to customize the style of map points:

1. Write a method in Java to perform the desired point customization.
2. Store this method in a managed bean for the map.

   For more information about managed beans, see the Creating and Using Managed Beans section in *Developing Web User Interfaces with Oracle ADF Faces*.

3. After you finish data-binding the map point theme, use the Properties window to specify a reference to the managed bean method in the `customPointCallback` attribute of the `dvt:mapPointTheme` tag.

   For example, if the managed bean is named `MapSampleBean` and the method is named `setCustomPointStyle`, then the reference becomes `#{mapSampleBean.CustomPointStyle}`.

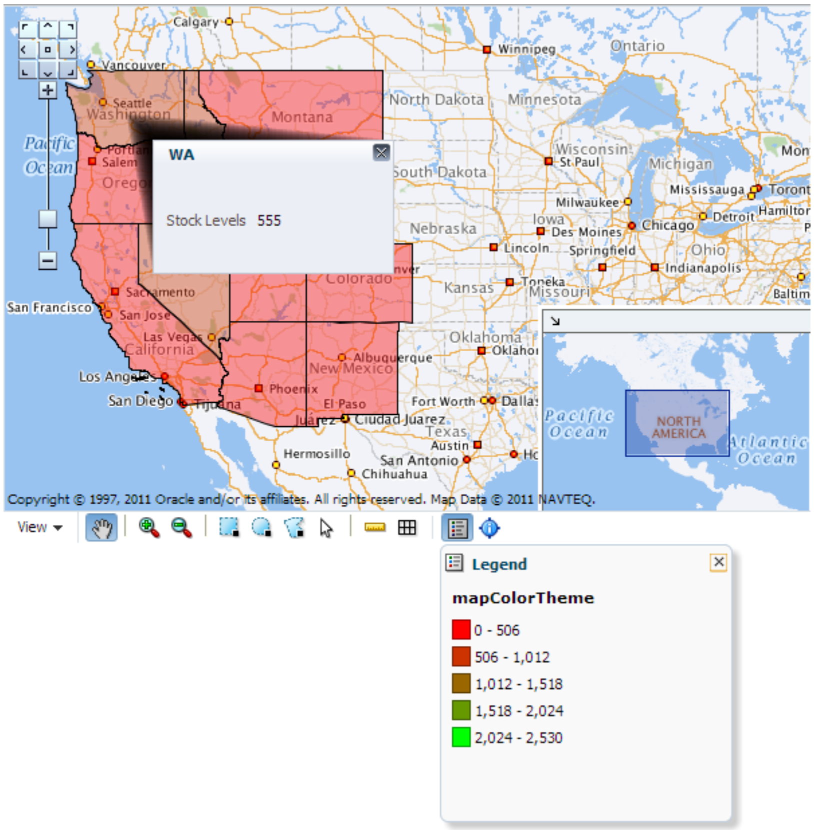
How to Create a Geographic Map with a Color Theme

You can create a geographic map with a color theme to display areas of the map in colors. For example, a color theme might identify the states with warehouses, styled with a range of color to represent stock levels in each state.

To create a geographic map with a color theme, you first configure the base map and then bind a color theme of the map to a data collection. JDeveloper allows you to do this declaratively by dragging and dropping a collection from the Data Controls panel. You can also layer a color theme on a base map that has already been configured and bound to another theme.

For example, the geographic map in Figure 41-10 has a color theme that identifies the states with warehouses styled to represent stock levels for each states.

Figure 41-10 Geographic Map with Color Theme



WA

Stock Levels 555

Copyright © 1997, 2011 Oracle and/or its affiliates. All rights reserved. Map Data © 2011 NAVTEQ.

Legend

mapColorTheme

🟥 0 - 506
🟥 506 - 1,012
🟫 1,012 - 1,518
🟩 1,518 - 2,024
🟩 2,024 - 2,530

Before you begin:

It may be helpful to have an understanding of databound geographic maps. For more information, see Creating Databound Geographic Maps.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Data Visualization Components.

You will need to complete these tasks:

- Create an application module that contains instances of the view objects that you want in your data model for the geographic map with color theme, as described in Creating and Modifying an Application Module.

  For example, the data source for the geographic map with color theme shown in Figure 41-10 comes from a view object created for the Summit ADF DVT sample application.

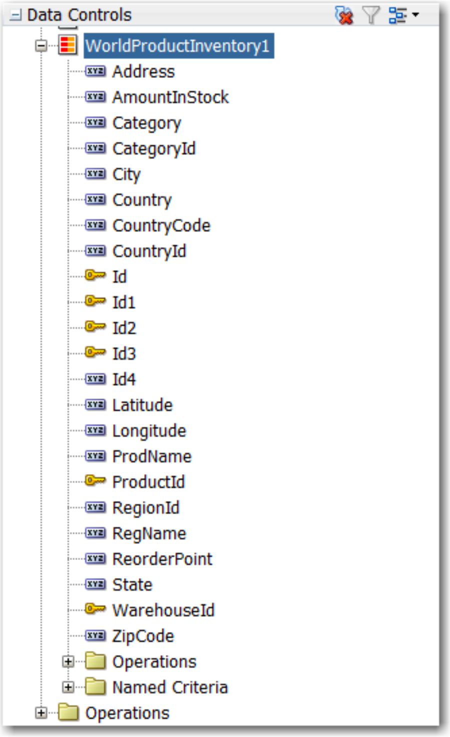
- Create a JSF page as described in the How to Create JSF Pages section of *Developing Web User Interfaces with Oracle ADF Faces*.

To create a geographic map with a databound color theme:

1. From the Data Controls panel, select a collection.

   Figure 41-11 shows an example where you could select the `WorldProductInventory1` collection in the Data Controls panel to create a color map theme that shows product inventory levels by the color of regions, in this example, US States.

Figure 41-11 Data Collection for Product Inventory Levels



Data Controls

WorldProductInventory1
Address
AmountInStock
Category
CategoryId
City
Country
CountryCode
CountryId
Id
Id1
Id2
Id3
Id4
Latitude
Longitude
ProdName
ProductId
RegionId
RegName
ReorderPoint
State
WarehouseId
ZipCode
Operations
Named Criteria
Operations

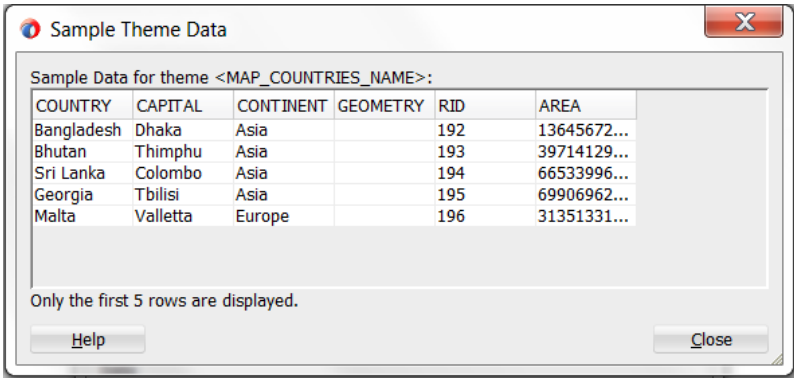
2. Drag the collection onto a JSF page which already contains a geographic map component and, from the context menu, choose **Geographic Map** > **Map and Color Theme**.

3. If you have not yet configured a map on the page, complete the Create Geographic Map dialog. For more information, see How to Configure a Geographic Base Map. Otherwise, do one of the following:
   - Use the dropdown list to select the map configuration you wish to use
   - Click **OK** to accept the available map configuration
   - Click **New** to configure a new base map
   - Click **Edit** to modify the available map configuration
4. If you are layering a color theme on a base map that has already been configured and bound to another theme, in the Create dialog select Color Theme.
5. In the Create Color Map Theme dialog, in the **Theme Id** field, enter a unique identifier for the map theme in the **Id** field.
6. In the **Base Map Theme** section, identify the base map color theme to use for the geographic map by doing the following:
   a. In the **Name** field, select the name of the base map theme.
   b. For **Location**, select the location column in the data collection that should be matched to the location column in the base map theme that you selected.
   c. Optionally, click **View Sample Theme Data** to display the Sample Theme Data dialog, in which you can examine the first several rows of the actual data so that you can identify the appropriate location column.

      For example, if you want to view the data for a region that consists of states in the United States map, you might select MAP_STATES_NAME as shown in Figure 41-12.

> **Note:**
>
> It is possible for an administrator of Oracle Spatial to disable the display of sample data. If this button is not available, then consult the administrator for guidance.

Figure 41-12 Sample Theme Data for Regions or States



Sample Data for theme <MAP_COUNTRIES_NAME>:

COUNTRY	CAPITAL	CONTINENT	GEOMETRY	RID	AREA
Bangladesh	Dhaka	Asia		192	13645672...
Bhutan	Thimphu	Asia		193	39714129...
Sri Lanka	Colombo	Asia		194	66533996...
Georgia	Tbilisi	Asia		195	69906962...
Malta	Valletta	Europe		196	31351331...

Only the first 5 rows are displayed.

7. In the **Appearance** section, specify the look of the color theme as follows:
   a. In **Data Bucket Count**, enter the number of groups for the data in this geographic map. Each group is coded with a color. After specifying this

number, you can provide colors for the minimum value and the maximum value. The colors for the other values are chosen automatically using an RGB algorithm.

b. In **Minimum Value Color**, select the color for the minimum value.

c. In **Maximum Value Color**, select the color for the maximum value.

> **Note:**
>
> If you want to specify an exact color for each data bucket, see What You May Need to Know About Customizing Colors in a Map Color Theme.

8. In the **Data** section, provide the following information about the data in the collection:

   a. For **Location**, select the column in the data collection that should match the values in the location column that you selected from the base map theme.

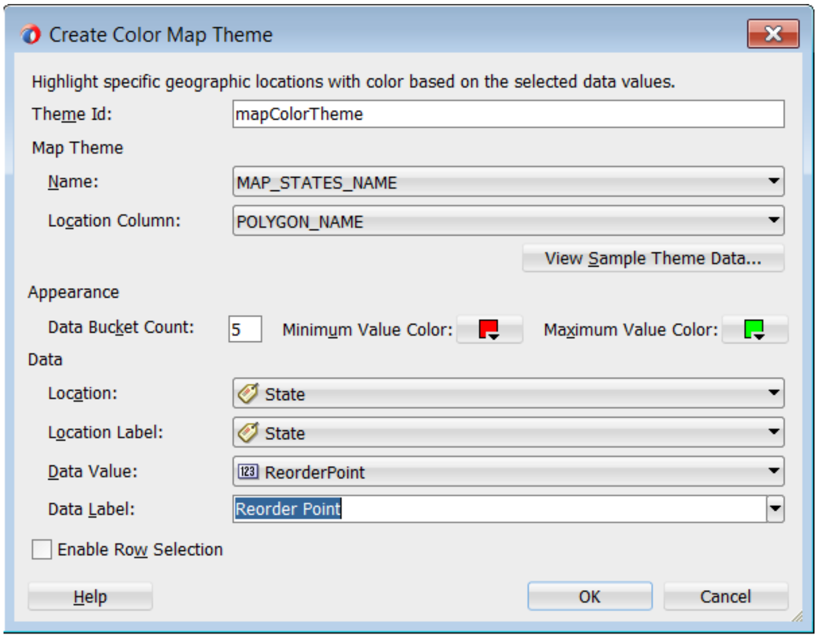
   b. For **Location Label**, select the column in the data collection that contains the labels associated with the values in the location column. These labels are shown in the information window that is displayed when you click or hover over a color.

   c. For **Data Label**, enter the label to use for describing the data in the information window and the tooltip that is displayed when you click or hover over a color. For example, the information window might include a label before the data value, such as **Product Popularity**. You can also enter a text resource to use for the data label. The text resource can be a translatable string from a resource bundle or an EL expression executed at runtime for a dynamic label. Use the dropdown list to open a Select Text Resource or Expression Builder dialog. If you need help, press F1 or click **Help**.

9. Use **Enable Row Selection** only if you want to enable master-detail relationships. This is useful when the data collection for the map theme is a master in a master-detail relationship with a detail view that is displayed in another UI component on the page.

Figure 41-13 shows the Create Color Map Theme dialog for the product popularity by state color theme.

Figure 41-13 Create Color Map Theme for Product Popularity By State



Create Color Map Theme

Highlight specific geographic locations with color based on the selected data values.

Theme Id: mapColorTheme

Map Theme

Name: MAP_STATES_NAME

Location Column: POLYGON_NAME

Appearance

Data Bucket Count: 5 Minimum Value Color: 🟥 Maximum Value Color: 🟩

Data

Location: State

Location Label: State

Data Value: ReorderPoint

Data Label: Reorder Point

☐ Enable Row Selection

What Happens When You Add a Color Theme to a Geographic Map

Dropping a color theme from the Data Controls panel to an existing geographic map has the following effect:

- Creates the bindings for the color theme and adds the bindings to the page definition file
- Adds the necessary color theme child tags within the geographic map tag to the JSF page

The example below shows the row set bindings that were generated for the color theme of the geographic map.

```
<bindings>
    <mapTheme IterBinding="WorldProductInventory1Iterator"
              id="WorldProductInventory1"
              xmlns="http://xmlns.oracle.com/adfm/dvt">
      <mapThemeDataMap convert="false" mapThemeType="color">
        <item type="location" value="State" label="State"/>
        <data>
          <item value="AmountInStock"
                label="${adfBundle['view.ViewControllerBundle'].STOCK_LEVELS}"/>
        </data>
      </mapThemeDataMap>
    </mapTheme>
  </bindings>
```

The example below shows the XML code generated on the JSF page for a color theme that represents product popularity in different states on the United States map.

```
<dvt:map id="map" startingX="-102.39" mapServerConfigId="mapConfig1"
         baseMapName="ELOCATION_MERCATOR.WORLD_MAP"
         mapZoom="3" inlineStyle="width:600px; height:375px;" startingY="40.27"
```

```
           summary="Geographic map with color theme">
    <dvt:mapColorTheme id="mapColorTheme" themeName="MAP_STATES_ABBRV"
                       value="#{bindings.WorldProductInventory1.geoMapModel}"
                       bucketCount="5" minColor="#ff0000"
                       maxColor="#00ff00" locationColumn="STATE_ABRV"/>
        </dvt:map>
```

What You May Need to Know About Customizing Colors in a Map Color Theme

While you are data-binding a map color theme, you can specify only a minimum color and a maximum color for the data buckets. The map uses an algorithm to determine the colors of the buckets between the minimum and maximum. However, after the data-binding is finished, you have the option of specifying the exact color to be used for each data bucket.

In the Properties window, for the `dvt:mapColorTheme` tag you can use the `colorList` attribute to specify the color for each bucket. You can either bind a color array to this attribute or you can specify a string of colors using a semicolon separator.

For example, if the value of this attributes is set to: `#ff0000;#00ff00;#0000ff`, then the color of the first bucket is red, the second bucket is green, and the third bucket is blue.

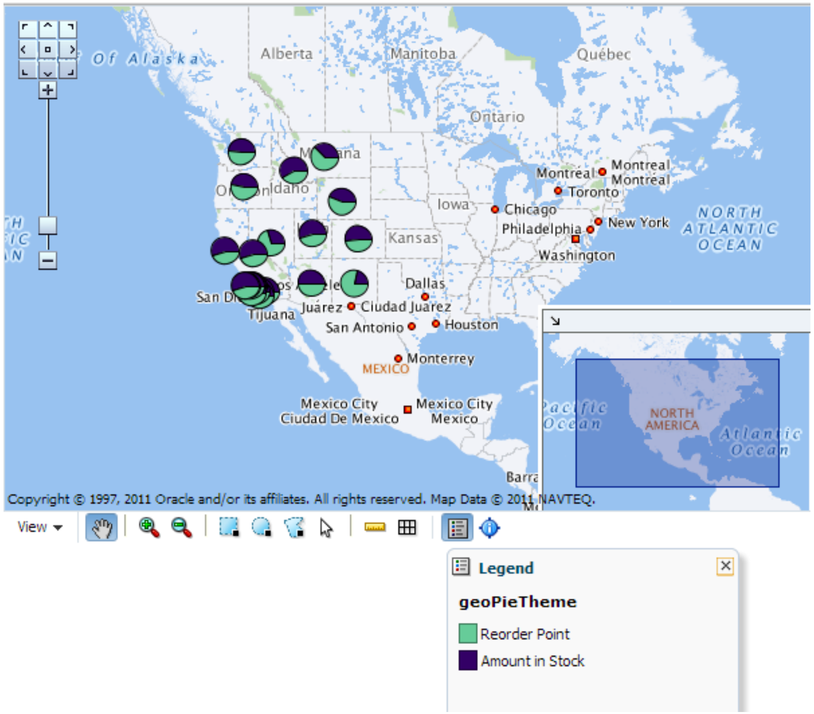
How to Create a Geographic Map with a Pie or Bar Graph Theme

You can create a geographic map with a pie or bar graph theme to display specific locations in a map, identified by latitude and longitude or address. For example, a point theme might identify the locations of warehouses in a map. If you customize the style of the point that is displayed, you could choose to use a different image for the warehouse product count (high, acceptable, low) in a set of warehouses to differentiate them from each other.

To create a geographic map with a pie or bar graph theme, you first configure the base map and then bind a pie or bar graph theme of the map to a data collection. JDeveloper allows you to do this declaratively by dragging and dropping a collection from the Data Controls panel. You can also layer a pie or bar graph theme on a base map that has already been configured and bound to another theme.

For example, the geographic map in Figure 41-14 has a point theme that identifies the location of warehouses and their product count levels in the United States.

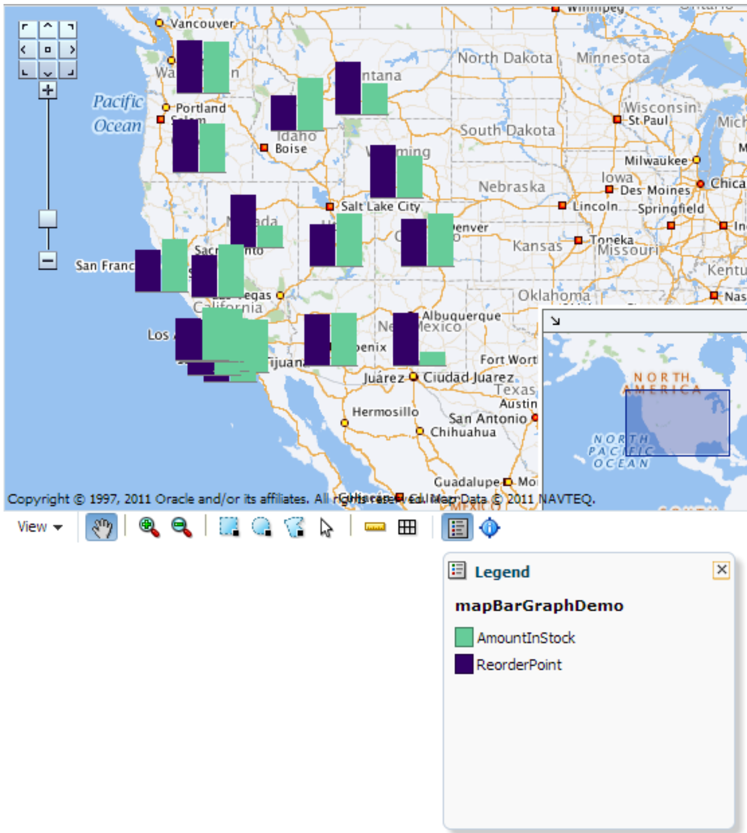
Figure 41-14   Geographic Map with Pie Graph Theme



Copyright © 1997, 2011 Oracle and/or its affiliates. All rights reserved. Map Data © 2011 NAVTEQ.

For example, the geographic map in Figure 41-14 has a point theme that identifies the location of warehouses and their product count levels in the United States.

Figure 41-15 Geographic Map with Bar Graph Theme



Before you begin:

It may be helpful to have an understanding of databound geographic maps. For more information, see Creating Databound Geographic Maps.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Data Visualization Components.

You will need to complete these tasks:

- Create an application module that contains instances of the view objects that you want in your data model for the geographic map with a pie or bar graph theme, as described in Creating and Modifying an Application Module.

  For example, the data source for the pie graph theme shown in Figure 41-14 and the bar graph theme shown in Figure 41-15 comes from a view object created for the Summit ADF DVT sample application.

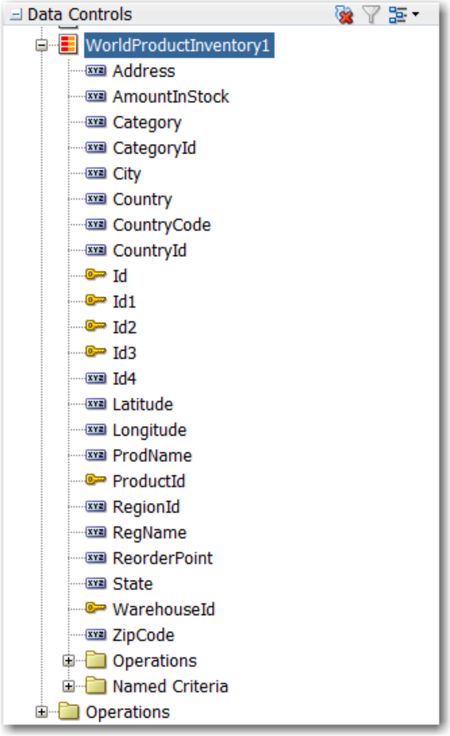
- Create a JSF page as described in the How to Create JSF Pages section of *Developing Web User Interfaces with Oracle ADF Faces*.

To create a geographic map with databound pie or bar graph theme:

1. From the Data Controls panel, select a collection.

Figure 41-16 shows an example where you could select the `WorldProductInventory1` collection to create a pie or bar graph bar theme in an existing geographic map component to represent the popular product categories within a state.

Figure 41-16   Data Collection for Inventory Levels by Product



Data Controls
- WorldProductInventory1
  - Address
  - AmountInStock
  - Category
  - CategoryId
  - City
  - Country
  - CountryCode
  - CountryId
  - Id
  - Id1
  - Id2
  - Id3
  - Id4
  - Latitude
  - Longitude
  - ProdName
  - ProductId
  - RegionId
  - RegName
  - ReorderPoint
  - State
  - WarehouseId
  - ZipCode
  - Operations
  - Named Criteria
- Operations

2. Drag the collection onto a JSF page and, from the context menu, choose **Geographic Map** > **Map and Pie (or Bar) Graph Theme**.

3. If you have not yet configured a map on the page, complete the Create Geographic Map dialog. For more information, see How to Configure a Geographic Base Map. Otherwise, do one of the following:

   - Use the dropdown list to select the map configuration you wish to use
   - Click **OK** to accept the available map configuration
   - Click **New** to configure a new base map
   - Click **Edit** to modify the available map configuration

4. If you are layering a pie graph theme on a base map that has already been configured and bound to another theme, in the Create dialog select Pie Graph Theme.

5. In the Create Pie Graph Theme dialog, do the following to identify the new theme and the base map theme elements that you want to work with:

   a. For **Theme Id**, enter a unique identifier for the pie graph theme that you are creating.

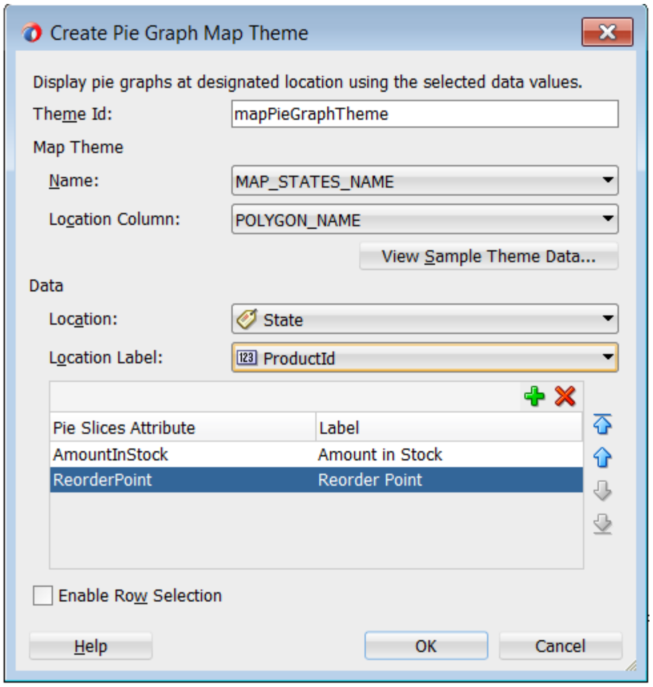
   b. In the **Base Map Theme** section, select the name of the base map and the region in which you want to place the pie graphs.

6. In the **Appearance** section, under **Data**, do the following:
   a. For **Location**, select the location column in the data collection that should be matched to the location column in the base map theme that you selected.

      If needed, click **View Sample Theme Data** to examine the first several rows of the actual data so that you can identify the appropriate location column.
   b. For **Location Label**, select the column in the data collection that contains labels for the locations in the data collection.
   c. In the grid for the **Pie Slices Attributes** (for pie graph) or **Series Attributes** (for bar graph), enter each attribute that contains values that you want represented in the pie or bar graph that you are creating.
   d. Beside each pie slice or series attribute, enter text that should be used as a label for the data value in the attribute. You can enter the text directly, select `No Label` to suppress the label as in the case of using a single data value, specify a text resource from a resource bundle, or use the EL Expression builder to evaluate the label text at runtime.
7. Select **Enable Row Selection** only if you want to enable the selection of rows in a related component. You select this component when the page contains a component that is linked to a data collection that is related to the geographic map that you are creating.
8. Click **OK**.

Figure 41-17 shows the completed Create Pie Graph Map Theme dialog for the inventory levels by product.

Figure 41-17 Create Pie Graph Map Theme for Inventory Levels by Product



Create Pie Graph Map Theme

Display pie graphs at designated location using the selected data values.

Theme Id: mapPieGraphTheme

Map Theme

Name: MAP_STATES_NAME

Location Column: POLYGON_NAME

View Sample Theme Data...

Data

Location: State

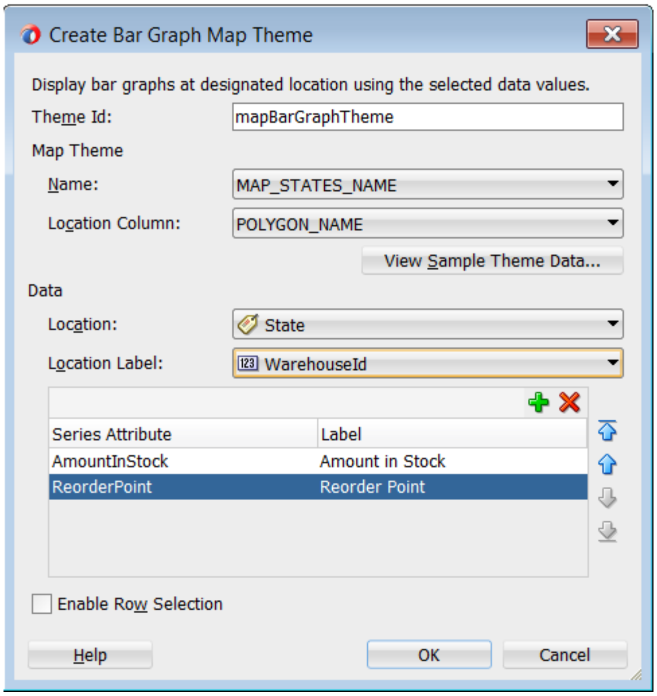
Location Label: ProductId

Pie Slices Attribute	Label
AmountInStock	Amount in Stock
ReorderPoint	Reorder Point

□ Enable Row Selection

Figure 41-17 shows the completed Create Bar Graph Map Theme dialog for the inventory levels by product.

Figure 41-18 Create Bar Graph Map Theme for Inventory Levels by Product



Create Bar Graph Map Theme

Display bar graphs at designated location using the selected data values.

Theme Id: mapBarGraphTheme

Map Theme

Name: MAP_STATES_NAME

Location Column: POLYGON_NAME

Data

Location: State

Location Label: WarehouseId

Series Attribute	Label
AmountInStock	Amount in Stock
ReorderPoint	Reorder Point

□ Enable Row Selection

What Happens When You Add a Pie or Bar Graph Theme to a Geographic Map

Dropping a pie graph theme from the Data Controls panel to an existing geographic map has the following effect:

- Creates the bindings for the pie graph theme and adds the bindings to the page definition file
- Adds the necessary pie graph theme code to the JSF page within the map XML

The example below shows the row set bindings that were generated for the pie graph theme of the geographic map.

```
<mapTheme IterBinding="PopularCategoriesByState1Iterator"
        id="PopularCategoriesByState1"
        xmlns="http://xmlns.oracle.com/adfm/dvt">
  <mapThemeDataMap mapThemeType="pieChart">
    <item type="location" value="StateProvince" label="StateProvince"/>
    <data>
      <item value="AudioVideo"
            label="${adfBundle['view.ViewControllerBundle'].AUDIO_VIDEO}"/>
      <item value="CellPhones"
            label="${adfBundle['view.ViewControllerBundle'].CELL_PHONES}"/>
      <item value="Games"
            label="${adfBundle['view.ViewControllerBundle'].GAMES}"/>
    </data>
  </mapThemeDataMap>
</mapTheme>
```

The example below shows the XML code generated on the JSF page for the pie graph theme of the geographic map.

```
<dvt:mapPieGraphTheme id="mapPieGraphTheme1"
     themeName="MAP_STATES_NAME"
     shortLabel="#{viewcontrollerBundle.POPULAR_CATEGORIES}"
     pieRadius="10"
     styleName="comet"
     value="#{bindings.PopularCategoriesByState1.geoMapModel}"
     locationColumn="POLYGON_ID"/>
</dvt:mapPieGraphTheme>
```

Creating Databound Thematic Maps

A thematic map represents business data as patterns in stylized areas or associated markers and does not require a connection to a remote Oracle MapViewer service. Thematic maps focus on data without the geographic details in a geographic map.

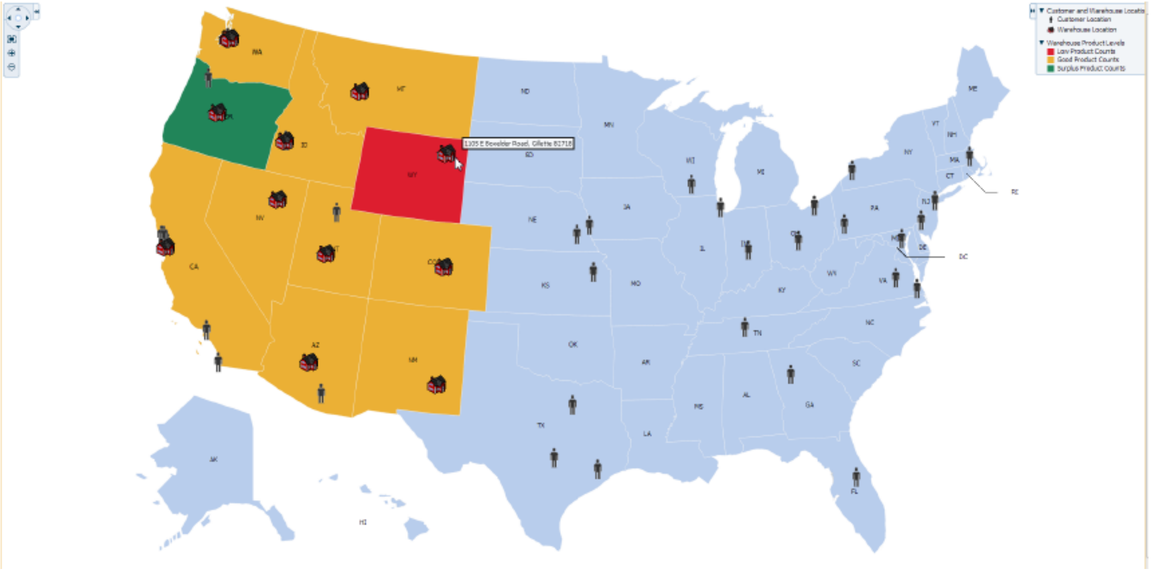
The thematic map is packaged with prebuilt base maps including a USA base map, a world base map, and base maps for continents and regions of the world including EMEA and APAC. Each base map includes several sets of regions and one fixed set of cities. A set of regions or cities is referred to as a layer. Each layer can be bound to a data collection and stylized to represent the data with color and pattern fills, or a data marker, or both. At runtime, only one map layer and its associated data can be displayed at a time, unless the thematic map has been enabled for drilling.

The data displayed in a thematic map is based on data collections. Using ADF data controls, JDeveloper makes data binding a declarative task. You drag and drop a collection from the Data Controls panel onto the JSF page and use a Component Gallery to select the base map and map layers on which to display the data. You can then use a Layer Browser and binding dialogs to bind data collection attributes to the data layers in the thematic map.

Stamping is used to associate map layers with a row of data in a data collection. Using stamping, each row of data in the data model can be identified by a style, for example a color or pattern; a marker, for example a circle or square; or an image. When you use stamping, child components are not created for every area, marker, or image in a thematic map. Rather, the content of the component is repeatedly rendered, or stamped, once per data attribute, such as the rows in a data collection.

Figure 41-19 shows a thematic map using a USA base map with a states map layer to display customer and warehouse locations, and the product inventory levels for states with warehouses in the Summit ADF DVT sample application. The example illustrates thematic map default features including a data bound legend and labels associated with the styled points and areas when you use the Data Controls panel and thematic map binding dialogs.

Figure 41-19   Thematic Map Displaying Customer and Warehouse Locations



For detailed information about thematic map end user and presentation features, use cases, tag structure, and adding special features to thematic maps, see the Using Thematic Map Components section in *Developing Web User Interfaces with Oracle ADF Faces*.

How to Create a Thematic Map Using ADF Data Controls

The `thematicMap` component uses a model to access the data in the underlying list. The specific model class is `oracle.adf.view.rich.model.CollectionModel`. You can also use other model instances, for example, `java.util.List`, `java.util.Array`, and `javax.faces.model.DataModel`. The data layer will automatically convert the instance into a `CollectionModel`.

Before you begin:

It may be helpful to have an understanding of databound thematic maps. For more information, see Creating Databound Thematic Maps.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Data Visualization Components.

You will need to complete these tasks:

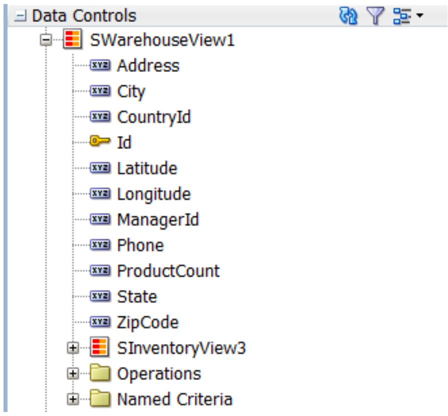
- Create an application module that contains instances of the view objects that you want in your data model, as described in Creating and Modifying an Application Module.
- Create a JSF page as described in the How to Create JSF Pages section of *Developing Web User Interfaces with Oracle ADF Faces*.

To create a thematic map using the Data Controls panel:

1. From the Data Controls panel, select a data collection.

Figure 41-20 shows the data collection for warehouse locations and product inventory levels.

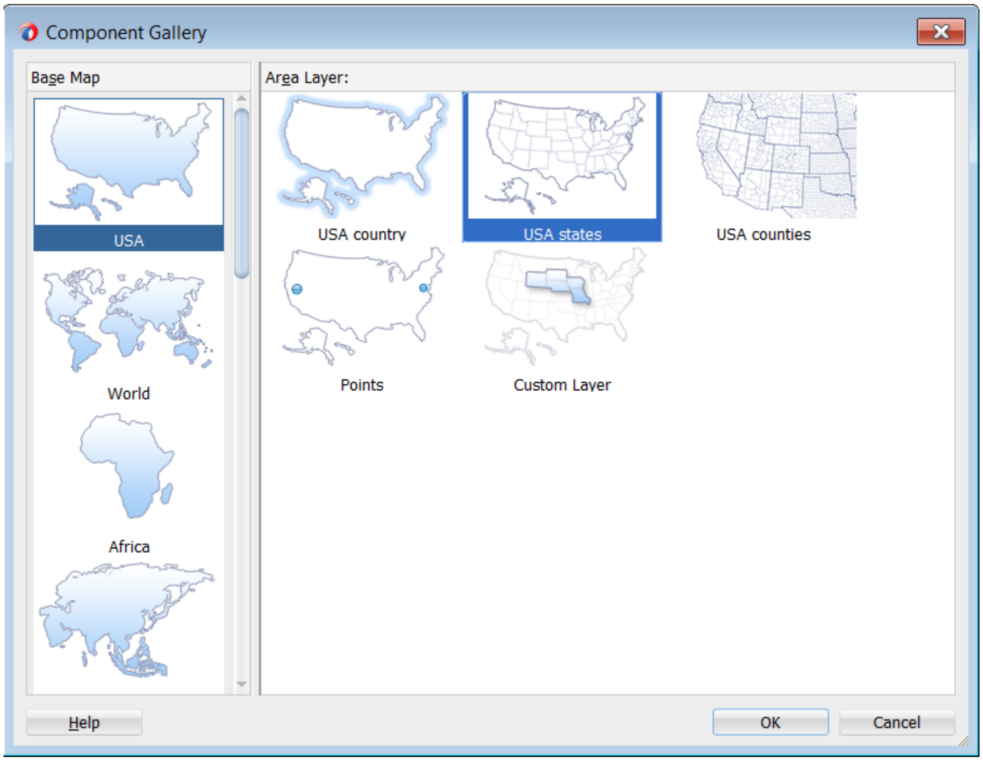
Figure 41-20 Data Collection for Warehouse Location and Inventory



Data Controls
- SWarehouseView1
  - Address
  - City
  - CountryId
  - Id
  - Latitude
  - Longitude
  - ManagerId
  - Phone
  - ProductCount
  - State
  - ZipCode
  - SInventoryView3
  - Operations
  - Named Criteria

2. Drag the collection onto a JSF page and, from the context menu, choose **Thematic Map**.
3. In the Component Gallery, select the map layer associated with the base map you want to configure for displaying data. In the example, a states map layer in the US base map is selected. Figure 41-21 shows the Component Gallery with the USA states map layer selected.

Figure 41-21 Thematic Map Component Gallery



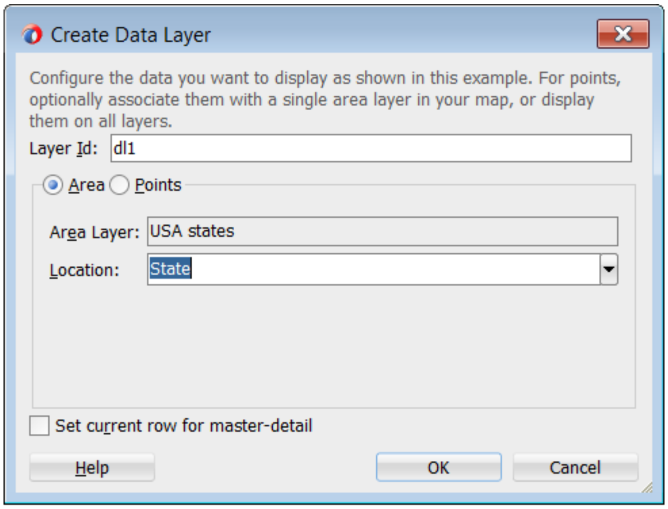
By default, the Create Data Layer dialog opens for adding an area or point data layer to the selected map layer. In the thematic map displayed in Figure 41-19, an area data layer is configured style USA states by warehouse product levels.

4. In the Create Data Layer dialog, enter the following:

   - **Layer Id**: Enter a unique name for the data layer you are defining. By default, a unique, consecutively numbered id is entered, `dl1`, `dl2`, and so on.
   - **Area**: Select to add an area data layer to the map layer.
   - **AreaLayer**: References the map layer to which you are adding a data layer. In the example, a **USA States** map layer.
   - **Location**: Select the attribute that represents the column in the data model that determines the location of the data for the areas in the data layer. The locations are Ids of the regions from the base map for which the data is being displayed. For more information, see What You May Need to Know About Base Map Location Ids.
   - **Set current row for master-detail**: Select if you want to enable master-detail relationships. This is useful when the data collection for the thematic map is a master in a master-detail relationship with a detail view that is displayed in another UI component on the page. For more information, see What You May Need to Know About Configuring Master-Detail Relationships.

   Figure 41-22 shows the completed Create Data Layer dialog.

Figure 41-22 Create Area Data Layer Dialog



Configure the data you want to display as shown in this example. For points, optionally associate them with a single area layer in your map, or display them on all layers.

Layer Id: dl1

● Area ○ Points

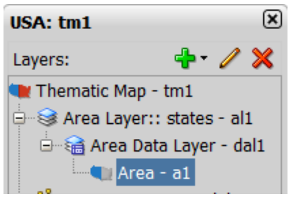
Area Layer: USA states

Location: State

□ Set current row for master-detail

An area data layer representing the `areaDataLayer` component, and an area, representing the `area` component is added in the Layer Browser hierarchy. Figure 41-23 shows the expanded Layer Browser after adding an area data layer and area to the map layer.

Figure 41-23 Thematic Map Layer Browser



USA: tm1

Layers:

Thematic Map - tm1
Area Layer:: states - al1
Area Data Layer - dal1
Area - a1

5. In the Layer Browser, expand the area data layer, select the area to be stylized, and click the **Edit** icon.

> **Note:**
>
> You configure an area with a default stamp across all areas in the thematic map layer, or you can you can use a child `attributeGroups` tag to generate the style attribute type automatically based on categorical groups in the data set. If the same style attribute is set in both the `area` tag, and by an `attributeGroups` tag, the `attributeGroups` style type will take precedence. For more information, see Styling Areas, Markers, and Images to Display Data.

6. In the Configure Area dialog, Attribute Groups page, enter the following:

   - **Grouping Rules**: Use this table to specify the styling of categorical groups of data in a data collection. Use the **Add** icon to add a row to the table for configuring rules for a categorical group and use the **Delete** icon to remove any row selected in the table. Each grouping rule is represented as a `attributeGroups` component, and assigned a unique, consecutively numbered Id, `ag1`, `ag2`, and so on.

     For each row added to the table, enter the following:

     – **Group by Value**: Enter or use the dropdown list to select the attribute representing the column in the data set by which you wish to group the data values. For example, `ProductCount` represents the warehouse product inventory levels by state.

     > **Note:**
     >
     > The selected attribute should consist of discrete values that can be categorized. For example, a range of numeric values between 40 and 45, are not automatically grouped.

     – **Area Properties**: Use the dropdown list to select the property to use for styling that area. Areas can be styled using **color**, **pattern**, **opacity**, or any combination of these valid values. Choose **Select multiple attributes** from the dropdown list for a dialog to specify any combination of values.

     The default style values that are generated for each property are defined using CSS style properties in the ADF skin. Each `attributeGroups` type has a default ramp defined in the skin, and these can be customized by setting the index-based properties to the desired values. For more information, see What You May Need to Know About Default Style Values for Attribute Groups.

  - **Legend Label**: Enter text or use the dropdown list to select the attribute representing the text to use for the categorical group in the thematic map legend. You can also select **Expression Builder** from the dropdown list to create an EL expression to specify the legend text. For more information, see Creating Databound Legends.

- **Value-Specific Rules**: Click to open the **Match Rules** and **Exception Rules** tables used to specify a finer detail for one or more data values in categorical groups in a data set. For example, use a match rule to style every state with a warehouse with a low product inventory level with a `#DD1E2F` color, instead of a predefined range of colors.

  > **Note:**
  >
  > Any match or exception rule specified in these tables will override the settings defined in the **Grouping Rules** table.

- **Match Rules**: Use to specify the style rule matched to one or more data values in a group of data in a data collection. Use the **Add** icon to add a row to the table for configuring a match rule for a categorical group and use the **Delete** icon to remove any row selected in the table. Each match rule is represented as a `attributeMatchRule` component, and assigned a unique, consecutively numbered Id, `amr1`, `amr2`, and so on. The property and property value is defined in a child `f:attribute` tag. For example:

  ```
  <dvt:attributeMatchRule id="amrl" group="Mountain Dew">
    <f:attribute name="color" value="#ffff00"/>
  </dvt:attributeMatchRule>
  ```

  For each row added to the table, enter the following:

  - **Group Value**: Enter the exact value for a **Group By Value** attribute that will trigger this Match Rule to execute. In the example, warehouse product inventory level data collection attribute, values include `low`, `med`, and high.

  - **Property**: Use the dropdown list to select the property to use for styling that data value. Areas can be styled using **color**, **pattern**, or **opacity** values. The property selected here must match one of the property types listed in the **Area Properties** for the attribute **Grouping Rules**.

  - **Property Value**: Enter or use the dropdown list to assign a value to the property. If the value provided by the match override is also in the prebuilt ramp returned by the **Grouping Rules**, then that value will only be used by the overrides and will be skipped in the prebuilt ramp.

    Valid values for **color** are RGB hexidecimal colors.

    Valid values for **pattern** include a choice of twelve prebuilt patterns, for example, **smallChecker**, **largeDiamond**, **smallDiagonalRight**, **largeCrosshatch**. If fill color is specified, the pattern displays in that color on the default white background.

    Valid values for **opacity** range from **0.0** for transparent to **1.0** for opaque.

- **Exception Rules**: Use to specify one or more exception to the style rules for categorical groups in the data set. Use the **Add** icon to add a row to the table for configuring an exception rule and use the **Delete** icon to remove any row selected in the table. Each exception rule is represented as an `attributeExceptionRule` component, and assigned a unique, consecutively numbered Id, `aer1`, `aer2`, and so on. The property and property value is defined in a child `f:attribute` tag.

  For each row added to the table, enter the following:

  – **Condition**: Enter an EL expression, or use the dropdown list to open an Expression Builder dialog to create an EL expression that replaces the style property value with another when certain conditions are met. For example:

    ```
    #{row.Sales gt 100000}
    ```

  – **Property**: Use the dropdown list to select the property to use for styling that data value. Areas can be styled using **color**, **pattern**, or **opacity** values. The property selected here must match one of the property types listed in the **Area Properties** for the attribute **Grouping Rules**.

  – **Property Value**: Enter or use the dropdown list to assign a value to the property. If the value provided by the match override is also in the prebuilt ramp returned by the **Grouping Rules**, then that value will only be used by the overrides and will be skipped in the prebuilt ramp.

    Valid values for **color** are RGB hexidecimal colors.

    Valid values for **pattern** include a choice of twelve prebuilt patterns, for example, **smallChecker**, **largeDiamond**, **smallDiagonalRight**, **largeCrosshatch**. If fill color is specified, the pattern displays in that color on the default white background.

    Valid values for **opacity** range from **0.0** for transparent to **1.0** for opaque.

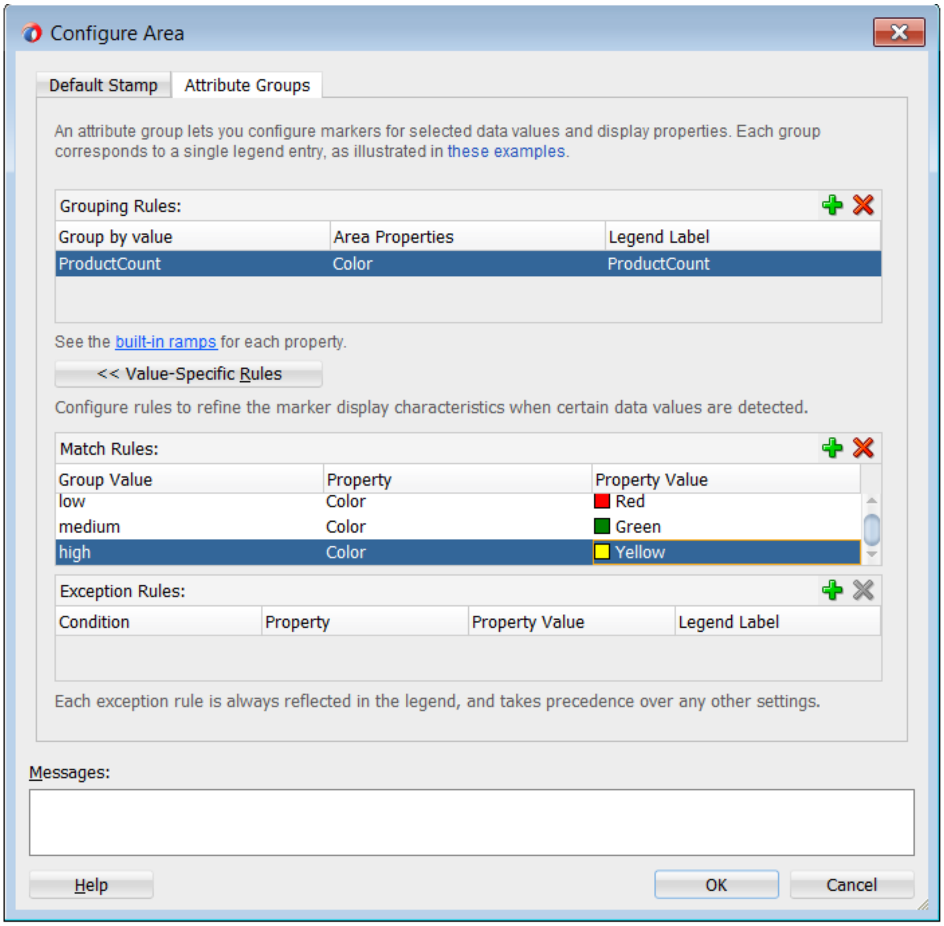
  – **Legend Label**: Enter a text resource to use for the legend label. The text resource can be a translatable string from a resource bundle or an EL expression executed at runtime. Use the dropdown list to open a Select Text Resource or Expression Builder dialog. If you need help, press F1 or click **Help**.

    > **Note:**
    >
    > the text resource option is only available for a fixed area.
    > For row-varying areas, use an EL expression to retrieve a row-varying key to look up the text resource in a resource bundle, for example:
    >
    > ```
    > #{viewController.ResourceBundle[row.label]}
    > ```

- **Messages**: Review and clear as necessary any alerts related to the configuration of the area.

Figure 41-24 shows the completed Configure Area dialog. The warning in the message pane alerts the user that the default area color specified in the Default Stamp page of the dialog will be overwritten by the color specified in the Grouping Rules and any value-specific overrides specified in the Attribute Groups page.

Figure 41-24 Configure Area Dialog Attribute Groups Page



Configure Area

An attribute group lets you configure markers for selected data values and display properties. Each group corresponds to a single legend entry, as illustrated in these examples.

Grouping Rules:

Group by value	Area Properties	Legend Label
ProductCount	Color	ProductCount

See the built-in ramps for each property.

Configure rules to refine the marker display characteristics when certain data values are detected.

Match Rules:

Group Value	Property	Property Value
low	Color	🟥 Red
medium	Color	🟩 Green
high	Color	🟨 Yellow

Exception Rules:

Condition	Property	Property Value	Legend Label

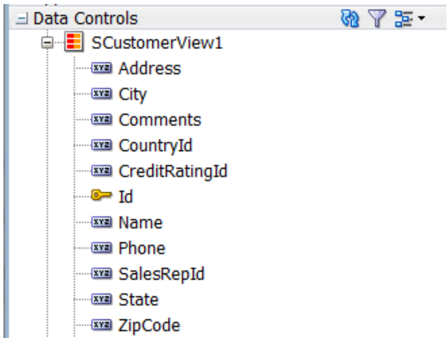
Each exception rule is always reflected in the legend, and takes precedence over any other settings.

Messages:

7. Use the Layer Browser to add a global point data layer representing warehouse locations using the same data collection you used to create the thematic map. For detailed instructions, see How to Add Data Layers to Thematic Maps.

8. Use the Layer Browser to add a global point data layer representing customer addresses using a different data collection than the one you used to create the thematic map. For detailed instructions, see How to Add Data Layers to Thematic Maps. Figure 41-25 shows the data collection for customer locations.

Figure 41-25 Data Collection for Customer Addresses

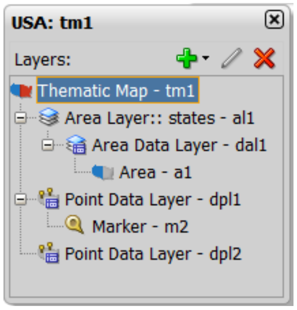


Data Controls
- SCustomerView1
  - Address
  - City
  - Comments
  - CountryId
  - CreditRatingId
  - Id
  - Name
  - Phone
  - SalesRepId
  - State
  - ZipCode

9. Use the Layer Browser to add and style a marker representing the customer locations in the global point data layer. For detail instructions, see Styling Areas, Markers, and Images to Display Data.

   Figure 41-26 shows the expanded Layer Browser for the thematic map in Figure 41-20.

   Figure 41-26 Thematic Map Layer Browser

10. Add and configure a data bound legend and labels associated with the styled points and areas. For detailed instructions, see Creating Databound Legends.

What Happens When You Use Data Controls to Create a Thematic Map

When you use ADF data controls to create a thematic map, JDeveloper:

- Defines the bindings for the thematic map in the page definition file of the JSF page, and
- Inserts code in the JSF page for the DVT thematic map components.

The example below shows the bindings defined in the page definition file of the JSF page for the example thematic map in Figure 41-19.

```
<bindings>
    <tree IterBinding="WorldProductInventory1Iterator"
id="WorldProductInventory1">
      <nodeDefinition DefName="model.WorldProductInventory"
Name="WorldProductInventory10">
        <AttrNames>
          <Item Value="Address"/>
          <Item Value="AmountInStock"/>
          <Item Value="Category"/>
          <Item Value="CategoryId"/>
          <Item Value="City"/>
          <Item Value="Country"/>
          <Item Value="CountryCode"/>
          <Item Value="CountryId"/>
          <Item Value="Id"/>
          <Item Value="Id1"/>
          <Item Value="Id2"/>
          <Item Value="Id3"/>
          <Item Value="Id4"/>
          <Item Value="Latitude"/>
          <Item Value="Longitude"/>
          <Item Value="ProdName"/>
          <Item Value="ProductId"/>
```

```
            <Item Value="RegName"/>
            <Item Value="RegionId"/>
            <Item Value="ReorderPoint"/>
            <Item Value="State"/>
            <Item Value="WarehouseId"/>
            <Item Value="ZipCode"/>
          </AttrNames>
        </nodeDefinition>
      </tree>
      <tree IterBinding="SCustomerView1Iterator" id="SCustomerView1">
        <nodeDefinition DefName="model.SCustomerView" Name="SCustomerView10">
          <AttrNames>
            <Item Value="Address"/>
            <Item Value="City"/>
            <Item Value="CountryId"/>
            <Item Value="Id"/>
            <Item Value="Name"/>
            <Item Value="State"/>
            <Item Value="ZipCode"/>
          </AttrNames>
        </nodeDefinition>
      </tree>
      <tree IterBinding="SWarehouseView1Iterator" id="SWarehouseView1">
        <nodeDefinition DefName="model.SWarehouseView" Name="SWarehouseView10">
          <AttrNames>
            <Item Value="Address"/>
            <Item Value="City"/>
            <Item Value="CountryId"/>
            <Item Value="Id"/>
            <Item Value="Latitude"/>
            <Item Value="Longitude"/>
            <Item Value="ManagerId"/>
            <Item Value="Phone"/>
            <Item Value="ProductCount"/>
            <Item Value="State"/>
            <Item Value="ZipCode"/>
          </AttrNames>
        </nodeDefinition>
      </tree>
    </bindings>
```

The example below shows the code inserted in the JSF page for the example thematic map in Figure 41-19.

```
<dvt:thematicMap id="tm1" basemap="usa" animationOnDisplay="alphaFade"
                 summary="#{viewcontrollerBundle.THEMATIC_MAP_DISPLAYING_
                 WAREHOUSE_PRODUCT_INVENTORY_LEVELS}">
        <dvt:areaLayer layer="states" id="al1">
          <dvt:areaDataLayer id="dal1"
                             value="#{bindings.SWarehouseView1.collectionModel}"
                             var="row">
            <dvt:areaLocation name="#{row.State}" id="al2">
              <dvt:area id="a1" shortDesc="#{row.ProductCount} units">
                <dvt:attributeGroups id="ag1" type="color"
                                     value="#{row.ProductCount le 5 ? 'low' :
                                     (row.ProductCount le 8 ? 'med' : 'high')}"
                                     label="#{row.ProductCount le 5 ? 'Low 
Product 
                                     Counts' : (row.ProductCount le 8 ? 'Good 
                                     Product Counts' : 'Surplus Product 
                                     Counts')}">
```

```
                  <dvt:attributeMatchRule id="amr1" group="low">
                    <f:attribute name="color" value="#DD1E2F"/>
                  </dvt:attributeMatchRule>
                  <dvt:attributeMatchRule id="amr2" group="med">
                    <f:attribute name="color" value="#EBB035"/>
                  </dvt:attributeMatchRule>
                  <dvt:attributeMatchRule id="amr3" group="high">
                    <f:attribute name="color" value="#218559"/>
                  </dvt:attributeMatchRule>
                </dvt:attributeGroups>
              </dvt:area>
            </dvt:areaLocation>
          </dvt:areaDataLayer>
        </dvt:areaLayer>
        <dvt:pointDataLayer id="dpl1"
                            value="#{bindings.SCustomerView1.collectionModel}"
                            var="row">
          <dvt:pointLocation type="pointName" pointName="#{row.State}_
                             #{fn:toUpperCase(fn:replace(row.City, ' ', '_'))}"
                             id="pl1">
            <dvt:marker id="m2" fillColor="#00000" opacity="1.0" shape="human"
                        scaleX="4.0" scaleY="4.0"
                        shortDesc="#{row.Name}">
              <f:attribute name="legendLabel" value="Customer Location"/>
            </dvt:marker>
          </dvt:pointLocation>
        </dvt:pointDataLayer>
        <dvt:pointDataLayer id="dpl2"
                            value="#{bindings.SWarehouseView1.collectionModel}"
                            var="row">
          <dvt:pointLocation type="pointXY" pointX="#{row.Longitude}"
                             pointY="#{row.Latitude}" id="pl2">
            <af:image id="img1" source="/images/normalHouse.gif"
                      shortDesc="#{row.Address}, #{row.City} #{row.ZipCode}">
              <f:attribute name="legendLabel" value="Warehouse Location"/>
            </af:image>
          </dvt:pointLocation>
        </dvt:pointDataLayer>
        <dvt:legend id="l1">
          <dvt:showLegendGroup id="slg1" label="Customer and Warehouse
Locations">
            <dvt:legendSection source="dpl1:m2" id="ls2"/>
            <dvt:legendSection source="dpl2:img1" id="ls1"/>
          </dvt:showLegendGroup>
          <dvt:showLegendGroup id="slg2" label="Warehouse Product Levels">
            <dvt:legendSection source="al1:dal1:ag1" id="ls3"/>
          </dvt:showLegendGroup>
        </dvt:legend>
      </dvt:thematicMap>
```

Once you have created a data bound thematic map you can add additional map layers representing available regions in the geographical hierarchy of the base map, and associate area or point data layers using the same data collection you used to create the thematic map. The Layer Browser represents the logical structure of the map layers, area and point data layers, and stylized areas and markers. Use the Layer Browser to:

- Add additional map layers in the base map geographical hierarchy to the thematic map.

- Define a custom map layer in the geographical hierarchy of the base map, using lower level regions to aggregate the regions in the custom layer. For more information, see How to Define a Custom Map Layer.
- Add, edit, or delete area or point (global or map layer specific) data layers. For more information, see How to Add Data Layers to Thematic Maps.
- Add, edit, or delete stylized areas or markers. For more information, see Styling Areas, Markers, and Images to Display Data.

After creating a thematic map using data controls, you can customize the default map labels, legend display, and add interactivity and animation effects. For more information, see in *Developing Web User Interfaces with Oracle ADF Faces*.

How to Add Data Layers to Thematic Maps

You use data layers to associate map layers with a data collection. Using stamping, each row of data in the data model can be identified by a style, for example a color or pattern; a marker, for example a circle or square; or an image. When a map layer is displayed at runtime, the data appears as stylized areas, markers, or images.

How to Add an Area Data Layer to a Map Layer

Map layers can display data using an area data layer and/or one or more point data layers. Area data layers can be styled using areas, markers, or images. Point data layers can be styled using markers or images.

Before you begin:

It may be helpful to have an understanding of databound thematic maps. For more information, see Creating Databound Thematic Maps.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Data Visualization Components.

You will need to complete these tasks:

- Create an application module that contains instances of the view objects that you want in your data model, as described in Creating and Modifying an Application Module.
- Create a JSF page as described in the How to Create JSF Pages section of *Developing Web User Interfaces with Oracle ADF Faces*.
- Create a data bound thematic map to your JSF page, as described in How to Create a Thematic Map Using ADF Data Controls.

To add an area data layer to a map layer:

1. Select the thematic map in the Visual Editor.
2. In the Layer Browser, select the map layer to which you wish to bind a row in the data collection. If the Layer Browser is not open in the Visual Editor, right-click inside the map and choose **Open Layer Browser**.
3. From the **Add** icon dropdown list, choose **Add Data Layer**.
4. In the Create Data Layer dialog, enter the following:

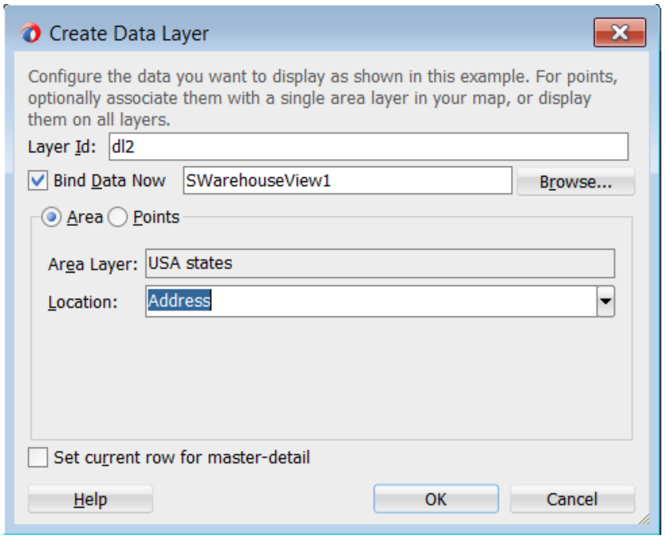
- **Layer Id**: Enter a unique name for the data layer you are defining. By default, a unique, consecutively numbered id is entered, `dl1`, `dl2`, and so on.
- **Bind Data Now**: Select and click **Browse** to open the Picker dialog > Data Controls Definitions page. Select the data collection in the ADF data controls you are using to data bind your data layer.

> **Note:**
>
> Alternatively, you can use the Expression Builder page to select an ADF managed bean you are using to data bind your area data layer and areas. You can also use the Expression Builder dialog in each remaining field.

- **Area**: Select to add an area data layer to the map layer.
- **AreaLayer**: References the map layer to which you are adding a data layer.
- **Location**: Select attribute that represents the column in the data model that determines the location of the data for the areas in the data layer. The locations are Ids of the regions from the base map for which the data is being displayed. For more information, see What You May Need to Know About Base Map Location Ids.
- **Set current row for master-detail**: Select if you want to enable master-detail relationships. This is useful when the data collection for the thematic map is a master in a master-detail relationship with a detail view that is displayed in another UI component on the page. For more information, see What You May Need to Know About Configuring Master-Detail Relationships.

Figure 41-27 shows the completed Create Area Data Layer dialog.

Figure 41-27 Create Area Data Layer Dialog



Create Data Layer

Configure the data you want to display as shown in this example. For points, optionally associate them with a single area layer in your map, or display them on all layers.

Layer Id: dl2

☑ Bind Data Now SWarehouseView1 Browse...

● Area ○ Points

Area Layer: USA states

Location: Address

□ Set current row for master-detail

By default, an area data layer representing the `areaDataLayer` component, and an area, representing the `area` component is added in the Layer Browser hierarchy. The example below shows the code added to the JSF page when you add an area data layer to a map layer.

```
<dvt:thematicMap id="tm1" basemap="usa">
  <dvt:areaLayer layer="counties" id="al1">
    <dvt:areaDataLayer id="dl2"
                       value="#{bindings.TmapStatesView11.collectionModel}"
                       var="row">
      <dvt:areaLocation name="#{row.Name}" id="al2">
        <dvt:area id="a1"/>
      </dvt:areaLocation>
    </dvt:areaDataLayer>
  </dvt:areaLayer>
...
</dvt:thematicMap>
```

After adding the area data layer to the thematic map, you style the data using areas, markers, or images. For more information, see Styling Areas, Markers, and Images to Display Data.

How to Add a Point Data Layer to a Map Layer

Map layers can display data using an area data layer and/or one or more point data layers. Area data layers can be styled using areas, markers, or images. Point data layers can be styled using markers or images.

When a point data layer is associated with the base map as a direct child of `thematicMap` instead of as a child of a specific map layer, the data displays at all times and is called a global point layer.

Before you begin:

It may be helpful to have an understanding of databound thematic maps. For more information, see Creating Databound Thematic Maps.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Data Visualization Components.

You will need to complete these tasks:

- Create an application module that contains instances of the view objects that you want in your data model, as described in Creating and Modifying an Application Module.
- Create a JSF page as described in the How to Create JSF Pages section of *Developing Web User Interfaces with Oracle ADF Faces*.
- Create a data bound thematic map to your JSF page, as described in How to Create a Thematic Map Using ADF Data Controls.

To add a point data layer to a map layer:

1. Select the thematic map in the Visual Editor.
2. In the Layer Browser, select the map layer to which you wish to bind a data layer. If the Layer Browser is not open in the Visual Editor, right-click inside the map and choose **Open Layer Browser**.
3. From the **Add** icon dropdown list, choose **Add Data Layer**.

> **Note:**
>
> If you wish to create a global point layer that displays at all times on the thematic map, choose **Create Global Point Layer**.

4. In the Create Data Layer dialog, enter the following:
   - **Layer Id**: Enter a unique ID for the data layer you are defining. By default, a unique, consecutively numbered id is entered, `dl1`, `dl2`, and so on.
   - **Bind Data Now**: Select and click **Browse** to open the Picker dialog > Data Controls Definitions page. Select the data collection in the ADF data controls you are using to data bind your data layer.

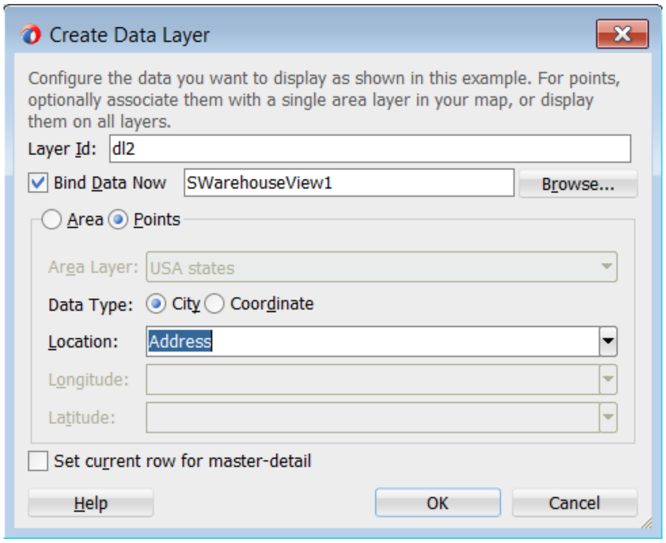
     > **Note:**
     >
     > Alternatively, you can use the Expression Builder page to select an ADF managed bean you are using to data bind your point data layer and markers. You can also use the Expression Builder dialog in each remaining field.

   - **Points**: Select to add a point data layer to the map layer.
   - **AreaLayer**: Use the dropdown list to select the map layer to which you are associating a data layer. If you select **All**, the point data layer will be configured as a global point layer and display at all times. If you select an available map layer, the point data layer will only display with that map layer is displayed.
   - **Data Type**: Choose one of the following:
     - **City**: Select to use a `pointName` data type representing the named points, such as cities, in the data collection that map to named points in the base map.
     - **Coordinates**: Select to use a `pointXY` data type representing the columns in the data collection that join `pointX` and `pointY` to define the point locations.
   - **Location**: Available if a City Data Type is specified. Use the dropdown list to select the attribute that represents the column in the data model that determines the location of the data for the points in the data layer. The locations are IDs of the points from the base map for which the data is being displayed. For more information, see What You May Need to Know About Base Map Location Ids.
   - **Longitude**: Available if a Coordinates Data Type is specified. Use the dropdown list to select the attribute in the data collection that represents the longitude, or `pointX` of the marker.
   - **Latitude**: Available if a Coordinates Data Type is specified. Use the dropdown list to select the attribute in the data collection that represents the latitude, or `pointY` of the markers.
   - **Set current row for master-detail**: Select if you want to enable master-detail relationships. This is useful when the data collection for the thematic map is

a master in a master-detail relationship with a detail view that is displayed in another UI component on the page. For more information, see What You May Need to Know About Configuring Master-Detail Relationships.

Figure 41-28 shows the completed Create Point Data Layer dialog for a `pointName` data type.

Figure 41-28   Create Point Data Layer Dialog



Create Data Layer

Configure the data you want to display as shown in this example. For points, optionally associate them with a single area layer in your map, or display them on all layers.

Layer Id: dl2

☑ Bind Data Now: SWarehouseView1

○ Area ● Points

Area Layer: USA states

Data Type: ● City ○ Coordinate

Location: Address

Longitude:

Latitude:

☐ Set current row for master-detail

By default, a point data layer representing the `pointDataLayer` component, and a marker, representing the `marker` component is added in the Layer Browser hierarchy. The example below shows the code added to the JSF page when you add a point data layer to a map layer.

```
<dvt:thematicMap id="tm1" basemap="usa">
  <dvt:areaLayer layer="states" id="al1">
    <dvt:pointDataLayer id="dl1"
                        value="#{bindings.TmapCitiesView11.collectionModel}"
                        var="row">
      <dvt:pointLocation type="pointName" pointName="#{row.City}" id="pl1">
        <dvt:marker id="m2"/>
      </dvt:pointLocation>
    </dvt:pointDataLayer>
  </dvt:areaLayer>
...
</dvt:thematicMap>
```

The example below shows the code added to the JSF page when you add a global point data layer to a thematic map.

```
<dvt:thematicMap>
  <areaLayer layer="states" id="al1"/>
  <dvt:pointDataLayer id="dl2"
                      value="#{bindings.TmapCitiesView12.collectionModel}"
                      var="row">
    <dvt:pointLocation type="pointXY" pointX="#{row.Longitude}"
                       pointY="#{row.Latitude}" id="pl2">
      <dvt:marker id="m3"/>
    </dvt:pointLocation>
  </dvt:pointDataLayer>
...
</dvt:thematicMap>
```

After adding the point data layer to the thematic map, you style the data using markers or images. For more information, see Styling Areas, Markers, and Images to Display Data.

Styling Areas, Markers and Images to Display Data

An area data layer is used to associate map layers with a data collection. Using stamping, each row of data in the data model can be identified by a style, for example a color or pattern; a marker, for example a circle or square; or an image.

A point data layer is used to associate a set of points on a map with a data collection. The data point can be specified by a named point in a map layer, for example, cities in the US map, or by longitude and latitude. Using stamping, each row of data in the data model can be identified by a marker, for example a circle or square, or an image.

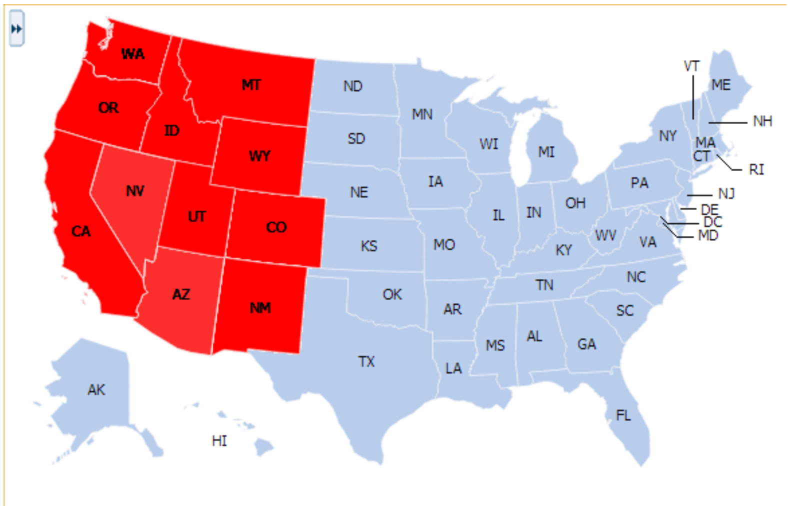
How to Style Areas to Display Data

You configure an area with a default stamp across all areas in the thematic map `areaLayer`, or you can use a child `attributeGroups` tag to generate the style attribute type automatically based on categorical groups in the data set. If the same style attribute is set in both the `area` tag, and by an `attributeGroups` tag, the `attributeGroups` style type will take precedence.

By default, when you add an area data layer, a configurable area is added to the Layer Browser. You can use the Layer Browser to add additional areas or markers to an area data layer, or markers to a point data layer. The Layer Browser reflects the logical structure of the map layers, area or point data layers, and areas or markers configured in the thematic map.

For example, you can style the states in the `states` layer of the `usa` base map with an attribute group to display states with warehouses using the color red as illustrated in Figure 41-29.

Figure 41-29 Thematic Map with Area Stamp Configured



Before you begin:

It may be helpful to have an understanding of databound thematic maps. For more information, see Creating Databound Thematic Maps.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Data Visualization Components.

You will need to complete these tasks:

- Create an application module that contains instances of the view objects that you want in your data model, as described in Creating and Modifying an Application Module.
- Create a JSF page as described in the How to Create JSF Pages section of *Developing Web User Interfaces with Oracle ADF Faces*.
- Create a data bound thematic map to your JSF page, as described in How to Create a Thematic Map Using ADF Data Controls.

To style areas using a default stamp:

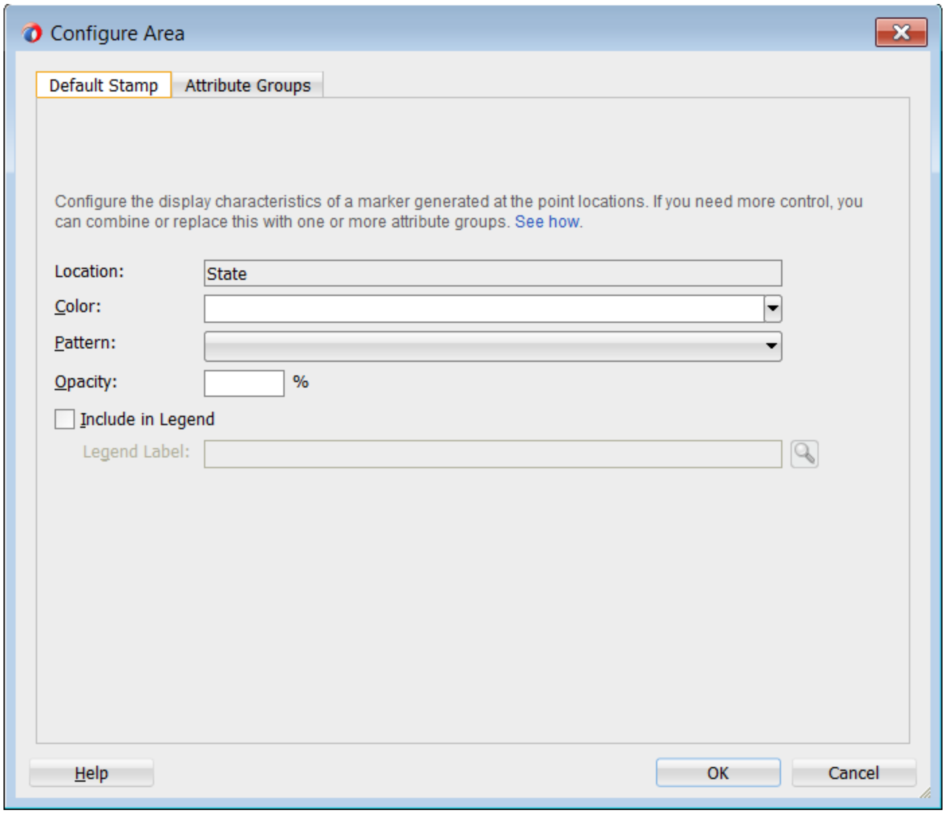
1. In the Layer Browser, select the area at the area location inside the map layer you are configuring.
2. Click the **Edit** icon.
3. In the Configure Area dialog, Default Stamp page, enter the following:
   - **Location**: By default, the attribute that represents the column in the data model that determines the location of the data for the areas in the data layer. The locations are Ids of the regions from the base map for which the data is being displayed. This read-only field displays the EL expression that maps the stamped area to a region in the base map. For more information, see What You May Need to Know About Base Map Location Ids.
   - **Color**: Optionally, from the dropdown list select the fill color for the area. Valid values are RGB hexidecimal. Choose **Custom Color** from the dropdown list to open a Color Picker dialog
   - **Pattern**: Optionally, from the dropdown list select one of twelve prebuilt patterns to style the area, for example, **smallChecker**, **largeDiamond**, **smallDiagonalRight**, **largeCrosshatch**. If fill color is specified, the pattern displays in that color on the default white background.
   - **Opacity**: Optionally, specify the opacity of the fill color of the area. Valid values range from **0.0** for transparent to **1.0** for opaque.
   - **Include in Legend**: Select and use the search icon to open a Select Text Resource dialog to select or create an application text resource to use for the legend text. The text resource can be a translatable string from a resource bundle or an EL expression executed at runtime. If you need help, press F1 or click **Help**.

> **Note:**
>
> the text resource option is only available for a fixed area. For row-varying areas, use an EL expression to retrieve a row-varying key to look up the text resource in a resource bundle, for example:
>
> `#{viewController.ResourceBundle[row.label]}`

- **Messages**: Review and clear as necessary any alerts related to the configuration of the area.

Figure 41-30 shows the Configure Area dialog for the Default Stamp page.

Figure 41-30 Configure Area Dialog Default Stamp Page



Configure the display characteristics of a marker generated at the point locations. If you need more control, you can combine or replace this with one or more attribute groups. See how.

Location: State

Color:

Pattern:

Opacity: %

□ Include in Legend

Legend Label:

The example below shows the code inserted in the JSF page for the configured area.

```
<dvt:thematicMap id="tm1" basemap="usa">
  <dvt:areaLayer layer="states" id="al1">
    <dvt:areaDataLayer id="dl1"
                       value="#{bindings.TmapStatesView1.collectionModel}"
                       var="row">
      <dvt:areaLocation name="#{row.Name}" id="al2">
        <dvt:area id="a1" fillColor="#ff0000">
          <f:attribute name="legendLabel"
                value="#{Bundle.US_STATES}"/>
        </dvt:area>
      </dvt:areaLocation>
    </dvt:areaDataLayer>
  </dvt:areaLayer>
</dvt:thematicMap>
```

If you wish to style areas using categorical groups of data in the data collection, use the Attribute Groups page of the Configure Area dialog. For details about using the Attribute Groups page including sample code, see the use case described in How to Create a Thematic Map Using ADF Data Controls.

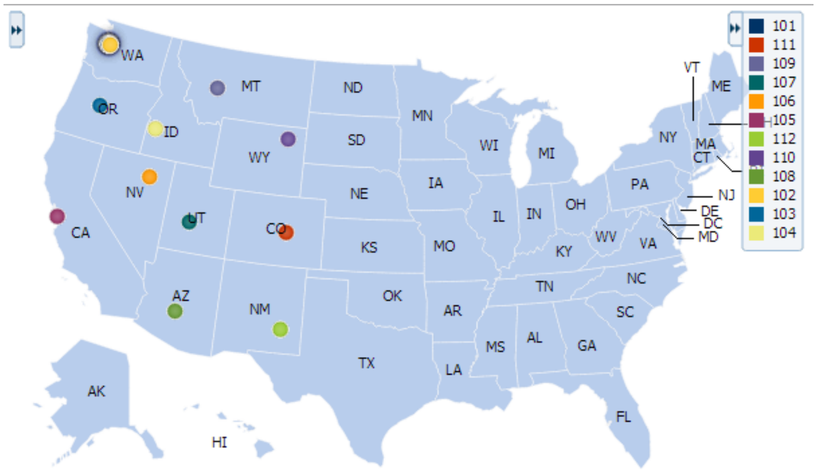
How to Style Markers to Display Data using a Default Stamp

You configure a marker with a default stamp across all markers in the thematic map layer, or you can use a child `attributeGroups` tag to generate the style attribute type automatically based on categorical groups in the data set. If the same style attribute is set in both the `marker` tag, and by an `attributeGroups` tag, the `attributeGroups` style type will take precedence.

By default, when you add a point data layer, a configurable marker element is added to the Layer Browser. You can use the Layer Browser to add additional areas or markers to an area data layer, or markers to a point data layer. The Layer Browser reflects the logical structure of the map layers, area or point data layers, and areas or markers configured in the thematic map.

For example, using the default stamp, you can identify all the predefined cities in the states layer of the USA base map with a red circle as illustrated in Figure 41-31. In the example, markers are styled on a point data layer. Styling markers on an area data layer is similar.

Figure 41-31   Thematic Map with Marker Default Stamp Configured



Before you begin:

It may be helpful to have an understanding of databound thematic maps. For more information, see Creating Databound Thematic Maps.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Data Visualization Components.

You will need to complete these tasks:

- Create an application module that contains instances of the view objects that you want in your data model, as described in Creating and Modifying an Application Module.
- Create a JSF page as described in the How to Create JSF Pages section of *Developing Web User Interfaces with Oracle ADF Faces*.
- Create a data bound thematic map to your JSF page, as described in How to Create a Thematic Map Using ADF Data Controls.

To style markers using a default stamp:

1. In the Layer Browser, select the marker at the point location inside the map layer you are configuring.
2. Click the **Edit** icon.
3. In the Configure Marker dialog, Default Stamp page, enter the following:
   - **Location**: By default, the attribute that represents the column in the data model that determines the location of the data for the markers in the data layer. The locations are Ids of the points from the base map for which the data is being displayed. This read-only field displays the EL expression that maps the stamped marker to a point in the base map. For more information, see What You May Need to Know About Base Map Location Ids.
   - **Color**: Optionally, from the dropdown list select the fill color for the marker. Valid values are RGB hexidecimal. Choose **Custom Color** from the dropdown list to open a Color Picker dialog.
   - **Pattern**: Optionally, from the dropdown list select one of twelve prebuilt patterns to style the marker, for example, **smallChecker**, **largeDiamond**, **smallDiagonalRight**, **largeCrosshatch**. If fill color is specified, the pattern displays in that color on the default white background.
   - **Opacity**: Optionally, specify the opacity of the fill color of the marker. Valid values range from **0.0** for transparent to **1.0** for opaque.
   - **Shape**: Optionally, select the shape of the marker. Valid values are **circle** (default), **square**, **plus**, **diamond**, **triangleUp**, **triangleDown**, and **human**.

     Optionally, use the **Custom Shape** field to specify the `shapePath` value to a Scalable Vector Graphics (SVG) file to use for the marker. Enter the path or use the **Search** icon to open the Select SVG File dialog and navigate to the file location. This option is only available if the default shape is selected in the **Shape** field.

     Shapes can also be specified using CSS style properties. Predefined marker shapes can be overwritten, and the paths to SVG files for custom markers can also be defined without using the `shapePath` attribute. For more information, see What You May Need to Know About Styling Markers.
   - **Size**: Optionally, enter a percentage by which to scale the marker from its default size for **ScaleX** (horizontal), and **ScaleY** (vertical). The percentages are then scaled to a float. For example, you can double the marker width by

setting **ScaleX** to `200`, written to the tag as `2.0`, and halve the height by setting **ScaleY** to `50`, written to the tag as `0.5`.

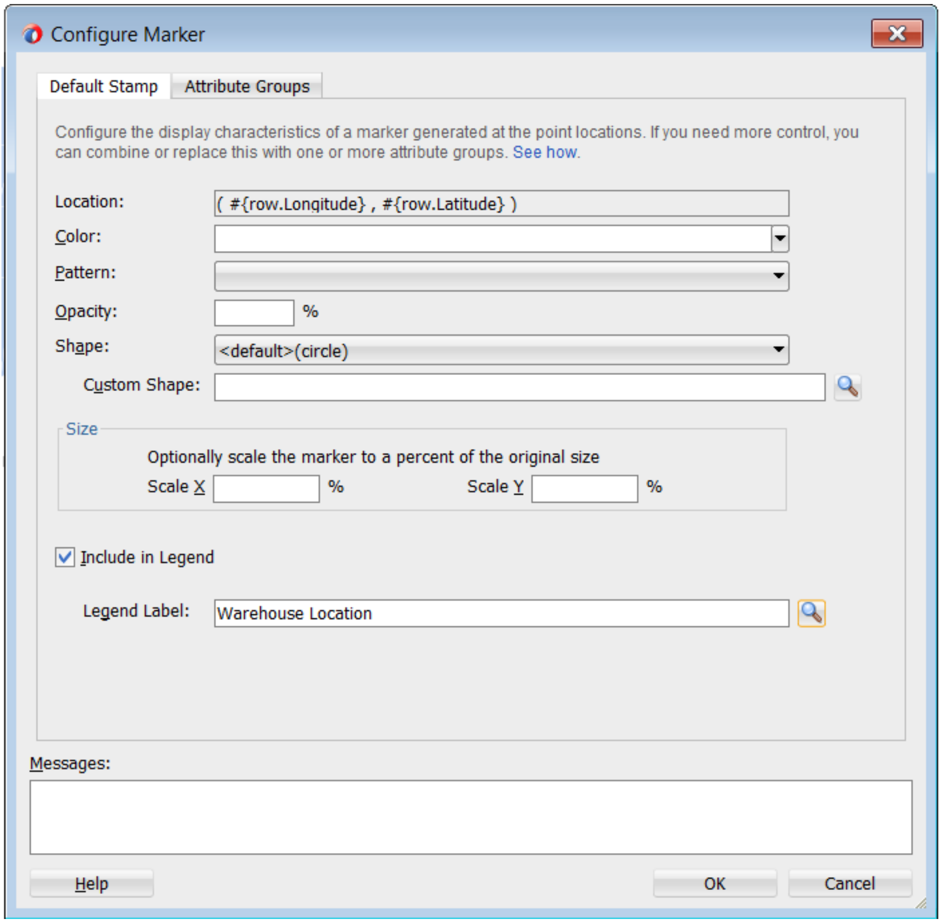
- **Include in Legend**: Enter a text resource to use for the legend label. The text resource can be a translatable string from a resource bundle or an EL expression executed at runtime. Use the dropdown list to open a Select Text Resource or Expression Builder dialog. If you need help, press F1 or click **Help**.

> **Note:**
>
> the text resource option is only available for a fixed area. For row-varying areas, use an EL expression to retrieve a row-varying key to look up the text resource in a resource bundle, for example:
>
> ```
> #{viewController.ResourceBundle[row.label]}
> ```

- **Messages**: Review and clear any messages related to the configuration of the marker.

Figure 41-32 shows the Configure Marker dialog, Default Stamp page.

Figure 41-32   Configure Marker Dialog Default Stamp Page



Configure the display characteristics of a marker generated at the point locations. If you need more control, you can combine or replace this with one or more attribute groups. See how.

Location: (#{row.Longitude} , #{row.Latitude})
Color:
Pattern:
Opacity: %
Shape: <default>(circle)
Custom Shape:

Size
Optionally scale the marker to a percent of the original size
Scale X % Scale Y %

☑ Include in Legend

Legend Label: Warehouse Location

Messages:

The example below shows the code inserted in the JSF page for the configured marker.

```
<dvt:thematicMap id="tm1" basemap="usa">
  <dvt:areaLayer layer="states" id="al1">
    <dvt:pointDataLayer id="dl2"
                        value="#{bindings.TmapCitiesView1.collectionModel}"
                        var="row">
      <<dvt:pointLocation type="pointName" pointName="#{row.City}" id="pl1">
        <dvt:marker id="m2" fillColor="#ff0000">
          <f:attribute name="legendLabel"
                value="#{Bundle.US_CITIES}"/>
        </dvt:marker>
      </dvt:pointLocation>
    </dvt:pointDataLayer>
  </dvt:areaLayer>
</dvt:thematicMap>
```

How to Style Markers to Display Data using Categorical Groups

You configure a marker with a default stamp across all markers in the thematic map layer, or you can use a child `attributeGroups` tag to generate the style attribute type automatically based on categorical groups in the data set. If the same style attribute is set in both the `marker` tag, and by an `attributeGroups` tag, the `attributeGroups` style type will take precedence.

If you wish to style markers using categorical groups of data in the data collection, use the Attribute Groups page of the Configure Marker dialog. You can configure markers for a point data layer or an area data layer.

Before you begin:

It may be helpful to have an understanding of databound thematic maps. For more information, see Creating Databound Thematic Maps.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Data Visualization Components.

You will need to complete these tasks:

- Create an application module that contains instances of the view objects that you want in your data model, as described in Creating and Modifying an Application Module.
- Create a JSF page as described in the How to Create JSF Pages section of *Developing Web User Interfaces with Oracle ADF Faces*.
- Create a data bound thematic map to your JSF page, as described in How to Create a Thematic Map Using ADF Data Controls.

To style markers using categorical groups of data:

1. In the Layer Browser, select the marker at the marker location inside the map layer you are configuring.
2. Click the **Edit** icon.
3. In the Configure Marker dialog, Attribute Groups page, enter the following:
   - **Grouping Rules**: Use this table to specify the styling of categorical groups of data in a data collection. Use the **Add** icon to add a row to the table

for configuring rules for a categorical group and use the **Delete** icon to remove any row selected in the table. Each grouping rule is represented as a `attributeGroups` component, and assigned a unique, consecutively numbered Id, `ag1`, `ag2`, and so on.

For each row added to the table, enter the following:

- **Group by Value**: Enter or use the dropdown list to select the attribute representing the column in the data set by which you wish to group the data values.

  > **Note:**
  >
  > The selected attribute should consist of discrete values that can be categorized. For example, a range of numeric values between 40 and 45, are not automatically grouped.

- **Marker Properties**: Use the dropdown list to select the property to use for styling that marker. Markers can be styled using **color**, **pattern**, **opacity**, **scaleX**, **scaleY**, or any combination of these valid values. Choose **Select multiple attributes** from the dropdown list for a dialog to specify any combination of values.

  The default style values that are generated for each property are defined using CSS style properties in the ADF skin. Each `attributeGroups` type has a default ramp defined in the skin, and these can be customized by setting the index-based properties to the desired values. For more information, see What You May Need to Know About Default Style Values for Attribute Groups.

- **Legend Label**: Enter text or use the dropdown list to select the attribute representing the text to use for the categorical group in the thematic map legend. You can also select **Expression Builder** from the dropdown list to create an EL expression to specify the legend text. For more information, see Creating Databound Legends.

- **Value-Specific Rules**: Click to open the **Match Rules** and **Exception Rules** tables used to specify a finer detail for one or more data values for categorical groups in a data set.

  > **Note:**
  >
  > Any match or exception rule specified in these tables will override the settings defined in the **Grouping Rules** table.

- **Match Rules**: Use to specify the style rule matched to one or more data values in a group of data in a data collection. Use the **Add** icon to add a row to the table for configuring a match rule for a categorical group and use the **Delete** icon to remove any row selected in the table. Each match rule is represented as a `attributeMatchRule` component, and assigned a unique, consecutively numbered Id, `amr1`, `amr2`, and so on. The property and property value is defined in a child `f:attribute` tag. For example:

```
<dvt:attributeMatchRule id="amrl" group="Mountain Dew">
  <f:attribute name="color" value="#ffff00"/>
</dvt:attributeMatchRule>
```

For each row added to the table, enter the following:

- **Group Value**: Enter the exact value for the **Group by Value** attribute that will trigger the Match Rule to execute.
- **Property**: Use the dropdown list to select the property to use for styling that data value. Markers can be styled using **color**, **pattern**, **opacity**, **scaleX**, **scaleY**, or any combination of these valid values. The property selected here must match one of the property types listed in the **Marker Properties** for the attribute **Grouping Rules**.
- **Property Value**: Enter or use the dropdown list to assign a value to the property. If the value provided by the match override is also in the prebuilt ramp returned by the **Grouping Rules**, then that value will only be used by the overrides and will be skipped in the prebuilt ramp.

  Valid values for **color** are RGB hexidecimal colors.

  Valid values for **pattern** include a choice of twelve prebuilt patterns, for example, **smallChecker**, **largeDiamond**, **smallDiagonalRight**, **largeCrosshatch**. If fill color is specified, the pattern displays in that color on the default white background.

  Valid values for **opacity** range from **0.0** for transparent to **1.0** for opaque.

  Valid values for **scaleX** and **scaleY** are percentages that are then scaled to a float.

- **Exception Rules**: Use to specify one or more exceptions to the style rules for categorical groups in the data set. Use the **Add** icon to add a row to the table for configuring an exception rule and use the **Delete** icon to remove any row selected in the table. Each exception rule is represented as an `attributeExceptionRule` component, and assigned a unique, consecutively numbered Id, `aer1`, `aer2`, and so on. The property and property value is defined in a child `f:attribute` tag. For example:

```
<dvt:attributeExceptionRule id="aer1" condition="#{row.name=='TX'}"
               label="Texas">
  <f:attribute name="color" value="#ff00ff"/>
</dvt:attributeExceptionRule>
```

For each row added to the table, enter the following:

- **Condition**: Enter an EL expression, or use the dropdown list to open an Expression Builder dialog to create an EL expression that replaces the style property value with another when certain conditions are met. For example:

  ```
  #{row.Sales gt 100000}
  ```

- **Property**: Use the dropdown list to select the property to use for styling that data value. Markers can be styled using **color**, **pattern**, **opacity**, **scaleX**, or **scaleY**. The property selected here must match one of the property types listed in the **Marker Properties** for the attribute **Grouping Rules**.
- **Property Value**: Enter or use the dropdown list to assign a value to the property. If the value provided by the match override is also in the prebuilt

ramp returned by the **Grouping Rules**, then that value will only be used by the overrides and will be skipped in the prebuilt ramp

Valid values for **color** are RGB hexidecimal colors.

Valid values for **pattern** include a choice of twelve prebuilt patterns, for example, **smallChecker**, **largeDiamond**, **smallDiagonalRight**, **largeCrosshatch**. If fill color is specified, the pattern displays in that color on the default white background.

Valid values for **opacity** range from **0.0** for transparent to **1.0** for opaque.

Valid values for **scaleX** and **scaleY** are percentages that are then scaled to a float.

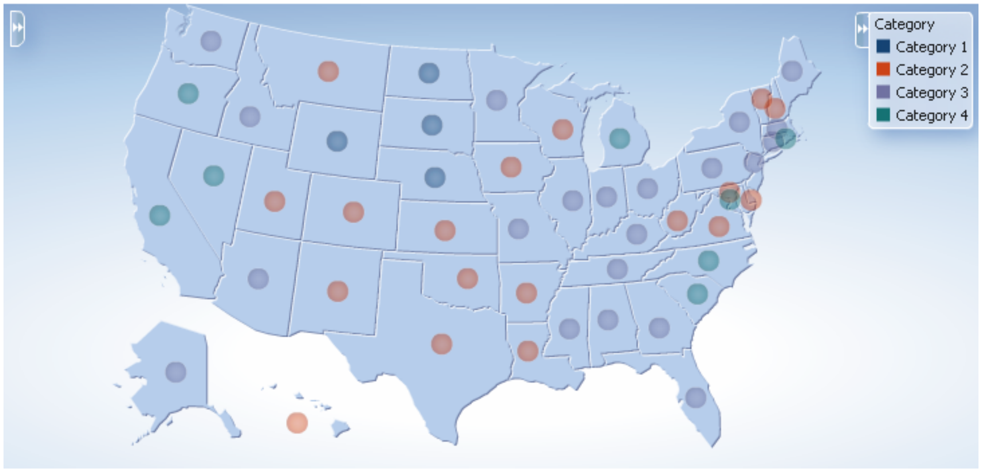
– **Legend Label**: Enter a text resource to use for the legend label. The text resource can be a translatable string from a resource bundle or an EL expression executed at runtime. Use the dropdown list to open a Select Text Resource or Expression Builder dialog. If you need help, press F1 or click **Help**.

> **Note:**
>
> the text resource option is only available for a fixed area.
> For row-varying areas, use an EL expression to retrieve a row-varying key to look up the text resource in a resource bundle, for example:
>
> ```
> #{viewController.ResourceBundle[row.label]}
> ```

- **Messages**: Review and clear as necessary any alerts related to the configuration of the marker.

For example, you can use markers in an area data layer to display categorical groups of data using colors as illustrated in Figure 41-33.

Figure 41-33 Marker Attribute Groups by Color

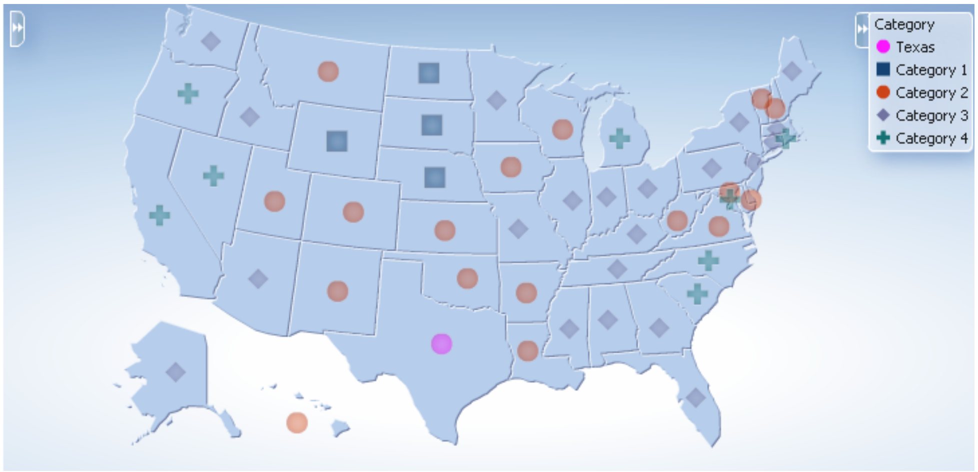


The example below shows the sample code for marker attribute groups by color.

```
<dvt:thematicMap basemap="usa" id="tm1"
  <dvt:areaLayer layer="states" id="al1" labelDisplay="off">
    <dvt:areaDataLayer id="dl1" var="row" value="#{stateData.colorModel}">
      <dvt:areaLocation id="al2" name="#{row.name}">
        <dvt:marker id="m1"
                    scaleX="3.0"
                    scaleY="3.0"
                    shape="circle">
          <dvt:attributeGroups id="ag1" type="color" value="#{row.category}"
                               label="#{row.category}"/>
        </dvt:marker>
      </dvt:areaLocation>
    </dvt:areaDataLayer>
  </dvt:areaLayer>
  <dvt:legend id="l1">
    <f:facet name="separator"/>
    <dvt:legendSection id="ls1" label="Category" source="ag1"/>
  </dvt:legend>
</dvt:thematicMap>
```

You can also use markers to display categorical groups of data using multiple attributes such as color and shape, and display an exception to the grouping rules when certain conditions are met. Figure 41-34 shows a thematic map with categorical groups using color and shape with an exception for the state of Texas.

Figure 41-34 Multiple Marker Attribute Groups with Exception Rule



The example below shows sample code for multiple marker attribute groups with exception rule.

```
<dvt:thematicMap basemap="usa" id="tm2"
  <dvt:areaLayer layer="states" id="al1" labelDisplay="off">
    <dvt:areaDataLayer id="dl1" var="row" value="#{stateData.colorModel}">
      <dvt:areaLocation id="al2" name="#{row.name}">
        <dvt:marker id="m1"
               scaleX="3.0"
               scaleY="3.0"
               shape="circle">
          <dvt:attributeGroups id="ag1" type="shape color"
value="#{row.category}"
               label="#{row.category}">
```

```
            <dvt:attributeExceptionRule id="aer1" condition="#{row.name=='TX'}"
               label="Texas">
              <f:attribute name="color" value="#ff00ff"/>
            </dvt:exceptionRule>
          </dvt:attributeGroups>
        </dvt:marker>
      </dvt:areaLocation>
    </dvt:areaDataLayer>
  </dvt:areaLayer>
  <dvt:legend id="l1">
    <f:facet name="separator"/>
    <dvt:legendSection id="ls1" label="Category" source="ag1"/>
  </dvt:legend>
</dvt:thematicMap>
```

What You May Need to Know About Styling Markers

Thematic maps support a predefined set of seven shapes (`circle`, `square`, and so on) that can be specified using the `shape` attribute in the `marker` component. For custom markers, the `shapePath` attribute can be used to specify the path of an SVG file that will get displayed in place of a predefined shape.

Scalable Vector Graphics (SVG) is the supported file format for creating custom shapes for thematic map markers.

SVG features that are not supported by custom shapes include:

- Image tags within the SVG file. Everything must be declared using SVG's vector shapes.
- Pattern fills
- Gradients on strokes

Marker shapes can be also specified through CSS style properties in an ADF skin. Using thematic map style properties, predefined marker shapes can be overwritten, and the paths to SVG files for custom markers can be defined without using the `shapePath` attribute. When using style properties, the `shape` attribute in the `marker` component is used for defining both predefined and custom shapes.

A predefined shape will be overwritten if a global or component-specific style property for that shape is specified in the ADF skin. For example, you can overwrite the predefined `circle` shape by specifying the `newCircle.svg` file in the thematic map component style property as follows:

```
af|dvt-thematicMap::shape-circle{
  -tr-path: url(/resources/path/newCircle.svg);
}
```

In the JSF page, the `marker` component `shape` attribute is set as follows:

```
<dvt:marker id="m1" shape="circle"/>
```

To specify a custom shape in the `marker` component `shape` attribute, you must use a prefix of `custom` in the shape style property name. For example, if the custom shape is named `customName`, then the ADF skin file should define either a global `.AFDVTShapeCustomName:alias` style property, or the thematic map specific `af|dvt-thematicMap::shape-customName` with the `-tr-path` property pointing to the SVG file as follows:

```
af|dvt-thematicMap::shape-customName{
  -tr-path: url(/resource/path/newCShape.svg);
}
```

In the JSF page, the marker component shape attribute is set as follows:

```
<dvt:marker id="m1" shape="customName"/>
```

For information about using ADF skins and style properties, see the Customizing the Appearance Using Styles and Skins chapter in *Developing Web User Interfaces with Oracle ADF Faces*.

What You May Need to Know About Default Style Values for Attribute Groups

The `attributeGroups` component is used to generate stylistic property values such as colors or shapes based on categorical grouping of the data in a data collection. Based on the attribute representing the column in the data model to group by, the `attributeGroups` component can generate style values for each unique value, or group, in the data.

The type of stylistic properties to generate values for is specified by the `type` attribute of the `attributeGroups` component. Supported types for `area` components are `color`, `pattern`, and `opacity`. Supported types for `marker` components are `color`, `shape`, `pattern`, `opacity`, `scaleX`, and `scaleY`. These types can be combined in a space-delimited list to generate multiple stylistic properties for each unique data value.

The default style values that are generated are defined using CSS style properties in the ADF skin. Each `attributeGroups` type has a default ramp defined in the ADF skin that can be customized by setting the index-based selectors to the desired values. The example below show sample code for using CSS style properties to specify attribute groups using CSS style properties.

```
af|dvt-attributeGroups::shape1{
  -tr-shape: square;
}
af|dvt-attributeGroups::shape2{
  -tr-shape: square;
}
...
af|dvt-attributeGroups::color1{
  -tr-fill-color: #003366;
}
```

The default ramps for each attribute groups type are displayed in Table 41-2.

Table 41-2 Default Ramps for Thematic Map Attribute Groups

Type	Default Ramps
`color`	j`#003366` (blue), `#CC3300` (red), `#666699` (lavender), `#0006666` (emerald), `#FF9900` (orange yellow), `#993366` (purple), `#99CC33` (lime green), `#624390` (violet,), `#669933` (green), `#FFCC33` (yellow), `#006699` (turquoise blue), and `#EBEA79` (pale yellow).
`shape`	`square`, `circle`, `diamond`, `plus`, `triangleDown`, `triangleUp`, and `human`

Table 41-2 (Cont.) Default Ramps for Thematic Map Attribute Groups

Type	Default Ramps
`pattern`	`smallDiagonalLeft`, `smallDiagonalRight`, `smallTriangle`, `smallChecker`, `smallChecker`, `smallCrosshatch`, `smallDiamond`, `largeDiagonalLeft`, `largeDiagonalRight`, `largeTriangle`, `largeChecker`, `largeCrosshatch`, `largeDiamond`
`opacity`	`0.25`, `0.50`, `0.75`, `1.0`
`scaleX` and `scaleY`	`1.0`, `2.0`, `3.0`, `4.0`, `5.0`

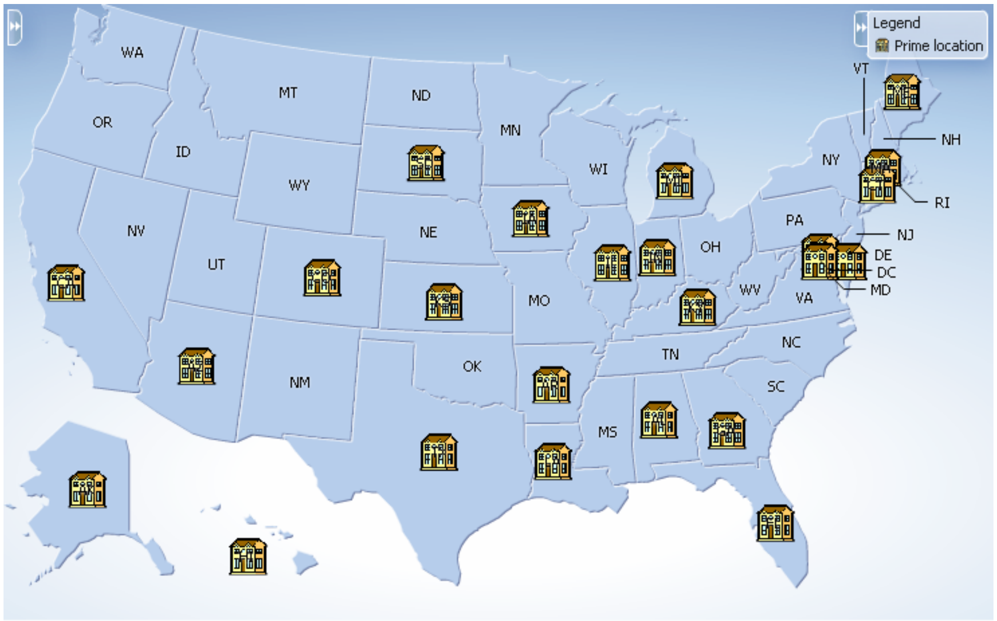
For information about using ADF skins and style properties, see the Customizing the Appearance Using Styles and Skins chapter in *Developing Web User Interfaces with Oracle ADF Faces*.

How to Use Images to Display Data

As an alternative to using built-in or custom shapes to style data in a thematic map, you can use images to represent data. Images can be associated with either an area or point data layer.

For example, in an area data layer you can use a house image to identify prime real estate locations on the states map layer of a US base map as illustrated in Figure 41-35.

Figure 41-35 Thematic Map Styling Data with Images



Before you begin:

It may be helpful to have an understanding of databound thematic maps. For more information, see Creating Databound Thematic Maps.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Data Visualization Components.

You will need to complete these tasks:

- Create an application module that contains instances of the view objects that you want in your data model, as described in Creating and Modifying an Application Module.
- Create a JSF page as described in the How to Create JSF Pages section of *Developing Web User Interfaces with Oracle ADF Faces*.
- Create a data bound thematic map to your JSF page, as described in How to Create a Thematic Map Using ADF Data Controls.

To use an image to display data:

1. In the Layer Browser, select the area or point data layer where you are configuring an image to display data.
2. In the Structure window, right-click the area or point data layer child `areaLocation` or `pointLocation` component and select **insert inside dvt:areaLocation** or **insert inside dvt:pointLocation > Image**.
3. In the Insert Image dialog, enter the following:
   - **Source**: Enter the URI specifying the location of the image source. You can use the dropdown menu to choose **Edit** and open an Edit Property: Source dialog or the Expression Builder to specify the location of the image source.
   - **ShortDesc**: Enter the short description of the image used as the `alt` text for screen reader users.

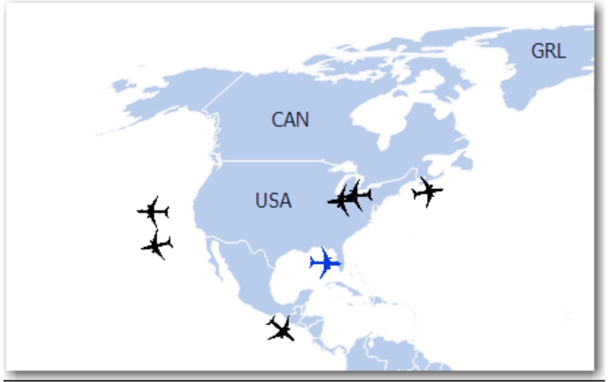
The example below shows a code sample for using images to display data for the thematic map illustrated in Figure 41-35.

```
<dvt:thematicMap id="thematicMap" imageFormat="flash" basemap="usa"
                 summary="Thematic map showing the important real estate
markets">
  <dvt:legend label="Legend">
    <dvt:legendSection source="areaLayer:dataLayer:img1"/>
  </dvt:legend>
  <dvt:areaLayer id="areaLayer" layer="states">
    <dvt:areaDataLayer id="dataLayer" contentDelivery="immediate"
                       value="#{tmapBean.colorModel}"
                       var="row"varStatus="rowStatus">
      <dvt:areaLocation id="dataLoc" name="#{row.name}">
        <af:image id="img1" source="/resources/images/geoMap/mansion.gif"
                            rendered="#{row.category == 'category1'}"
                            shortDesc="House image">
          <f:attribute name="legendLabel" value="Prime location"/>
        </af:image>
      </dvt:areaLocation>
    </dvt:areaDataLayer>
  </dvt:areaLayer>
</dvt:thematicMap>
```

How to Use a Marker Image Source to Display Data

You can configure a marker to use an image file to display data. Separate images can be specified to display user selected and hover states on the thematic map. Support for the rotation of images is also supported when using the `dvt:marker` component. For example, the thematic map in Figure 41-36 represents the locations and tracks of airplane flights, rotating the image as it moves along the track.

Figure 41-36   Thematic Map Flight Tracker



Before you begin:

It may be helpful to have an understanding of databound thematic maps. For more information, see Creating Databound Thematic Maps.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Data Visualization Components.

You will need to complete these tasks:

- Create an application module that contains instances of the view objects that you want in your data model, as described in Creating and Modifying an Application Module.
- Create a JSF page as described in the How to Create JSF Pages section of *Developing Web User Interfaces with Oracle ADF Faces*.
- Create a data bound thematic map to your JSF page, as described in How to Create a Thematic Map Using ADF Data Controls.

To use an marker image source to display data:

1. In the Layer Browser, select the area or point data layer where you are configuring an image to display data.
2. In the Structure window, right-click the area or point data layer child `areaLocation` or `pointLocation` component and select **insert inside Area Location** or **insert inside Point Location > Marker**.
3. In the Structure window, right-click the **dvt:marker** component and choose **Go to Properties**.
4. In the Properties window, expand the **Appearance** section and set the following attributes:
   - **Source**: Enter the URI specifying the location of the image source. You can use the dropdown menu to choose **Edit** and open an Edit Property: Source dialog or the Expression Builder to specify the location of the image resource.

- **SourceHover:** Optionally enter the URI specifying the location of the image resource on hover.
- **SourceHoverSelected:** Optionally enter the URI specifying the location of the selected image resource on hover.
- **SourceSelected**: Optionally enter the URI specifying the location of the image resource on selection.
- **ShortDesc**: Enter the short description of the image used as the `alt` text for screen reader users.

5. Optionally, expand the **Other** section and use the **Rotation** attribute to specify the marker rotation in clockwise degrees around the center of the image.

The example below shows the code sample for the thematic map flight tracker in Figure 41-36.

```
<dvt:thematicMap id="thematicMap" basemap="world" animationOnDisplay="none"
                 tooltipDisplay="auto"
                 summary="flight tracker demo">
  <dvt:areaLayer id="al1" layer="countries">
    <dvt:pointDataLayer id="pdl1" contentDelivery="immediate"
                        value="#{flightTrackerBean.flights}"
                        selectionListener="#{flightTrackerBean.processSelection}"
                        var="row"
                        varStatus="rowStatus"
                        selectionMode="multiple" partialTriggers="::::sbcb1"
                        selectedRowKeys="#{flightTrackerBean.selectedKeys}">
    <dvt:pointLocation id="pl1" type="pointXY" pointX="#{row.currentLongLat.x}"
                       pointY="#{row.currentLongLat.y}">
      <dvt:marker id="m1" labelPosition="bottom" scaleX="3" scaleY="3"
                  rotation="#{row.rotation}"
                  shortDesc="OracleAir#{row.flightNumber}"#{row.flightNumber}"
                  sourceHover="/resources/images/thematicMap/planeHover.png"
                  source="/resources/images/thematicMap/plane.png"
                  sourceSelected="/resources/images/thematicMap/planeSel.png"
                  sourceHoverSelected="/resources/images/thematicMap/
                                      planeHoverSel.png"
                  labelDisplay="#{flightTrackerBean.showFlightNo ? 'on' :
'off'}"
                  value="Flight #{row.flightNumber}"/>
      </dvt:pointLocation>
    </dvt:pointDataLayer>
  </dvt:areaLayer>
</dvt:thematicMap>
```

What You May Need to Know About Base Map Location Ids

Each base map provided for the thematic map component has two or more prebuilt map layers that represent a set of regions. For example, the `world` base map includes a map layer for `continents` and another layer for `countries`. The regions in the lower level map layers are aggregated to make up the next level in the geographical hierarchy. The map layer is specified in the `layer` attribute of the `areaLayer` component.

When you are binding your data collection to a thematic map, you must provide a column in the data model that specifies the location of the area or point data using the map location Ids of the regions from the base map for which the data is being displayed. Area locations are specified in the `name` attribute of the `areaLocation` component, and point locations are specified in the pointName attribute for the pointLocation component when its type attribute is set to pointName.

For the United States base map, the locations Ids are determined by the following naming rules:

- country layer: USA
- states layer: Use the two-letter postal abbreviation. For example, the location Id for Massachusetts is MA, and the location Id for Texas is TX.
- counties layer: Use the states layer location Id, followed by an underscore, and then the name of the county, all in capital letters with underscores replacing characters that are not letters. For example, the location Id for Middlesex county in Massachusetts is MA_MIDDLESEX, and the location Id for Red River county in Texas is TX_RED_RIVER.
- cities layer: Use the states layer location Id, followed by an underscore, and then the name of the city, all in capital letters with underscores replacing characters that are not letters. For example, the location Id for the id for Boston, Massachusetts is MA_BOSTON, and the location Id for San Antonio, Texas is TX_SAN_ANTONIO.

For all other base maps, location Ids are determined by the following naming rules:

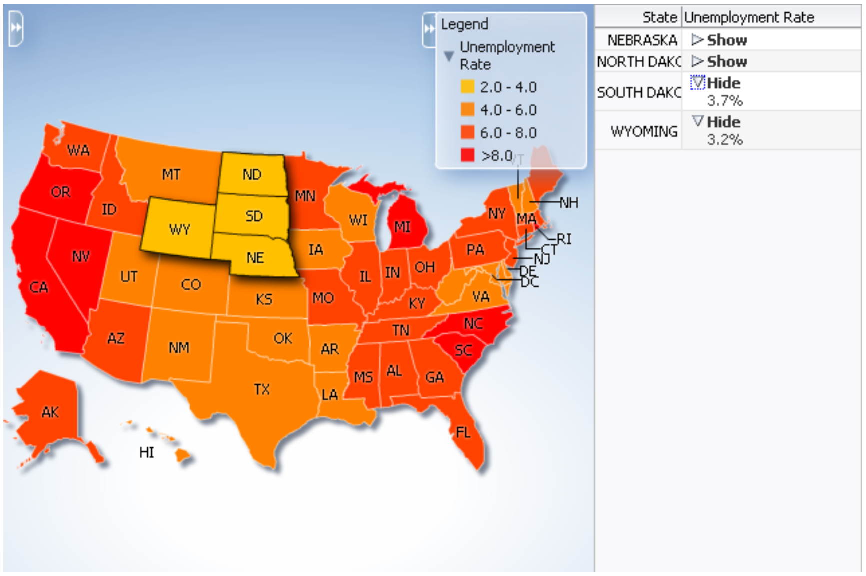
- continents layer: AF (Africa), AS (Asia), AU (Australia), EU (Europe), NA (North America), and SA (South America) for the world, africa, asia, australia, europe, northAmerica, and southAmerica base maps.
- worldRegions layers: APAC (Asia-Pacific), EMEA (Europe and the Middle East), LAT (Latin America), NA (United States and Canada) for the worldRegions, apac, emea, latinAmerica, and usaAndCanada base maps.
- countries layer: Use the ISO 3166-1 alpha-3 country codes. For example, the location Id for Canada is CAN, and the location Id for China is CHN.
- cities layer: Use the three-letter countries location ID, followed by an underscore, and then the name of the city, all in capital letters with underscores replacing characters that are not letters. For example, the location Id for Toronto, Canada is CAN_TORONTO, and the location Id for Los Angeles, United States is USA_LOS_ANGELES.

You can download a comma-separated value (CSV) file for each of the prebuilt map layers with a complete listing of all the thematic map base map location Ids. Find these links in the tag documentation for the areaLocation component, name attribute. To access tag documentation for the data visualization components, select the component in the Structure window and click the help button in the Properties window.

What You May Need to Know About Configuring Master-Detail Relationships

You can configure a thematic map to display its associated data in another UI component on the page such as a table. In this configuration the data collection for the thematic map is a master in a master-detail relationship with a detail view in another UI component. Figure 41-37 shows a thematic map displaying unemployment rates by state. Users can select one or multiple states to display the data detail view in the table.

Figure 41-37 Thematic Map Master and Detail Table View



State	Unemployment Rate
NEBRASKA	▷ **Show**
NORTH DAKC	▷ **Show**
SOUTH DAKC	▽ **Hide** 3.7%
WYOMING	▽ **Hide** 3.2%

The following requirements must be met to achieve this master-detail processing declaratively:

- You must use the same data collection to provide data for both views as follows:
  - Bind the thematic map `areaDataLayer` or `pointDataLayer` to the data collection whose attributes represent the data to be styled by areas or markers in the thematic map layer.
  - Bind the other ADF component (such as a table) to same data collection.
- Select **Set the current row for master-detail** in the Create Data Layer dialog to automatically set a value for the `selectionListener` attribute of the thematic map `areaLayer` component and use the `processSelection` method that is already part of the thematic map binding.

  For example, if the `value` attribute of the thematic map area data layer component is `value="#{stateData.employmentData}"`, then the `selectionListener` attribute is set to:

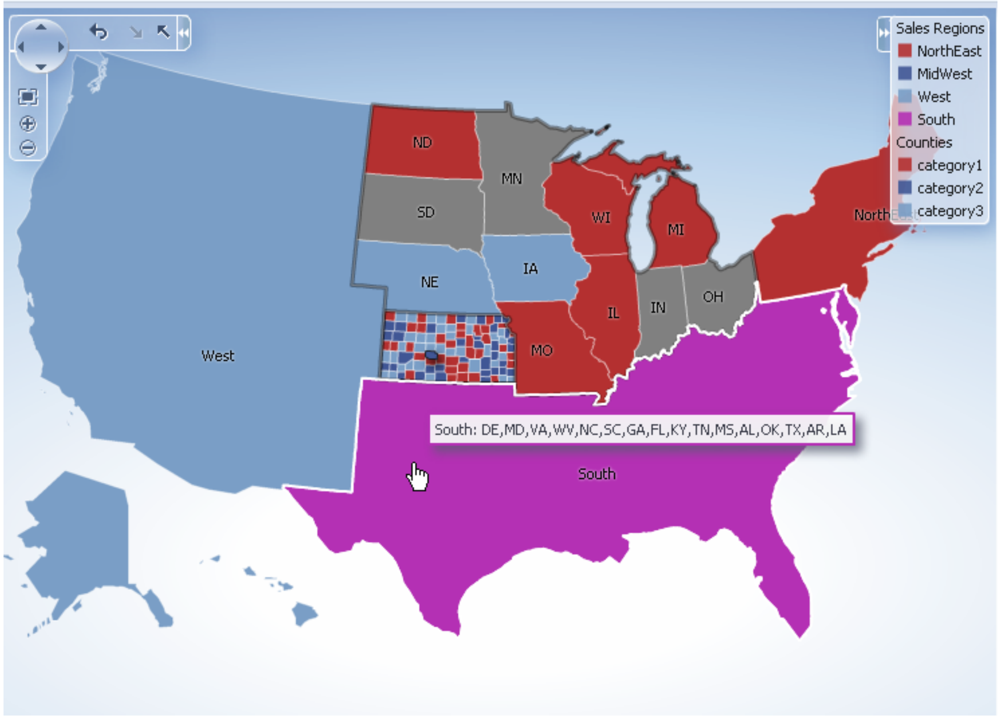
  ```
  selectionListener="#{stateData.employmentData.processSelection}".
  ```
- Ensure that the `selectionMode` attribute on the `areaDataLayer` or `pointDataLayer` component is set to `single` or `multiple`, depending on the requirements for the thematic map.

How to Define a Custom Map Layer

You can define a custom map layer from your own regional data and insert it into the natural geographical hierarchy of a thematic map. The custom layer is created by extending a predefined map layer and aggregating the lower level regions to form the new regions in the custom layer. After defining a custom map layer, it is used in the same way as any other map layer.

For example, you could define geographical regions for the NorthEast, Midwest, West and South in the United States in a US Regions custom map layer as illustrated in Figure 41-38. In the figure, the US Regions custom layer is extended from the states layer in the US base map. The new areas in the layer are aggregated from the states in the states layer. The label for the South US Region lists the states in that region. The MidWest US Region is drilled down to display the sales categories in the lower level counties region. For more information about drilling, see How to Configure Drilling in Thematic Maps.

Figure 41-38 US Regions Custom Map Layer



The `customAreaLayer` component uses a model to access the data in the underlying list. The specific model class is `oracle.adf.view.rich.model.CollectionModel`. You can also use other model instances, for example, `java.util.List`, `java.util.Array`, and `javax.faces.model.DataModel`. The `customAreaLayer` will automatically convert the instance into a `CollectionModel`.

Before you begin:

It may be helpful to have an understanding of how thematic map attributes and thematic map child tags can affect functionality. For more information, see the Configuring Thematic Maps section in *Developing Web User Interfaces with Oracle ADF Faces*.

You should already have an ADF data control or ADF managed bean that defines the aggregated areas in the predefined base map you are extending.

You should already have a thematic map on your page. If you do not, follow the instructions in this chapter to create a thematic map. For more information, see How to Create a Thematic Map Using ADF Data Controls.

How to add and configure a custom map layer:

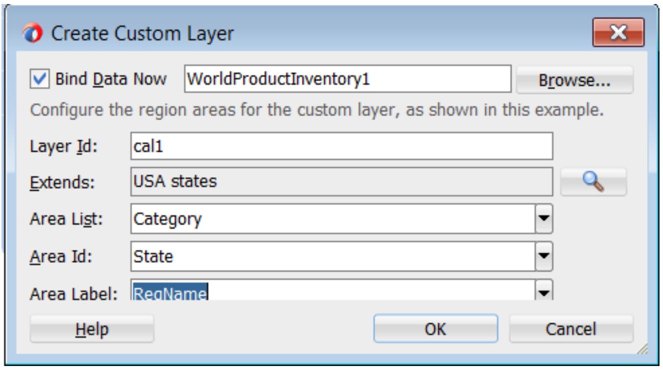
1. Select the thematic map in the Visual Editor.
2. In the Layer Browser, from the **Add** icon dropdown list, choose **Add Custom Layer**. If the Layer Browser is not open in the Visual Editor, right-click in the map and choose **Open Layer Browser**.
3. In the Create Custom Layer dialog, enter the following:
   - **Bind Data Now**: Select and click **Browse** to open the Picker dialog > Data Controls Definitions page. Select the data collection in the ADF data controls you are using to data bind your custom layer and areas.

     > **Note:**
     >
     > Alternatively, you can use the Expression Builder page to select an ADF managed bean you are using to data bind your custom layer and areas. You can also use the Expression Builder dialog in each remaining field.

   - **Layer Id**: Enter a unique identifier for the `customAreaLayer` component. For example, if you divide the US into aggregate regions (Northeast, Midwest, West, and South), then you might define them with corresponding Ids (NE, MW, W, and S).
   - **Extends**: Use the search icon to display the built-in map layers that can be used to aggregate areas for the custom layer. Select the map layer that the custom layer will extend.
   - **Area List**: Use the dropdown list to select the data collection attribute representing the list of lower level map regions that are used to aggregate the areas in the custom map layer. The comma separated list of values aggregate the regions in the may layer defined in the in the **Extends** attribute.
   - **Area Id**: Use the dropdown list to select the data collection attribute representing the unique identifier of lower level map regions that are used to aggregate the areas in the custom map layer. By default, a unique identifier, `ca1, ca2`, and so on, is used in the for the `customArea` component.
   - **Area Label**: Use the dropdown list to select the data collection attribute representing the names of lower level map regions that are used to aggregate the areas in the custom map layer. At runtime the label will display the comma separated list of the aggregated regions.

Figure 41-39 shows the completed Create Custom Layer dialog.

Figure 41-39   Create Custom Layer Dialog



☑ Bind Data Now: WorldProductInventory1

Configure the region areas for the custom layer, as shown in this example.

Layer Id: cal1

Extends: USA states

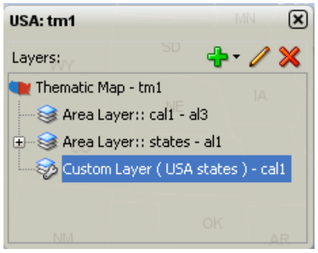
Area List: Category

Area Id: State

Area Label: RegName

[Figure 41-40](#) shows the Layer Browser after defining a custom layer. In the layer structure, the custom layer `cal1` is referenced in the map layer `al1`, where you add area or point data layers to display data. For more information, see [How to Add Data Layers to Thematic Maps](#).

Figure 41-40   Custom Map Layer in Layer Browser



USA: tm1

Layers:

- Thematic Map - tm1
  - Area Layer:: cal1 - al3
  - Area Layer:: states - al1
  - Custom Layer (USA states) - cal1

The example below shows the code inserted in the JSF page

```
<dvt:thematicMap>
...
  <dvt:areaLayer layer="states" id="al1"/>
  <dvt:areaLayer layer="cal1" id="al3"/>
  <dvt:customAreaLayer id="cal1"
                       value="#{bindings.TmapStatesView.collectionModel}"
                       var="row"
                       extendsLayer="states">
    <dvt:customArea areaId="#{row.RowID}" areaList="#{row.RowID}"
                    label="#{row.RowID}" id="ca1"/>
  </dvt:customAreaLayer>
...
 </dvt:thematicMap>
```

After configuring and adding a custom layer to the map layer hierarchy, you can then use the map layer in the same way as any other map layer. For example, to create the thematic map illustrated in [Figure 41-38](#), you will need to do the following:

- Configure the thematic map to support drilling. For more information, see [How to Configure Drilling in Thematic Maps](#).

> **Note:**
>
> If drilling is enabled for the thematic map, drilling between a custom layer and the map layer used to aggregate the custom layer is available without configuring data display for either layer.

- Add and configure an area data layer for the county data view. For more information, see How to Add Data Layers to Thematic Maps.
- Add and style area components to display the related data for each map layer. For more information, see Styling Areas, Markers, and Images to Display Data.
- Configure the thematic map legend. For more information, see Creating Databound Legends.

The example below shows sample code for using a custom layer in a thematic map.

```
<dvt:thematicMap id="thematicMap" imageFormat="flash" basemap="usa" drilling="on"
                 maintainDrill="true"
                 controlPanelBehavior="initExpanded" summary="US Custom Regions">
  <dvt:customAreaLayer id="crl1" value="#{tmapRegions.collectionModel}" var="row"
                       varStatus="rowStatus"
                       extendsLayer="states">
    <dvt:customArea areaId="#{row.name}" label="#{row.name}"
                    areaList="#{row.regions}" id="ca1"/>
  </dvt:customAreaLayer>
  <dvt:areaLayer id="custom" layer="crl1">
    <dvt:areaDataLayer contentDelivery="immediate"
                       value="#{tmapRegions.collectionModel}"
                       selectionMode="single"
                       var="row" varStatus="rowStatus" id="adl1">
      <dvt:areaLocation name="#{row.name}" id="al1">
        <dvt:area fillColor="#{row.color}" shortDesc="#{row.regions}"
                  id="a1" value="#{row.name}"/>
      </dvt:areaLocation>
    </dvt:areaDataLayer>
  </dvt:areaLayer>
  <dvt:areaLayer id="areaLayerS" layer="states"
    <dvt:areaDataLayer id="dataLayerS" selectionMode="multiple"
                       contentDelivery="immediate"
                       value="#{tmapStates.collectionModel}" var="row"
                       varStatus="rowStatus">
      <dvt:areaLocation id="areaLocS" name="#{row.name}">
        <dvt:area id="area1S" fillColor="#{row.color}"></dvt:area>
      </dvt:areaLocation>
    </dvt:areaDataLayer>
  </dvt:areaLayer>
  <dvt:areaLayer id="areaLayer" layer="counties">
    <dvt:areaDataLayer id="dataLayer" selectionMode="single"
                       contentDelivery="immediate"
                       value="#{tmapCounty.collectionModel}" var="row"
                       varStatus="rowStatus">
      <dvt:areaLocation id="areaLoc" name="#{row.name}">
        <dvt:area id="area1" fillColor="#{row.color}"
                        value="#{row.category}"></dvt:area>
      </dvt:areaLocation>
    </dvt:areaDataLayer>
  </dvt:areaLayer>
  <dvt:legend label="Sales Regions" id="l1">
```

```
    <dvt:legendSection source="custom:adl1:a1" id="ls1"/>
    <dvt:legendSection label="Counties" source="areaLayer:dataLayer:areaLoc"
                       id="ls3"/>
  </dvt:legend>
</dvt:thematicMap>
```

How to Configure Drilling in Thematic Maps

A thematic map with related data views in different map layers can be configured for drilling between the higher and lower level data views. For example, a thematic map can display data for sales category by USA state drilled down to USA county data as illustrated in Figure 41-38. When a state or county is selected, drilling up or down is initiated through a context menu choice or the Control Panel.

The following requirements must be met to achieve thematic map area drilling declaratively:

- For each map layer (`areaLayer`) in the drilling hierarchy, you must bind its child `areaDataLayer` with a data control that defines the related data for that map layer.
- Each `areaDataLayer` in the map layer drill hierarchy must have its `selectionMode` attribute set to `single` or `multiple`.
- You must configure the `area` in the lower level map layer drill hierarchy to display data using the same default stamp or categorical attribute style used in the higher level map layer `area` component.
- The `thematicMap` component `drilling` attribute must be set to `on`.

> **Note:**
>
> If drilling is enabled for the thematic map, drilling between a custom layer and the map layer used to aggregate the custom layer is available.

Before you begin:

It may be helpful to have an understanding of how thematic map attributes and thematic map child tags can affect functionality. For more information, see the Configuring Thematic Maps section in *Developing Web User Interfaces with Oracle ADF Faces*.

You should already have a thematic map on your page. If you do not, follow the instructions in this chapter to create a thematic map. For more information, see How to Create a Thematic Map Using ADF Data Controls.

You should already have data controls that define the data model for each of the map layers in the drill hierarchy.

To configure a thematic map area for drilling:

1. In the Structure window, select the **dvt:thematicMap** component.
2. In the Properties window, expand the **Behavior** section. Use this section to set the following attributes:

- **Drilling**: Use to enable drilling the area data view between thematic map layers. From the dropdown list select **on** to enable drilling. The default value is **off**.
- **MaintainDrill**: Optionally, use to specify an optional **true** value for maintaining the drilled state of a previously drilled area when a new area is drilled. The default value is **false**.
- **DrillBehavior**: Optionally, use to specify an optional **zoomToFit** effect on the area being drilled. The default value is **none**.

3. In the Layer Browser, select each **Area Layer** component in the desired drilling hierarchy and do the following:
   - Click the **Add icon** and choose **Add Data Layer** to open the Create Data Layer dialog. Complete the dialog to add an area data layer and bind the data layer to the data control for that map layer. If you need help, press F1 or click **Help**.

     > **Note:**
     >
     > If the **Area Data Layer** is already present, click the **Edit icon** to confirm binding to the data control.

   - In the Properties window, expand the **Behavior** section and set the **SelectionMode** attribute to **single** or **multiple**.
4. In the Layer Browser, select each **Area Data Layer** component in the desired drilling hierarchy and do the following:
   - Click the **Add icon** and choose **Add Area** to open the Configure Area dialog. Complete the dialog to define a default stamp or use attribute groups to style the area for that map layer. If you need help, press F1 or click **Help**.

     > **Note:**
     >
     > If the **Area** is already present, click the **Edit icon** to confirm the styling of the area.

Figure 41-40 shows sample code for drilling enabled for USA states and counties map layers with an area styled to display data about sales categories.

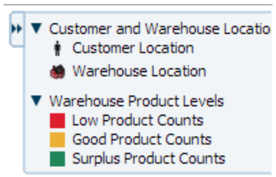
```
<dvt:thematicMap id="thematicMap"
                 basemap="usa"
                 drilling="on"
                 maintainDrill="true"
                 drillBehavior="zoomToFit"
                 animationOnDisplay="none"
  <dvt:areaLayer id="al1" layer="states">
    <dvt:areaDataLayer id="adl1"
                       selectionMode="single"
                       contentDeliver="immediate"
                       value="#{row.state}"
                       var="row"
                       var="rowStatus">
      <dvt:areaLocation id="areaLocS" name="#{row.stname}">
```

```
        <dvt:area id="a1" fillColor="#{row.state}"/>
      </dvt:areaLocation>
    </dvt:areaDataLayer>
  </dvt:areaLayer>
  <dvt:areaLayer id="al2" layer="counties">
    <dvt:areaDataLayer id="adl2"
                       selectionMode="single"
                       contentDelivery="immediate"
                       value="#{row.county}"
                       var="row"
                       varStatus="rowStatus">
      <dvt:areaLocation id="areaLocC" name="#{row.coname}">
        <dvt:area id="a2" fillColor="#{row.county}"/>
      </dvt:areaLocation>
    </dvt:areaDataLayer>
  </dvt:areaLayer>
...
</dvt:thematicMap>
```

Creating Databound Legends

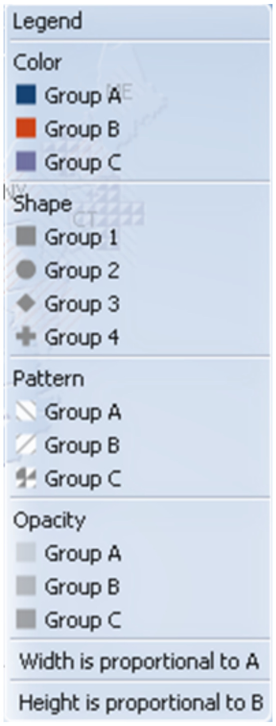
Legends provide an explanatory table of the thematic map's styled data in symbol and label pairs. Thematic map legend components (`legend`) support symbols for color, shape, custom shape, fill pattern, opacity, images and size. One or more child legend item components (`legendSection`) are sourced from thematic map `area`, `marker`, `attributeGroups`, or `af:image` components stamped to style the data displayed in the map. The legend section structure supports control over content ordering and appearance. You can wrap legend items into a disclosable section using a `showLegendSection` component. Figure 41-41 shows a legend with a disclosable section for map areas and a marker.

Figure 41-41   Legend with Disclosable Section and Marker



Legend items sourced from the `attributeGroups` component automatically split area or marker attribute types into different sections. You can specify a separator facet to draw separators between legend sections. Figure 41-42 shows a legend with attribute groups for color, shape, fill pattern, opacity, and size with separators between each section.

Figure 41-42   Legend with Attribute Groups



Legend

Color
- ■ Group A
- ■ Group B
- ■ Group C

Shape
- ■ Group 1
- ● Group 2
- ◆ Group 3
- ✚ Group 4

Pattern
- ▨ Group A
- ▨ Group B
- ▨ Group C

Opacity
- ■ Group A
- ■ Group B
- ■ Group C

Width is proportional to A

Height is proportional to B

Legends can be displayed in both Flash (default) and PNG image formats and both formats support locales with right-to-left display. When rendered in a PNG format, for example when printing the thematic map, disclosable sections in the legend are not supported, and legend items display as disclosed.

When you create a thematic map using the Data Controls panel and the thematic map binding dialogs, the legend data bindings are created for you. If you configure an area or marker as a default stamp across all areas in the thematic map, you can assign a static text resource to a fixed area or marker for the legend.

For default stamps displaying row-varying data, you can use an EL expression to assign the legend text and optionally, a managed bean to retrieve a row-varying key to look up the text resource in a resource bundle. The example below shows a code sample to generate legend entries for area and marker stamps. The code illustrates a disclosable section for a row-varying area and a fixed marker with an assigned text resource.

```
<dvt:thematicMap id="tm1" basemap="usa" ...>
  <dvt:legend label=“Legend“>
    <dvt:showLegendGroupLabel label="Voting Majority">
      <dvt:legendSection id="ls1" source="al1:adl1:areaStamp“/>
    </dvt:showLegendGroupLabel>
    <dvt:legendSection id="ls2" source="al1:adl1:fixedMarker">
  </dvt:legend>
  <dvt:areaLayer id="al1" layer="states">
    <dvt:areaDataLayer id="adl1"...>
      <dvt:areaLocation id="aloc1" ,,,>
        <dvt:area id="areaStamp">
                  fillColor="#{row.value > 50 ? tmapLegendBean,color1 :
                  tmapLegendBean.color2}"
          <f:attribute name="legendLabel" value="#{row.value > 50 ?
```

```
                          'Candidate 2' : 'Candidate 1'}" />
        <dvt:marker id=“fixedMarker“ shape=“human" fillColor=“"#FF9900"
                                     scaleX="3" scaleY="3">
          <f:attribute name="legendLabel“
                       value="#{Bundle.Office_Locations}"/>
        </dvt:marker>
      </dvt:areaLocation>
    </dvt:areaDataLayer>
  </dvt:area>
</dvt:thematicMap>
```

If you configure an area or marker to use an attributes group to specify the styling of categorical groups of data in a data collection, you can use an EL expression to assign the legend text and optionally, a managed bean to retrieve a row-varying key to look up the text resource in a resource bundle.

If you specify a match rule for the attributes group, the legend text is specified in the `group` attribute. If you specify an exception rule, you can specify a text resource from the application resource bundle. The example below shows sample code for a legend with attribute groups, including both a match rule and an exception rule, and specifying a separator between legend sections.

```
<dvt:thematicMap id="tm1" basemap="usa" ...>
  <dvt:legend id="l1" label="Legend">
    <f:facet name="separator"
      <af:separator/>
    <dvt:legendSection id="ls1" source="al1:adl1:attributeGroupColor" />
    <dvt:legendSection id="ls2" source="al1:adl1:attributeGroupShape" />
    <dvt:legendSection id="ls3" source="al1:adl1:attributeGroupPattern" />
    <dvt:legendSection id="ls4" source="al1:adl1:attributeGroupOpacity" />>
  </dvt:legend>
  <dvt:areaLayer id="al1" layer="states">
    <dvt:areaDataLayer id="adl1" value=" " var=" " ...>
      <dvt:areaLocation id="dataLoc" name="#{row.name}">
        <dvt:marker id="m1"... >
          <dvt:attributeGroups id="attributeGroupColor" type="color"
                 label="#{row.category1}" value="#{row.category1}" />
            <dvt:attributeMatchRule id="amrl" group="Mountain Dew">
              <f:attribute name="color" value="#ffff00"/>
          </dvt:attributeMatchRule>
          <dvt:attributeGroups id="attributeGroupShape" type="shape"
                 label="#{row.category2}" value="#{row.category2}" />
            <dvt:attributeExceptionRule id="aer1" condition="#{row.name=='TX'}"
                 label="#{viewcontroller.Texas}">
              <f:attribute name="shape" value="human"/>
            </dvt:exceptionRule>
          <dvt:attributeGroups id="attributeGroupShape" type="pattern"
                 label="#{row.category3}" ... />
          <dvt:attributeGroups id="attributeGroupShape" type="opacity"
                 label="#{row.category4}" ... />
        </dvt:marker>
      </dvt:areaLocation>
    </dvt:areaDataLayer>
  </dvt:areaLayer>
</dvt:thematicMap>
```

You configure thematic map legend areas or markers in a custom region in the same way as any other map layer. shows sample code for a custom map layer legend.

```
<dvt:thematicMap>
  <dvt:legend>
```

```
    <dvt:legendSection label=“Sales Regions" source="customAreaStamp"/>
      </dvt:legend>
      ...
  <dvt:customAreaLayer id="crl1“>
  ...
  </dvt:customAreaLayer>
  <dvt:areaLayer layer="crl1">
    <dvt:areaDataLayer var="row">
      <dvt:areaLocation name="#{row.name}" id="al1">
        <dvt:area id=“customAreaStamp" fillColor="#{row.color}“>
          <f:attribute name="legendLabel“ value=“#{row.name}"/>
        </dvt:area>
      </dvt:areaLocation>
    </dvt:areaDataLayer>
  </dvt:areaLayer>
</dvt:thematicMap
```

You can customize the appearance of thematic map legends using ADF skins. For more information, see the Customizing the Appearance Using Styles and Skins chapter in *Developing Web User Interfaces with Oracle ADF Faces*

42

Creating Databound Gantt Chart and Timeline Components

This chapter describes how to create Gantt charts and timelines from data modeled with ADF Business Components, using ADF data controls and ADF Faces components in a Fusion web application. Specifically, it describes how you can use ADF Data Visualization `projectGantt`, `resourceUtilizationGantt`, `schedulingGantt`, and `timeline` components to create Gantt charts and timelines that visually represent business data. It describes how to use ADF data controls to create these components with data-first development.
If you are designing your page using simple UI-first development, then you can add the Gantt chart or timeline to your page and configure the data bindings later. For information about the data requirements, tag structure, and options for customizing the look and behavior of the Gantt chart and timeline components, see the Using Gantt Chart Components and Using Timeline Components chapters in *Developing Web User Interfaces with Oracle ADF Faces*.

This chapter includes the following sections:

About ADF Data Visualization Components

ADF Data Visualization components provide extensive graphical and tabular capabilities for visually displaying and analyzing business data. Each component needs to be bound to data before it can be rendered since the appearance of the component is dictated by the data that is displayed.

There are three types of ADF Gantt chart components; a project Gantt chart used for project management, a scheduling Gantt chart used for resource scheduling, and a resource utilization Gantt chart used for displaying resource metrics. All Gantt charts display two regions combined with a splitter. The list region displays a list of tasks or resources, and the chart region displays task progress, resource utilization, or resource progress graphed over time.

The ADF timeline is an interactive data visualization tool that allows users to view events in chronological order and easily navigate forwards and backwards within a defined time range. Events are represented as timeline items using simple ADF components to display information such as text and images, or supply actions such a links. A dual timeline can be configured to display two series of events to allow a side-by-side comparison of related information.

The prefix `dvt:` occurs at the beginning of each data visualization component name indicating that the component belongs to the ADF Data Visualization Tools (DVT) tag library.

Data Visualization Components Use Cases and Examples

For detailed descriptions of each data visualization use cases and examples, see the following:

- Gantt chart components: Gantt Chart Component Use Cases and Examples section in *Developing Web User Interfaces with Oracle ADF Faces*.
- Timeline components: Timeline Use Cases and Examples section in *Developing Web User Interfaces with Oracle ADF Faces*.

End User and Presentation Features

Visually compelling data visualization components enable end users to understand and analyze complex business data. The components are rich in features that provide out-of-the-box interactivity support. For detailed descriptions of the end user and presentation features for each component, see the following:

- Gantt chart components: End User and Presentation Features section in *Developing Web User Interfaces with Oracle ADF Faces*.
- Timeline components: End User and Presentation Features section in *Developing Web User Interfaces with Oracle ADF Faces*.

Additional Functionality for Data Visualization Components

You may find it helpful to understand other **Oracle ADF** features before you data bind your data visualization components. Additionally, once you have added a data visualization component to your page, you may find that you need to add functionality such as validation and accessibility. Following are links to other functionality that data visualization components use:

- Partial page rendering: You may want a data visualization component to refresh to show new data based on an action taken on another component on the page. For more information, see the Rerendering Partial Page Content chapter in *Developing Web User Interfaces with Oracle ADF Faces*.
- Personalization: Users can change the way the data visualization components display at runtime, those values will not be retained once the user leaves the page unless you configure your application to allow user customization. For more information, see the Allowing User Customization on JSF Pages chapter in *Developing Web User Interfaces with Oracle ADF Faces*.
- Accessibility: By default, data visualization components are accessible. You can make your application pages available to screen readers. For more information, see the Developing Accessible ADF Faces Pages chapter in *Developing Web User Interfaces with Oracle ADF Faces*.
- Skins and styles: You can customize the appearance of data visualization components using an ADF skin that you apply to the application or by applying CSS style properties directly using a style-related property (`styleClass` or `inlineStyle`). For more information, see the Customizing the Appearance Using Styles and Skins chapter in *Developing Web User Interfaces with Oracle ADF Faces*.
- Placeholder data controls: If you know the data visualization components on your page will eventually use ADF data binding, but you need to develop the pages

before the data controls are ready, then you should consider using placeholder data controls, rather than manually binding the components. Using placeholder data controls will provide the same declarative development experience as using developed data controls. For more information, see Designing a Page Using Placeholder Data Controls.

Creating Databound Gantt Charts

A Gantt chart is a type of bar chart displayed on the horizontal axis. It is used in planning and tracking projects to show tasks or resources in a time frame with a distinct beginning and end.

When you create a Gantt chart, you can choose from the following types:

- Project

  A project Gantt chart lists tasks vertically and shows the duration of each task as a bar on a horizontal time line.

- Resource Utilization

  A resource utilization Gantt chart shows graphically whether resources are over or under allocated. It shows resources vertically while showing their allocation and, optionally, capacity on the horizontal time axis.

- Scheduling

  A scheduling Gantt chart is based on manual scheduling boards and shows resources vertically with corresponding activities on the horizontal time axis. Examples of resources include people, machines, or rooms.

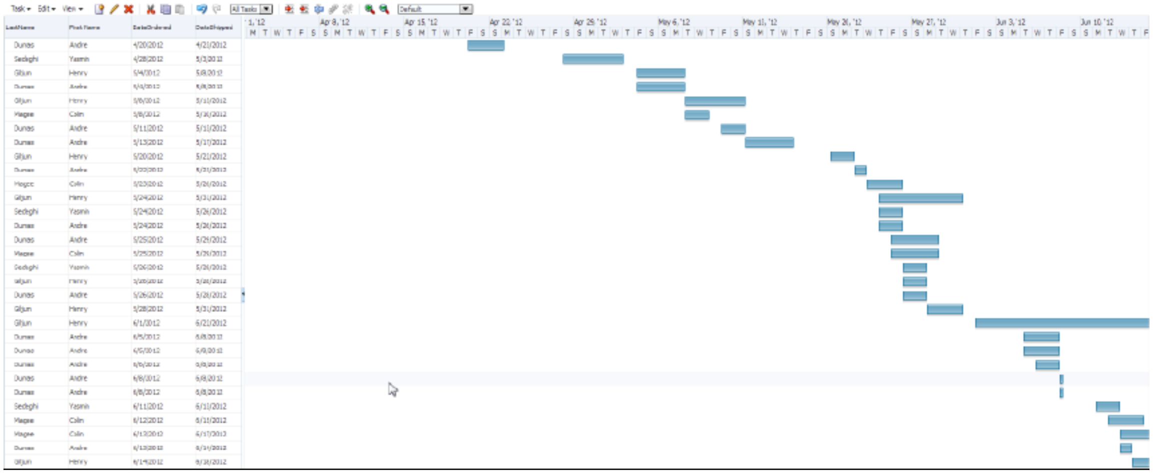
How to Create a Databound Project Gantt Chart

For a project Gantt chart, you must specify values for tasks. Optionally, you can specify values for split tasks, subtasks, recurring tasks, and dependencies between tasks, if your data collection has accessors for this additional information.

The project Gantt chart is displayed with default values for overall start time and end time and for the major and minor time axis values. In a project Gantt chart, the setting for the major time axis defaults to weeks and the setting for the minor time axis defaults to days.

Figure 42-1 shows a project Gantt chart in which each task is an order to be filled. The list region on the left side of the splitter shows columns with the name of the person responsible for the order and columns for the order date and ship date. In the chart region on the right side of the splitter, the Gantt chart displays a horizontal bar from the order date to the ship date for each order.

Figure 42-1 Project Gantt Chart for Orders Shipped



LastName	FirstName	DateOrdered	DateShipped
Dumas	Andre	4/20/2012	4/21/2012
Sedeghi	Yasmin	4/28/2012	5/3/2012
Giljun	Henry	5/4/2012	5/8/2012
Dumas	Andre	5/4/2012	5/8/2012
Giljun	Henry	5/6/2012	5/11/2012
Magee	Colin	5/8/2012	5/10/2012
Dumas	Andre	5/11/2012	5/11/2012
Dumas	Andre	5/13/2012	5/17/2012
Giljun	Henry	5/20/2012	5/21/2012
Dumas	Andre	5/22/2012	5/21/2012
Magee	Colin	5/23/2012	5/26/2012
Giljun	Henry	5/24/2012	5/31/2012
Sedeghi	Yasmin	5/24/2012	5/26/2012
Dumas	Andre	5/24/2012	5/26/2012
Dumas	Andre	5/25/2012	5/29/2012
Magee	Colin	5/25/2012	5/29/2012
Sedeghi	Yasmin	5/26/2012	5/26/2012
Giljun	Henry	5/26/2012	5/28/2012
Dumas	Andre	5/26/2012	5/28/2012
Giljun	Henry	5/28/2012	5/31/2012
Giljun	Henry	6/1/2012	6/21/2012
Dumas	Andre	6/5/2012	6/8/2012
Dumas	Andre	6/5/2012	6/8/2012
Dumas	Andre	6/6/2012	6/8/2012
Dumas	Andre	6/8/2012	6/8/2012
Dumas	Andre	6/8/2012	6/8/2012
Sedeghi	Yasmin	6/11/2012	6/11/2012
Magee	Colin	6/12/2012	6/15/2012
Magee	Colin	6/13/2012	6/17/2012
Dumas	Andre	6/13/2012	6/14/2012
Giljun	Henry	6/14/2012	6/18/2012

To create a project Gantt chart using a data control, bind the project Gantt chart component to a data collection. JDeveloper allows you to do this declaratively by dragging and dropping a collection from the Data Controls panel.

> **Tip:**
>
> You can also create a project Gantt chart by dragging a project Gantt chart component from the Components window and completing the Create Project Gantt dialog. This approach allows you to design the Gantt chart user interface before binding the component to data.

Before you begin:

It may be helpful to have an understanding of databound Gantt charts. For more information, see Creating Databound Gantt Charts.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Data Visualization Components.

You will need to complete these tasks:

- Create an **application module** that contains instances of the **view objects** that you want in your data model, as described in Creating and Modifying an Application Module.

  For example, the data source for the project Gantt chart in Figure 42-1 comes from a view object representing shipping orders including their shipping and delivery dates, in the Summit sample application for ADF DVT components.

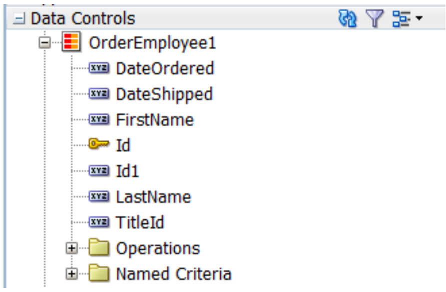
- Create a JSF page as described in the How to Create JSF Pages section of *Developing Web User Interfaces with Oracle ADF Faces*.

To create a project Gantt chart using the Data Controls panel:

1. From the Data Controls panel, select a data collection. Select a row set collection that produces a flat list of tasks, or a basic tree collection that produces a hierarchical list of tasks.

   Figure 42-2 shows an example where you could select the `OrderEmployee1` collection in the Data Controls panel to create a project Gantt chart that displays the progress of order shipping.

   Figure 42-2 Data Collection for Shipping Orders

   Data Controls
   - OrderEmployee1
     - DateOrdered
     - DateShipped
     - FirstName
     - Id
     - Id1
     - LastName
     - TitleId
     - Operations
     - Named Criteria

2. Drag the collection onto a JSF page and, from the context menu, choose **Gantt** > **Project**.
3. In the Create Project Gantt dialog, in the **Tasks** page, select the columns in the data collection that correspond to the following attributes:
   - **Task Id**: The unique Id for the tasks represented in the Gantt chart.
   - **Start Time**: The starting time of the task.
   - **End Time**: The ending time of the task.
   - **Task Type**: The type used to specify the appearance of the task bar when it is rendered in the Gantt chart. If you do not bind the type to a column in your data collection, then all tasks default to `Normal`. Task types include:
     - `Normal`: The basic task type, a plain horizontal bar that shows the start time, end time, and duration of the task.
     - `Summary`: Shows start time and end time for a group of tasks, typically subtasks. A summary task cannot be moved or extended. However, your application should recalculate the summary task times when the dates of any subtask changes.
     - `Milestone`: Shows a specific date in the Gantt chart. There is only one date associated with a milestone task. A milestone task cannot be extended but it can be moved.
4. Click **Show More** to select additional columns in the data collection that correspond to the following attributes for the specified task type:
   - **Actual Start** and **Actual End**: When these attributes are specified, instead of drawing one bar, two bars are drawn. One bar indicates the base start and end date, the other bar indicates the actual start and end date. This is applicable to normal and milestone task types.
   - **% Complete**: When specified, an additional inner bar is drawn to indicate the percentage complete for the task.
   - **Completed Through**: Allows you to specify a date rather than a percentage. When specified, an additional inner bar is drawn to indicate the percentage complete based on the date that the attribute references.

- **Critical**: Changes the color of the bar to red to mark it as critical. This is applicable to normal, summary, and milestone task types.
- **Edits Allowed**: Boolean value to specify if the task start date, end date, or duration is supported in the chart region.
- **Is Container**: Boolean value to specify if the node definition is a container.

5. If the data collection has an accessor for subtasks, use the **Subtasks** page to select the **Subtasks Accessor** from the dropdown list, and then the columns in the data collection that correspond to the following attributes:

   - **Subtask ID**: The unique Id for the list of subtasks in the data collection.
   - **Start Time**: The starting time of the subtask.
   - **End Time**: The ending time of the subtask.
   - **Subtask Type**: The type used to specify the appearance of the subtask bar when it is rendered in the Gantt chart. If you do not bind the type to a column in your data collection, then all tasks default to `Normal`. Task types include:
     - `Normal`: The basic task type, a plain horizontal bar that shows the start time, end time, and duration of the task.
     - `Summary`: Shows start time and end time for a group of tasks, typically subtasks. A summary task cannot be moved or extended. However, your application should recalculate the summary task times when the dates of any subtask changes.
     - `Milestone`: Shows a specific date in the Gantt chart. There is only one date associated with a milestone task. A milestone task cannot be extended but it can be moved.

   If you do not bind subtasks, then the Gantt chart cannot render a hierarchy view of tasks. If you bind subtasks, you can drill from tasks to subtasks in the hierarchy view of the Gantt chart.

6. If the data collection has an accessor for dependent tasks, use the Dependent Tasks page to select the **Dependent Task** accessor from the dropdown list, and then the columns in the data collection that correspond to the following attributes:

   - **Dependency Type**: Specifies the type of the dependency. Valid values are `start-start`, `start-finish`, `finish-finish`, `finish-start`, `start-before`, `start-together`, `finish-after`, and `finish-together`.
   - **From Task Id**: Specifies the first task in the dependent relationship.
   - **To Task Id**: Specifies the last task in the dependent relationship.

7. If the data collection has an accessor for split tasks, use the **Split Tasks** page to select the **Split Task** accessor from the dropdown list, and then the columns in the data collection that correspond to the following attributes:

   - **Split Task Id**: The unique Id for a task split into two horizontal bars, usually linked by a line. The time between the bars represents idle time due to traveling or down time.
   - **Start Time**: The starting time of the split task.
   - **End Time**: The ending time of the split task.

8. If the data collection has an accessor for recurring tasks, use the **Recurring Tasks** page to select the **Recurring Tasks** accessor from the dropdown list, and then the columns in the data collection that correspond to the following attributes:

- **Recurring Task Id**: A unique Id for a task repeated in a Gantt chart, each instance with its own start and end date. Individual recurring tasks can optionally contain a subtype. All other properties of the individual recurring tasks come from the task which they are part of. However, if an individual recurring task has a subtype, this subtype overrides the task type.
- **Type**: Specifies the type of the recurring task.
- **Start Time**: The starting time of the recurring task.
- **End Time**: The ending time of the recurring task.

9. In the **Appearance** page, specify the attributes that correspond to the **Label** of the task bar, and up to three icons to associate with the task bar.
10. In the **Table Columns** section, specify the columns to display for each row in the list region of the Gantt chart. Use the **New** icon to add new rows, the **Delete** icon to remove rows, and the arrow icons to arrange the rows in the sequence that you want the columns to appear in the Gantt chart list.

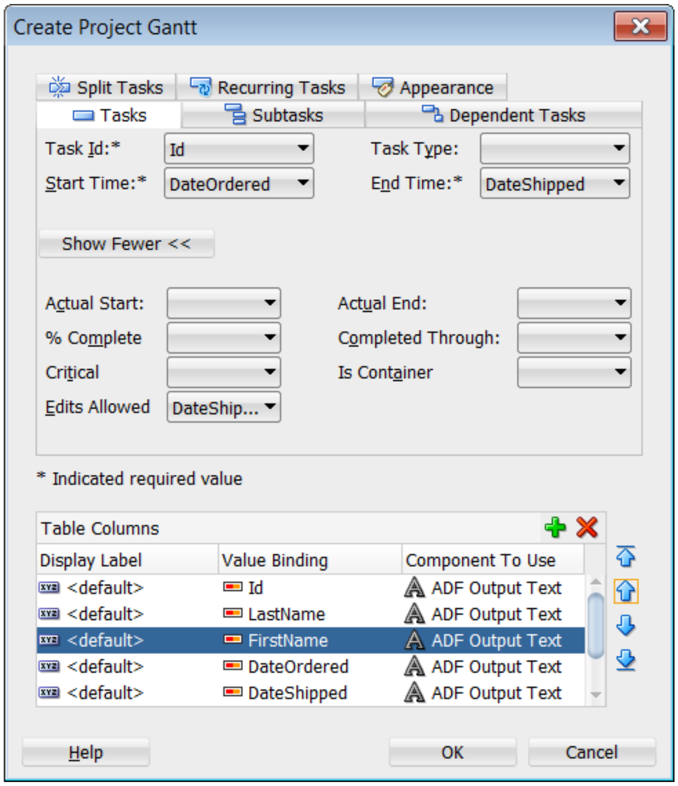
   > **Note:**
   >
   > The first row that you specify in the **Table Columns** section designates the `nodestamp` column for the list region. If you specify a subtask accessor, this row is used as the parent in a hierarchical relationship that can be expanded or collapsed.

   For each row, specify the following:

   - **Display Label**: Select the values for the headers of the columns in the Gantt chart list. If you select **<default>**, the text for the header is automatically retrieved from the data binding.
   - **Value Binding**: Select the columns in the data collection to use for the columns in the Gantt chart list. The available values are the same as those for the tasks group.
   - **Component to Use**: Select the type of component to display in the cell of the Gantt chart list. You can specify that a cell is editable by using an ADF input component. The default is the ADF Output Text component.
11. Click **OK**.

   Figure 42-3 shows the dialog used to create the project Gantt chart in Figure 42-1 from the data collection for shipping orders.

Figure 42-3 Create Project Gantt Dialog for Orders Shipped



Task Id:* Id

Task Type:

Start Time:* DateOrdered

End Time:* DateShipped

Actual Start:

Actual End:

% Complete

Completed Through:

Critical

Is Container

Edits Allowed DateShip...

* Indicated required value

Table Columns

Display Label	Value Binding	Component To Use
<default>	Id	ADF Output Text
<default>	LastName	ADF Output Text
<default>	FirstName	ADF Output Text
<default>	DateOrdered	ADF Output Text
<default>	DateShipped	ADF Output Text

12. In the Structure window, right-click the `dvt:projectGantt` component and select **Go to Properties**.

13. In the Properties window, expand the **Common** section, and specify the dates (`yyyy-mm-dd`) for the **StartTime** and **EndTime** attributes representing the start and end dates respectively for the time period of the Gantt chart data.

    The dates that you specify determine the initial view that appears in the project Gantt chart at runtime.

14. If you want to include a legend in the Gantt chart, right-click the project Gantt chart node in the Structure window and choose **Insert Inside Project Gantt** > **Legend**.

    The legend shows information about each symbol and color coded bar used to represent different task types. It also shows detailed information about the task selected in the Gantt chart.

After creating the project Gantt chart, use the Properties window to specify values for additional attributes. For detailed information about Gantt chart use cases, end user and presentation features, tag structure, and adding special features to Gantt charts, see the Using Gantt Chart Components chapter in *Developing Web User Interfaces with Oracle ADF Faces*.

What Happens When You Create a Project Gantt Chart from a Data Control

Dragging and dropping a view object from the Data Controls panel to create a project Gantt chart has the following effect:

- Creates the bindings for the Gantt chart and adds the bindings to the page definition file
- Adds the necessary code for the UI components to the JSF page

The example below displays the row set bindings generated for the project Gantt chart in Figure 42-1 to display shipped orders. This code example shows the node defined for tasks with attributes for task Id, start date, and end dates. There are also nodes defined for subtasks, dependent tasks, split tasks, and recurring tasks with no defined attributes.

```
<bindings>
  <gantt IterBinding="OrderEmployee1Iterator" id="OrderEmployee1"
         xmlns="http://xmlns.oracle.com/adfm/dvt">    <ganttDataMap>
<nodeDefinition DefName="model.OrderEmployee" type="Tasks">        <AttrNames>
          <Item Value="TaskId" type="taskId"/>
          <Item Value="StartDate" type="startTime"/>
          <Item Value="EndDate" type="endTime"/>
          <Item Value="TaskType" type="taskType"/>
        </AttrNames>
      </nodeDefinition>
      <nodeDefinition type="subTasks">
        <AttrNames/>
      </nodeDefinition>
      <nodeDefinition type="Dependents">
        <AttrNames/>
      </nodeDefinition>
      <nodeDefinition type="SplitTasks">
        <AttrNames/>
      </nodeDefinition>
      <nodeDefinition type="RecurringTasks">
        <AttrNames/>
      </nodeDefinition>
  </ganttDataMap>
 </gantt>
</bindings>
```

The example below shows the code generated on the JSF page for the project Gantt chart. This tag code contains settings for the overall start and end time for the project Gantt chart. It also shows the default time axis settings for the major axis in weeks and the minor axis in days. Finally, it lists the specifications for each column that appears in the list region of the Gantt chart.

```
<dvt:projectGantt id="gantt1"
value="#{bindings.OrderEmployee1.projectGanttModel}"
                  dataChangeListener="#{bindings.OrderEmployee1.projectGanttModel
                      .processDataChanged}"
                  var="row"
                  startTime="2012-03-31" endTime="2012-06-30">
  <f:facet name="major">
    <dvt:timeAxis scale="weeks" id="ta1"/>
  </f:facet>
  <f:facet name="minor">
    <dvt:timeAxis scale="days" id="ta2"/>
  </f:facet>
  <f:facet name="nodeStamp">
    <af:column
        sortProperty="#{bindings.OrderEmployee1.hints.FirstName.name}"
        sortable="false"
        headerText="#{bindings.OrderEmployee1.hints.FirstName.label}"
        id="c1">
        <af:outputText value="#{row.FirstName}" id="ot1"/>
      </af:column>
    </f:facet>
      <af:column sortProperty="LastName" sortable="false"
                 headerText="#{bindings.OrderEmployee1.hints.LastName.label}">
        <af:outputText value="#{row.LastName}"/>
      </af:column>
      <af:column sortProperty="OrderDate" sortable="false"
                 headerText="#{bindings.OrderEmployee1.hints.OrderDate.label}">
         <af:outputText value="#{row.OrderDate}">
           <af:convertDateTime
               pattern="#{bindings.OrderEmployee1.hints.OrderDate.format}"/>
         </af:outputText>
       </af:column>
       <af:column sortProperty="ShippedDate" sortable="false"
           headerText="#{bindings.OrderEmployee1.hints.ShippedDate.label}">
         <af:outputText value="#{row.ShippedDate}">
         <af:convertDateTime
           pattern="#{bindings.OrderEmployee1.hints.ShippedDate.format}"/>
         </af:outputText>
       </af:column>
       <af:column sortProperty="TaskType" sortable="false"
           headerText="#{bindings.OrderEmployee1.hints.TaskType.label}">
         <af:outputText value="#{row.TaskType}"/>
       </af:column>
   </projectGantt>
```

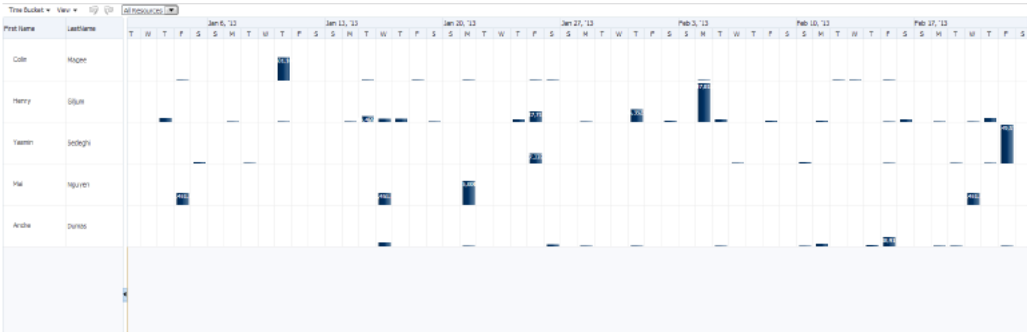
How to Create a Databound Resource Utilization Gantt Chart

For a resource utilization Gantt chart, you must supply identification for resources, identification for time, and start and end times for resource usage. Optionally, you can provide data values for subresources.

The resource utilization Gantt chart is displayed with default values for the major and minor time axis values. In a resource utilization Gantt chart, the setting for the major time axis defaults to weeks and the setting for the minor time axis defaults to days.

Figure 42-4 shows a resource utilization Gantt chart that lists each resource and an associated calendar that can display when the resource is in use.

Figure 42-4   Resource Utilization Gantt Chart



To create a resource utilization Gantt chart using a data control, you bind the resource utilization component to a data collection. JDeveloper allows you to do this declaratively by dragging a collection from the Data Controls panel and dropping it on a JSF page.

> **Tip:**
>
> You can also create a resource utilization Gantt chart by dragging a resource utilization Gantt chart component from the Components window and completing the Create Resource Utilization Gantt dialog. This approach gives you the option of designing the Gantt chart user interface before binding the component to data.

Before you begin:

It may be helpful to have an understanding of databound Gantt charts. For more information, see Creating Databound Gantt Charts.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Data Visualization Components.

You will need to complete these tasks:

- Create an application module that contains instances of the view objects that you want in your data model, as described in Creating and Modifying an Application Module.

  For example, the data source for the Gantt chart in Figure 42-4 comes from view objects in the Summit ADF DVT sample application. Use the `SalesRepViewObj1` view object representing the sales personnel with a view link to the `SOrdView2` view object that includes the time bucket accessor and attributes for order and ship dates.

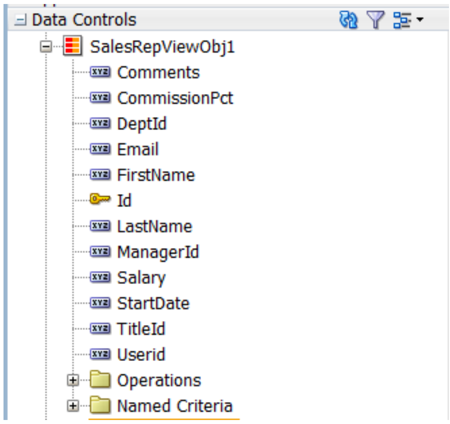
- Create a JSF page as described in the How to Create JSF Pages section of *Developing Web User Interfaces with Oracle ADF Faces*.

To create a resource utilization Gantt chart using the Data Controls panel:

1. From the Data Controls panel, select a data collection. For a Gantt chart, you can select a row set collection or a basic tree collection.

   Figure 42-5 shows an example where you could select the `SalesRepViewObj1` collection in the Data Controls panel to create a resource utilization Gantt chart to display the usage of a resource.

Figure 42-5 Data Collection for Resource Utilization Gantt Chart


Data Controls
SalesRepViewObj1
Comments
CommissionPct
DeptId
Email
FirstName
Id
LastName
ManagerId
Salary
StartDate
TitleId
Userid
Operations
Named Criteria

2. Drag the collection onto a JSF page and, from the context menu, choose **Gantt** > **Resource Utilization**.

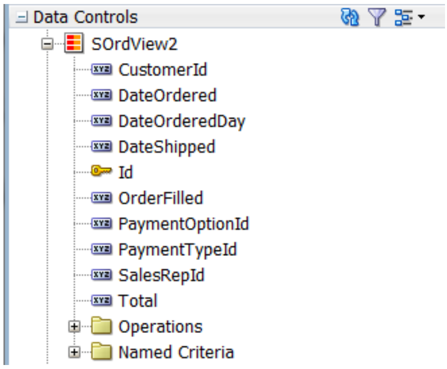
3. In the Create Resource Utilization Gantt dialog, select the columns in the data collection that correspond to the following attributes:

   - **Resource Id**: The unique identifier of the resource, for example `Id` representing the sales representatives in the view object.
   - **Is Container**: Boolean. Specify whether or not the node definition is a container.

4. In the **Time Buckets** page, select a value from the **Bucket Accessor** dropdown list that contains the time buckets assigned to the resource, and select a value from the **Bucket Date** dropdown list that corresponds to a unit of time.

   For example, the `SOrdView2` view object includes attributes representing the date ordered and date shipped for each order assigned to a sales representative. Figure 42-6 shows the view object used for the bucket accessor.

Figure 42-6 Resource Utilization Gantt Chart Bucket Accessor


Data Controls
SOrdView2
CustomerId
DateOrdered
DateOrderedDay
DateShipped
Id
OrderFilled
PaymentOptionId
PaymentTypeId
SalesRepId
Total
Operations
Named Criteria

5. In the **Bucket Metrics** list, specify attributes that appear as bars within the time bucket. Use the **New** icon to add new rows, the **Delete** icon to remove rows, and the arrow icons to arrange the rows in the sequence that you want the metrics to appear in the Gantt chart region.

   Each attribute that you specify in the **Bucket Metrics** list must be of type Number as the value of the attribute is used to calculate the height of the bar.

6. In the **Table Columns** section, specify the columns to display for each row in the list region of the Gantt chart. Use the **New** icon to add new rows, the **Delete** icon to remove rows, and the arrow icons to arrange the rows in the sequence that you want the columns to appear in the Gantt chart list.

   > **Note:**
   >
   > The first row that you specify in the **Table Columns** section designates the `nodestamp` column for the list region. If you specify a subresource accessor, this row displays the parent in a hierarchical relationship that can be expanded or collapsed.

   For each row, specify the following:

   - **Display Label**: Select the values for the headers of the columns in the Gantt chart list. If you select **<default>**, the text for the header is automatically retrieved from the data binding.
   - **Value Binding**: Select the columns in the data collection to use for the columns in the Gantt chart list. The available values are the same as those for the tasks group.
   - **Component to Use**: Select the type of component to display in the cell of the Gantt chart list. You can specify that a cell is editable by using an ADF input component. The default is the ADF Output Text component.

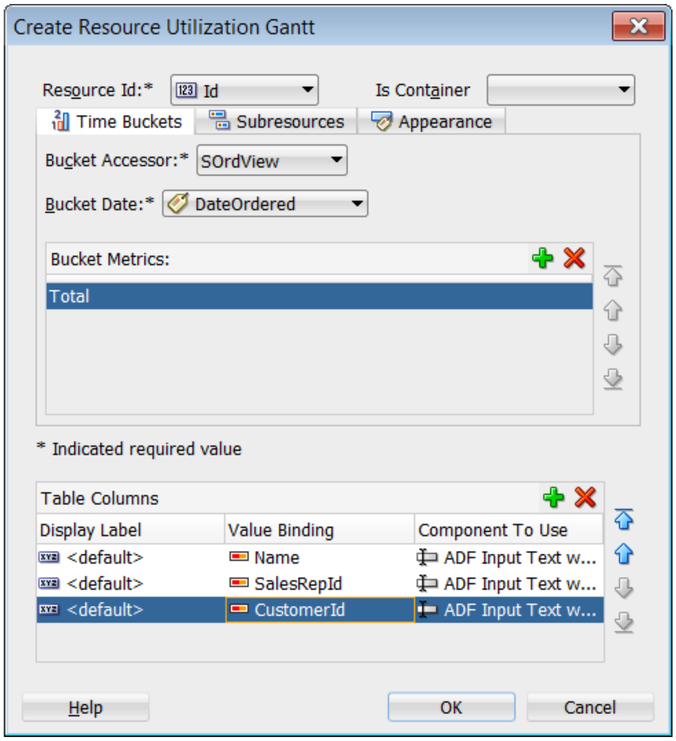
7. If the data collection has an accessor for subresources, use the **Subresources** page to select a **Subresources accessor** from the dropdown list, and then select a unique identifier for the subresource from the **Resource Id** dropdown list.

8. In the **Appearance** page, specify the attributes that correspond to the **Label** of the task bar.

9. Click **OK**.

   Figure 42-7 shows the dialog used to create a resource utilization Gantt chart from the data collection for resources available for a project.

Figure 42-7 Create Resource Utilization Gantt Chart Dialog



Create Resource Utilization Gantt

Resource Id:* Id

Is Container

Bucket Accessor:* SOrdView

Bucket Date:* DateOrdered

Bucket Metrics:

Total

* Indicated required value

Table Columns

Display Label	Value Binding	Component To Use
<default>	Name	ADF Input Text w...
<default>	SalesRepId	ADF Input Text w...
<default>	CustomerId	ADF Input Text w...

10. In the Structure window, right-click the `dvt:resourceUtilizationGantt` component and select **Go to Properties**.
11. In the Properties window, expand the **Common** section, and specify the dates (`yyyy-mm-dd`) for the **StartTime** and **EndTime** attributes representing the start and end dates respectively for the time period of the Gantt chart data.

    The dates that you specify determine the initial view that appears in the resource utilization Gantt chart at runtime.
12. If you want to include a legend in the Gantt chart, right-click the resource utilization Gantt chart node in the Structure window and choose **Insert Inside Resource Utilization Gantt** > **Legend**.

    The legend shows information about each symbol and color coded bar used to represent different tasks. It also shows detailed information about the task selected in the Gantt chart.

After creating the resource utilization Gantt chart, use the Properties window to specify values for additional attributes. For detailed information about Gantt chart use cases, end user and presentation features, tag structure, and adding special features to Gantt charts, see the Using Gantt Chart Components chapter in *Developing Web User Interfaces with Oracle ADF Faces*.

What Happens When You Create a Resource Utilization Gantt Chart

Dragging and dropping a view object from the Data Controls panel onto a JSF page to create a resource utilization Gantt chart has the following effects:

- Creates bindings for the resource utilization Gantt chart and adds the bindings to the page definition file

- Adds the necessary code for the UI components to the JSF page

The example below shows the row set bindings that were generated for the resource utilization Gantt chart illustrated in Figure 42-7.

```
<bindings>
  <gantt IterBinding="SalesRepViewObj1Iterator" id="SalesRepViewObj1"
         xmlns="http://xmlns.oracle.com/adfm/dvt">
    <ganttDataMap>
      <nodeDefinition DefName="model.SalesRepViewObj" type="Resources">
        <AttrNames>
          <Item Value="Id" type="resourceId"/>
        </AttrNames>
        <Accessors>
          <Item Value="SOrdView" type="timeBuckets"/>
        </Accessors>
      </nodeDefinition>
      <nodeDefinition type="TimeBuckets" DefName="model.SOrdView">
        <AttrNames>
          <Item Value="DateOrderedDay" type="time"/>
          <Item type="metric" Value="Total"/>
        </AttrNames>
      </nodeDefinition>
      <nodeDefinition type="Subresources">
        <AttrNames/>
      </nodeDefinition>
    </ganttDataMap>
  </gantt>
</bindings>
```

The example below shows the code generated on the JSF page for the resource utilization Gantt chart. This tag code contains settings for the overall start and end time for the resource utilization Gantt chart. These settings have to be edited manually. The code also shows the time axis settings for the major time axis (in weeks) and the minor time axis (in days). Finally, it lists the specifications for each column to appear in the list region of the resource utilization Gantt chart.

```
<dvt:resourceUtilizationGantt id="gantt1"
            value="#{bindings.SalesRepViewObj1.resourceUtilizationGanttModel}"
            var="row" metrics="#{bindings.SalesRepViewObj1.metrics}"
            taskbarFormatManager="#{bindings.SalesRepViewObj1.
                resourceUtilizationGanttTaskbarFormatManager}"
            startTime="2012-05-09" endTime="2013-05-15"
            summary="#{viewcontrollerBundle.RESOURCE_UTILIZATION_GANTT_CHART}">
  <f:facet name="major">
    <dvt:timeAxis scale="weeks" id="ta1"/>
  </f:facet>
  <f:facet name="minor">
    <dvt:timeAxis scale="days" id="ta2"/>
  </f:facet>
  <f:facet name="nodeStamp">
    <af:column sortProperty="#{bindings.SalesRepViewObj1.hints.FirstName.name}"
               sortable="false"
               headerText="#{bindings.SalesRepViewObj1.hints.FirstName.label}"
               id="c1">
      <af:outputText value="#{row.FirstName}"

shortDesc="#{bindings.SalesRepViewObj1.hints.FirstName.tooltip}"
                  id="ot1"/>
    </af:column>
  </f:facet>
```

```
    <af:column sortProperty="#{bindings.SalesRepViewObj1.hints.LastName.name}"
               sortable="false"
               headerText="#{bindings.SalesRepViewObj1.hints.LastName.label}"
               id="c2">
      <af:outputText value="#{row.LastName}"

shortDesc="#{bindings.SalesRepViewObj1.hints.LastName.tooltip}"
                   id="ot2"/>
    </af:column>
    <af:column sortProperty="#{bindings.SalesRepViewObj1.hints.Email.name}"
               sortable="false"
               headerText="#{bindings.SalesRepViewObj1.hints.Email.label}"
               id="c3">
      <af:outputText value="#{row.Email}"
                     shortDesc="#{bindings.SalesRepViewObj1.hints.Email.tooltip}"
                     id="ot3"/>
    </af:column>
</dvt:resourceUtilizationGantt>
```

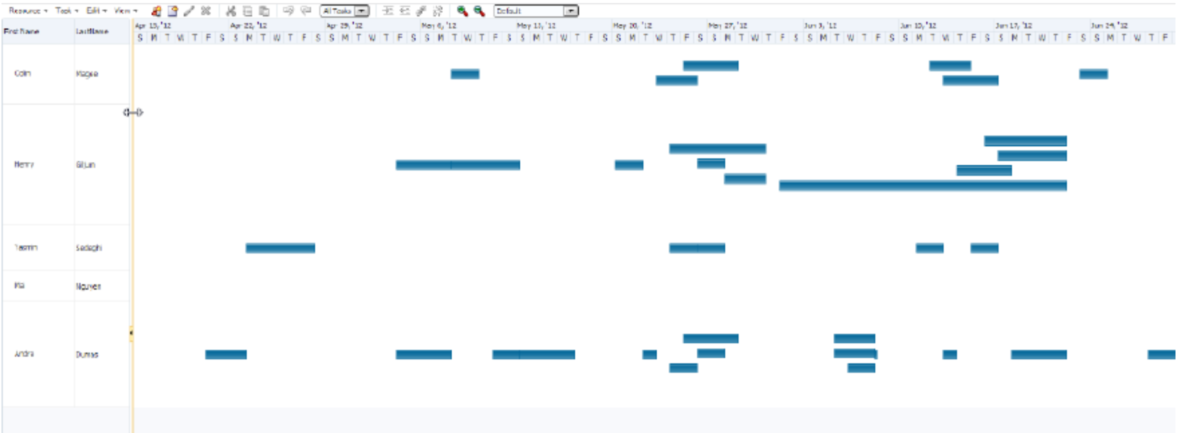
How to Create a Databound Scheduling Gantt Chart

For a scheduling Gantt chart, you must supply identification for resources, identification for tasks, and start and end times for tasks. Optionally, you can provide data values for subresources, recurring tasks, split tasks, and dependencies between tasks.

The scheduling Gantt chart is displayed with default values for overall start and end time and for the major and minor time axis values. In a scheduling Gantt chart, the setting for the major time axis defaults to weeks and the setting for the minor time axis defaults to days.

Figure 42-8 shows a scheduling Gantt chart that lists each resource and all the orders for which that resource is responsible. In contrast to a project Gantt chart, the scheduling Gantt chart shows all the tasks for a given resource on the same line, while the project Gantt chart lists each task on a separate line.

Figure 42-8   Scheduling Gantt Chart for Order Shipping



First Name	LastName
Colin	Magee
Henry	Gilun
Yasmin	Sadeghi
Mai	Nguyen
Andra	Dumas

To create a scheduling Gantt chart using a data control, you bind the `schedulingGantt` tag to a data collection. JDeveloper allows you to do this declaratively by dragging and dropping a collection from the Data Controls panel.

Before you begin:

It may be helpful to have an understanding of databound Gantt charts. For more information, see Creating Databound Gantt Charts.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Data Visualization Components.

You will need to complete these tasks:

- Create an application module that contains instances of the view objects that you want in your data model, as described in Creating and Modifying an Application Module.

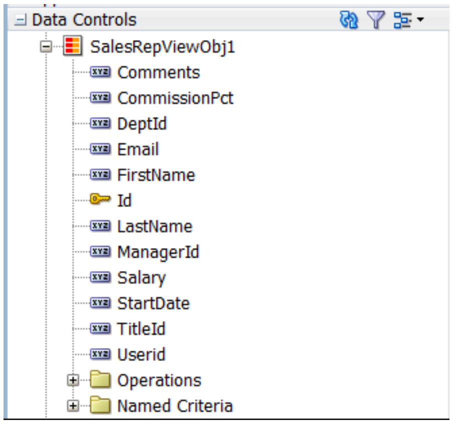
  For example, the data source for the Gantt chart in Figure 42-8 comes from view objects in the Summit ADF DVT sample application. Use the `SalesRepViewObj1` view object representing the sales personnel with a view link to the `SOrdView2` view object that includes the tasks accessor and attributes for order and ship dates.
- Create a JSF page as described in the How to Create JSF Pages section of *Developing Web User Interfaces with Oracle ADF Faces*.

To create a scheduling Gantt chart using the Data Controls panel:

1. From the Data Controls panel, select a data collection. For a Gantt chart, you can select a row set collection or a basic tree collection.

   Figure 42-9 shows an example where you could select the `SalesRepViewObj1` data collection to create a scheduling Gantt chart that displays the orders each sales representative is responsible for managing.

Figure 42-9 Data Collection for Scheduling Gantt Chart



2. Drag the collection onto a JSF page and, from the context menu, choose **Gantt** > **Scheduling**.
3. In the Create Scheduling Gantt dialog, select the columns in the data collection that correspond to the following attributes:
   - **Resource Id**: The unique Id for the resource represented in the scheduling Gantt chart.
   - **Working Start Time**: The work start date of the scheduling Gantt chart.

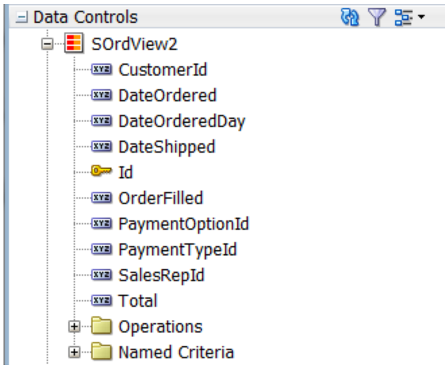
- **Working End Time**: The work end date of the scheduling Gantt chart.
- **Working Days of Week**: Specifies a list of the working days of the week.
- **Is Container**: Boolean. Specify whether or not the node definition is a container.

4. In the **Tasks** page, select the **Task Accessor** from the dropdown list, and then select the columns in the data collection that correspond to the following attributes:

   - **Task Id**: The unique Id for the task associated with a resource in the Gantt chart.
   - **Task Type**: The type used to specify the appearance of the task bar when it is rendered in the Gantt chart. If you do not bind the type to a column in your data collection, then all tasks default to `Scheduled`.
   - **Start Time**: The starting time of the task.
   - **End Time**: The ending time of the task.
   - **Startup Time**: The time to start up the task before beginning.
   - **Shutdown Time**: The time to shut down the task after completion.

   For example, the `SOrdView2` view object includes attributes representing the date ordered and date shipped for each order assigned to a sales representative. Figure 42-10 shows the view object used for the task accessor.

   Figure 42-10 Scheduling Gantt Chart Task Accessor

5. If the data collection has an accessor for dependent tasks, use the **Dependent Tasks** page to select the **Dependent Task** accessor, and then select the columns in the data collection that correspond to the following attributes:

   - **Dependency Type**: Specifies the type of the dependency. Valid values are `start-start`, `start-finish`, `finish-finish`, `finish-start`, `start-before`, `start-together`, `finish-after`, and `finish-together`.
   - **From Task Id**: Specifies the first task in the dependent relationship.
   - **To Task Id**: Specifies the last task in the dependent relationship.

   Dependent tasks are linked by their associations with the finish and start times between them.

6. If the data collection has an accessor for split tasks, use the **Split Tasks** page to select the **Split Tasks** accessor, and then select the columns in the data collection that correspond to the following attributes:

- **Split Task Id**: The unique Id for a task split into two horizontal bars, usually linked by a line. The time between the bars represents idle time due to traveling or down time.
- **Start Time**: The starting time of the split task.
- **End Time**: The ending time of the split task.

7. If the data collection has an accessor for recurring tasks, use the **Recurring Tasks** page to select the **Recurring Tasks** accessor from the dropdown list, and then select the columns in the data collection that correspond to the following attributes:

   - **Recurring Task Id**: A unique Id for a task repeated in a Gantt chart, each instance with its own start and end date. Individual recurring tasks can optionally contain a subtype. All other properties of the individual recurring tasks come from the task which they are part of. However, if an individual recurring task has a subtype, this subtype overrides the task type.
   - **Type**: Specifies the type of the recurring task.
   - **Start Time**: The starting time of the recurring task.
   - **End Time**: The ending time of the recurring task.

8. If the data collection has an accessor for subresources, use the **Subresources** page to select the **Subresources Accessor** from the dropdown list, and then select the column in the data collection that corresponds to the **Subresource Id**, the unique Id for the list of subresources in the data collection.

   If you do not bind subresources, then the Gantt chart cannot render a hierarchy view of resources. If you bind subresources, then you can drill from resources to subresources in the hierarchy view of the Gantt chart.

9. In the **Appearance** page, specify the attributes that correspond to the **Label** of the task bar, and up to three icons to associate with the task bar.

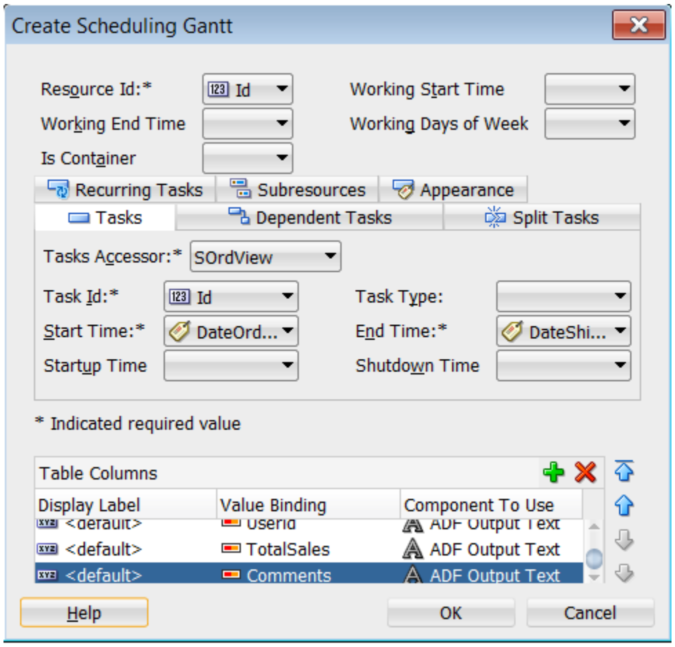
10. In the **Table Columns** section, specify the columns that will appear in the list region of the Gantt chart on the left side of the splitter. Specify one row of information for each column that is to appear. Use the **New** icon to add new rows. Use the arrow icon to arrange the rows in the exact sequence that you want the columns to appear in the Gantt chart list. For each row, you provide the following specifications:

    - **Display Label:** Select the values for the headers of the columns in the Gantt chart list. If you select `<default>`, then the text for the header is automatically retrieved from the data binding.
    - **Value Binding:** Select the columns in the data collection to use for the column in the Gantt chart list. The available values are the same as those for the tasks group.
    - **Component to Use:** Select the type of component to display in the cell of the Gantt chart list. The default is the **ADF Output Text** component.

11. Click **OK**.

    Figure 42-11 shows the dialog used to create the scheduling Gantt chart from the data collection for sales representatives responsible for shipping orders.

Figure 42-11 Create Scheduling Gantt Dialog



Resource Id:* Id
Working Start Time
Working End Time
Working Days of Week
Is Container

Tasks Accessor:* SOrdView
Task Id:* Id
Task Type:
Start Time:* DateOrd...
End Time:* DateShi...
Startup Time
Shutdown Time

* Indicated required value

Table Columns

Display Label	Value Binding	Component To Use
<default>	UserId	ADF Output Text
<default>	TotalSales	ADF Output Text
<default>	Comments	ADF Output Text

12. In the Structure window, right-click the `dvt:schedulingGantt` component and select **Go to Properties**.
13. In the Properties window, expand the **Common** section, and specify the dates (`yyyy-mm-dd`) for the **StartTime** and **EndTime** attributes representing the start and end dates respectively for the time period of the Gantt chart data.

    The dates that you specify determine the initial view that appears in the scheduling Gantt chart at runtime.
14. If you want to include a legend in the Gantt chart, right-click the scheduling Gantt chart node in the Structure window and choose **Insert Inside Scheduling Gantt** > **Legend**.

    The legend shows information about each symbol and color coded bar used to represent different tasks. It also shows detailed information about the task selected in the Gantt chart.

What Happens When You Create a Scheduling Gantt Chart

Dropping a scheduling Gantt chart from the Data Controls panel has the following effect:

- Creates the bindings for the Gantt chart and adds the bindings to the page definition file
- Adds the necessary code for the UI components to the JSF page

The example below shows the row set bindings that were generated for the scheduling Gantt chart that displays resources and orders shipped.

```
<bindings>
  <gantt IterBinding="SalesRepViewObj1Iterator" id="SalesRepViewObj1"
         xmlns="http://xmlns.oracle.com/adfm/dvt">
    <ganttDataMap>
      <nodeDefinition DefName="model.SalesRepViewObj" type="Resources">
```

```
          <AttrNames>
            <Item Value="Id" type="resourceId"/>
          </AttrNames>
          <Accessors>
            <Item Value="SOrdView" type="tasks"/>
          </Accessors>
        </nodeDefinition>
        <nodeDefinition type="Tasks" DefName="model.SOrdView">
          <AttrNames>
            <Item Value="Id" type="taskId"/>
            <Item Value="DateOrdered" type="startTime"/>
            <Item Value="DateShipped" type="endTime"/>
          </AttrNames>
        </nodeDefinition>
        <nodeDefinition type="Dependents">
          <AttrNames/>
        </nodeDefinition>
        <nodeDefinition type="SplitTasks">
          <AttrNames/>
        </nodeDefinition>
        <nodeDefinition type="RecurringTasks">
          <AttrNames/>
        </nodeDefinition>
        <nodeDefinition type="Subresources">
          <AttrNames/>
        </nodeDefinition>
      </ganttDataMap>
    </gantt>
  </bindings>
```

The example below shows the code generated on the JSF page for the scheduling Gantt chart. This tag code contains settings for the overall start and end time for the scheduling Gantt chart. It also shows the time axis settings for the major time axis in weeks and the minor time axis in days. Finally, it lists the specifications for each column that appears in the list region of the Gantt chart.

```
<dvt:schedulingGantt id="gantt1"
                     value="#{bindings.SalesRepViewObj1.schedulingGanttModel}"
                     dataChangeListener="#{bindings.SalesRepViewObj1.
                         schedulingGanttModel.processDataChanged}"
                     var="row" startTime="2012-04-15" endTime="2013-05-15"
                     summary="#{viewcontrollerBundle.SCHEDULING_GANTT_CHART}">
  <f:facet name="major">
    <dvt:timeAxis scale="weeks" id="ta1"/>
  </f:facet>
  <f:facet name="minor">
    <dvt:timeAxis scale="days" id="ta2"/>
  </f:facet>
<f:facet name="nodeStamp">
  <af:column sortProperty="#{bindings.SalesRepViewObj1.hints.FirstName.name}"
             sortable="false"
             headerText="#{bindings.SalesRepViewObj1.hints.FirstName.label}"
             id="c1">
    <af:outputText value="#{row.FirstName}"
                   shortDesc="#{bindings.SalesRepViewObj1.hints.
                       FirstName.tooltip}"
                   id="ot1"/>
  </af:column>
</f:facet>
  <af:column sortProperty="#{bindings.SalesRepViewObj1.hints.LastName.name}"
             sortable="false"
```

```
           headerText="#{bindings.SalesRepViewObj1.hints.LastName.label}"
           id="c2">
      <af:outputText value="#{row.LastName}"

shortDesc="#{bindings.SalesRepViewObj1.hints.LastName.tooltip}"
                  id="ot2"/>
    </af:column>
    <af:column sortProperty="#{bindings.SalesRepViewObj1.hints.Email.name}"
               sortable="false"
               headerText="#{bindings.SalesRepViewObj1.hints.Email.label}" id="c3">
      <af:outputText value="#{row.Email}"
                     shortDesc="#{bindings.SalesRepViewObj1.hints.Email.tooltip}"
                     id="ot3"/>
    </af:column>
</dvt:schedulingGantt>
```

What You May Need to Know About Data Change Event Handling

When you use ADF data controls to data bind project and scheduling Gantt charts, a data change listener is automatically provided for data change events that require an update to row attributes, for example:

- DURATION_CHANGE: End time of a task; resize task bar
- PROGRESS_CHANGE: Progress of task; resize progress bar
- TIME_CHANGE: Start and end time of a task; move the task bar

At design time, the data change listener provides a built-in processDataChange method for project and scheduling Gantt bindings as illustrated in the example below.

```
<dvt:projectGantt id="gantt1"
                  dataChangeListener="#{bindings.GanttProjectView1.
                      projectGanttModel.processDataChanged}"
                  value="#{bindings.GanttProjectView1.projectGanttModel}"
                  ...
</dvt:projectGantt>
```

Data change events requiring row insertion or deletion are not supported, for example:

- TASK_SPLITTED: Split a task
- TASK_MERGED: Merge task with another task
- LINK: Link two tasks together
- UNLINK: Unlink two tasks linked together
- COPY: Copy one or more tasks or resources
- CREATE: Create a new task
- CREATE_RESOURCE: Create a new resource
- CUT: Cut one or more tasks or resources
- PASTE: Paste tasks or resources
- UPDATE: Update a task
- UPDATE_RESOURCE: Update a resource
- TASKS_PROPERTIES_UPDATE: Task properties dialog event
- INDENT: Indent a task or resource

- OUTDENT: Outdent a task or resource
- DELETE: Delete one or more tasks or resources

If you wish to configure your application to handle events that do not have a default implementation, specify a method in a backing bean and delegate to the built-in processDataChange method. For example, the first block of code below shows a dataChangeListener configured to use a handleDataChange method to handle a TASK_PROPERTIES_UPDATE event using the backing bean shown in the class code below.

```
<dvt:projectGantt id="gantt1"
                  dataChangeListener="#{backingBeanScope.backing_
                      projectGantt.handleDataChange}"
                  value="#{bindings.GanttProjectView1.projectGanttModel}"
                  ...
</dvt:projectGantt>
```

```
public void handleDataChange(DataChangeEvent evt)
{
    // handle events not supported by built-in "processDataChange" method
    int _type = evt.getActionType();
    if (_type == DataChangeEvent.TASK_PROPERTIES_UPDATE)
    {
        // handle event.
    }
    else  // delegate to built-in "processDataChange" method.
    {
                Application app = facesContext.getApplication();
        String expression =
            "#{bindings.GanttProjectView1.projectGanttModel.processDataChanged}";
        ExpressionFactory elFactory = app.getExpressionFactory();
        ELContext elContext = facesContext.getELContext();
        MethodExpression methodExp = elFactory.createMethodExpression(elContext,
            expression, null, new Class[] {DataChangeEvent.class});
        methodExp.invoke(elContext, new Object[] {evt});
    }
}
```

Note:

The project and scheduling Gantt chart's default implementation of a dataChangeListener for ADF data controls will not perform commit or rollback changes to the database. You must configure your application to handle those operations.

Creating Databound Timelines

A timeline is an interactive data visualization tool that allows users to view date-based events in chronological order and easily navigate forwards and backwards within a defined time range.

A timeline is composed of the display of events as timeline items along a time axis, a movable overview window that corresponds to the period of viewable time in the timeline, and an overview time axis that displays the total time increment for the timeline. A horizontal zoom control is available to change the viewable time range. Timeline items corresponding to events display associated information or actions and are connected to the date of the event in the time axis. Timelines items are represented by a marker in the overview panel. No more that two series of events are supported by the timeline component.

A dual timeline can be used for a side-by-side comparison of events. For example, you can use a timeline to display the hire dates of employees, or use a dual timeline to compare multiple human resource events between two employees.

Figure 42-12 shows a timeline in the Summit ADF DVT sample application. In this example, two lines of text are displayed along with images. When selection is configured, the timeline item, line feeler, and the event marker in the overview panel are highlighted.

Figure 42-12 Sample Timeline

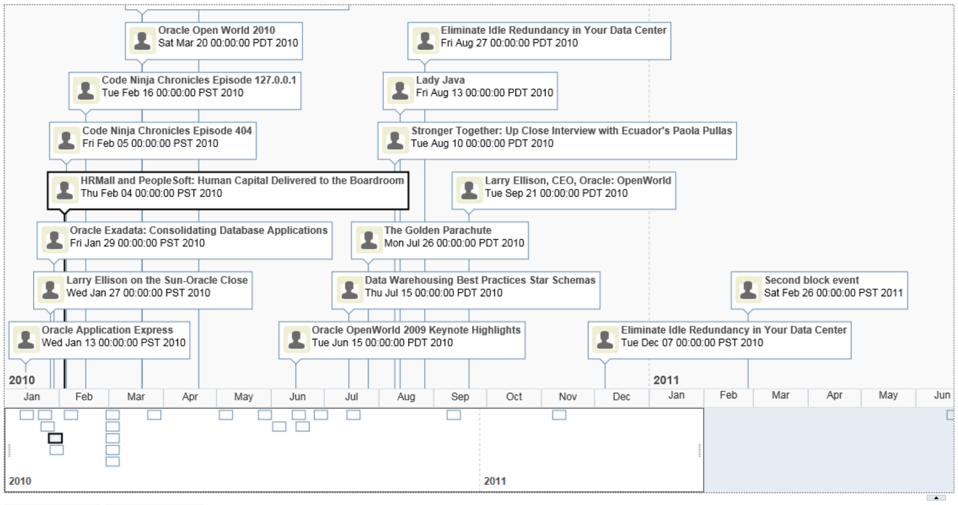
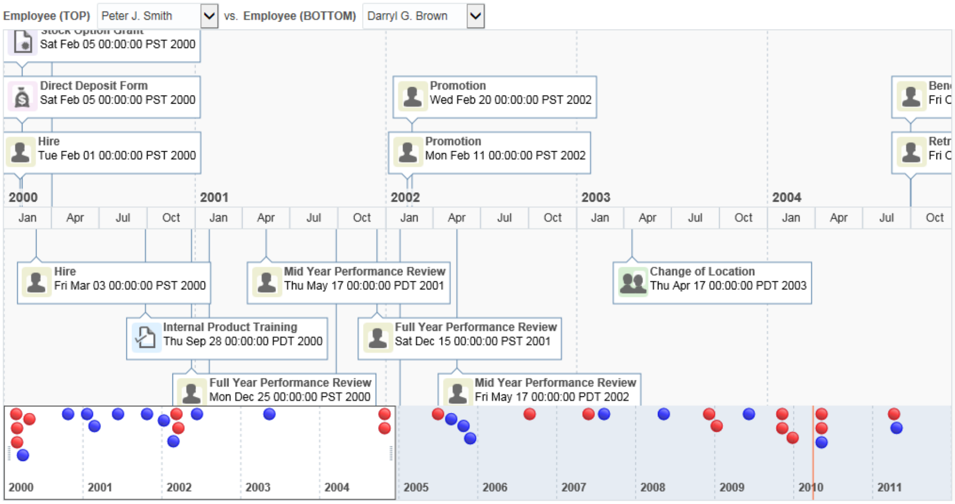

Oracle Open World 2010
Sat Mar 20 00:00:00 PDT 2010
Eliminate Idle Redundancy in Your Data Center
Fri Aug 27 00:00:00 PDT 2010
Code Ninja Chronicles Episode 127.0.0.1
Tue Feb 16 00:00:00 PST 2010
Lady Java
Fri Aug 13 00:00:00 PDT 2010
Code Ninja Chronicles Episode 404
Fri Feb 05 00:00:00 PST 2010
Stronger Together: Up Close Interview with Ecuador's Paola Pullas
Tue Aug 10 00:00:00 PDT 2010
HRMall and PeopleSoft: Human Capital Delivered to the Boardroom
Thu Feb 04 00:00:00 PST 2010
Larry Ellison, CEO, Oracle: OpenWorld
Tue Sep 21 00:00:00 PDT 2010
Oracle Exadata: Consolidating Database Applications
Fri Jan 29 00:00:00 PST 2010
The Golden Parachute
Mon Jul 26 00:00:00 PDT 2010
Larry Ellison on the Sun-Oracle Close
Wed Jan 27 00:00:00 PST 2010
Data Warehousing Best Practices Star Schemas
Thu Jul 15 00:00:00 PDT 2010
Second block event
Sat Feb 26 00:00:00 PST 2011
Oracle Application Express
Wed Jan 13 00:00:00 PST 2010
Oracle OpenWorld 2009 Keynote Highlights
Tue Jun 15 00:00:00 PDT 2010
Eliminate Idle Redundancy in Your Data Center
Tue Dec 07 00:00:00 PST 2010
2010
2011
Jan
Feb
Mar
Apr
May
Jun
Jul
Aug
Sep
Oct
Nov
Dec
Jan
Feb
Mar
Apr
May
Jun
2010
2011

Figure 42-13 shows a dual timeline comparing multiple human resource events between two employees. The time axis is positioned between the two series of events and the overview panel displays at the bottom of the timeline.

Figure 42-13 Dual Timeline Comparing Human Resource Events


Employee (TOP) Peter J. Smith vs. Employee (BOTTOM) Darryl G. Brown
Stock Option Grant
Sat Feb 05 00:00:00 PST 2000
Direct Deposit Form
Sat Feb 05 00:00:00 PST 2000
Hire
Tue Feb 01 00:00:00 PST 2000
Promotion
Wed Feb 20 00:00:00 PST 2002
Promotion
Mon Feb 11 00:00:00 PST 2002
Bene
Fri C
Retr
Fri C
2000
2001
2002
2003
2004
Jan Apr Jul Oct Jan Apr Jul Oct Jan Apr Jul Oct Jan Apr Jul Oct Jan Apr Jul Oct
Hire
Fri Mar 03 00:00:00 PST 2000
Mid Year Performance Review
Thu May 17 00:00:00 PDT 2001
Change of Location
Thu Apr 17 00:00:00 PDT 2003
Internal Product Training
Thu Sep 28 00:00:00 PDT 2000
Full Year Performance Review
Sat Dec 15 00:00:00 PST 2001
Full Year Performance Review
Mon Dec 25 00:00:00 PST 2000
Mid Year Performance Review
Fri May 17 00:00:00 PDT 2002
2000 2001 2002 2003 2004 2005 2006 2007 2008 2009 2010 2011

The content of timeline items, marker display, and time axis are configurable. For detailed information about timeline use cases, end user and presentation features, tag structure, and adding special features to timelines, see the Using Timeline Components chapter in *Developing Web User Interfaces with Oracle ADF Faces*.

How to Create a Timeline Using ADF Data Controls

The data displayed in a timeline is based on data collections. The `timeline` component uses a model to access the data in the underlying list. The specific model class is `oracle.adf.view.rich.model.CollectionModel`. You can also use other model instances, for example, `java.util.List`, `java.util.Array`, and `javax.faces.model.DataModel`. The data layer will automatically convert the instance into a `CollectionModel`.

Using data controls in Oracle ADF, JDeveloper makes this a declarative task. You drag and drop a data collection from the Data Controls panel that generates one (up to two) series of events onto the JSF page.

Before you begin:

It may be helpful to have an understanding of databound timelines. See Creating Databound Timelines.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. See Additional Functionality for Data Visualization Components.

You will need to complete these tasks:

- Create an application module that contains instances of the view objects that you want in your data model, as described in Creating and Modifying an Application Module.

  For example, the data source for the timeline in Figure 42-12 comes from a single table in the Summit ADF DVT sample application. Use the `SEmp` table to create a view object representing the employees including their hire dates.

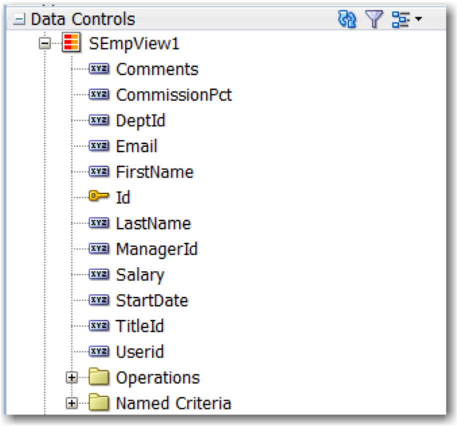
- Create a JSF page as described in the How to Create JSF Pages section of *Developing Web User Interfaces with Oracle ADF Faces*.

To create a timeline using the Data Controls panel:

1. From the Data Controls panel, select a collection.

   Figure 42-14 shows an example where you could select the `SEmpView1` collection in the Data Controls panel to create a timeline that displays the hire dates of employees.

   Figure 42-14   Data Collection for Timeline of Employee Hire Dates

2. Drag the collection onto the JSF page, and from the context menu choose **Timeline**.

3. In the Create Timeline dialog, configure the time axis and overview axis by setting the following values:

   a. **Start Time**: Enter the starting date to use for the time range of the timeline using the format `yyyy-mm-dd`. Select a start date that will include events in the data collection you wish to display on the timeline. From the dropdown list you can select `Current Date`. You can also click the Calendar icon to display a date picker dialog. The default value is set for three months before the current date.

   b. **End Time**: Enter the ending date to use for the timeline time range using the format `yyyy-mm-dd`. Select an end date that will include events in the data collection you wish to display on the timeline. From the dropdown list you can select `Current Date`. You can also click the Calendar icon to display a date picker dialog. The default value is `Current Date`.

   c. **Axis Scale**: Use the dropdown list to select the time scale to use for the timeline time axis. The time axis shows the user the current time increment for the timeline.

      You must use a smaller time scale than the **Overview Scale**. For example, you cannot set the axis scale to `years` if the overview scale is set to `months`. You can also specify a custom axis scale.

      Valid values are `twoyears`, `years`, `halfyears`, `quarters` (default), `twomonths`, `months`, `twoweeks`, `weeks`, `days`, `sixhours`, `threehours`, `hours`, `halfhours`, `quarterhours`. You can also specify a custom axis scale. See How to Add a Custom Time Scale to a Timeline in *Developing Web User Interfaces with Oracle ADF Faces*.

d. **Overview Scale**: Use the dropdown list to select the time scale to use for the timeline overview axis. The overview axis shows the user the total time increment covered by the timeline.

   Valid values are `twoyears`, `years` (default), `halfyears`, `quarters`, `twomonths`, `months`, `twoweeks`, `weeks`, `days`, `sixhours`,`threehours`, `hours`, `halfhours`, `quarterhours`. You can also specify a custom axis scale. See How to Add a Custom Time Scale to a Timeline in *Developing Web User Interfaces with Oracle ADF Faces*.

4. In the Create Timeline dialog, configure the timeline data model for at least one (at most two for a dual timeline), by setting the following values:

   a. **Series Value**: Use the dropdown list to select the data collection to use in the timeline. By default, the data collection you inserted from the Data Controls panel is displayed. JDeveloper automatically lists any collection from the application module that includes at least one qualifying date-based attribute from which to choose.

   b. **Item Date Value**: Use the dropdown list to select the date-based attribute to use for the timeline items stamped in the timeline. JDeveloper automatically lists all qualifying attributes in the collection referenced in the **Series Value** from which to choose.

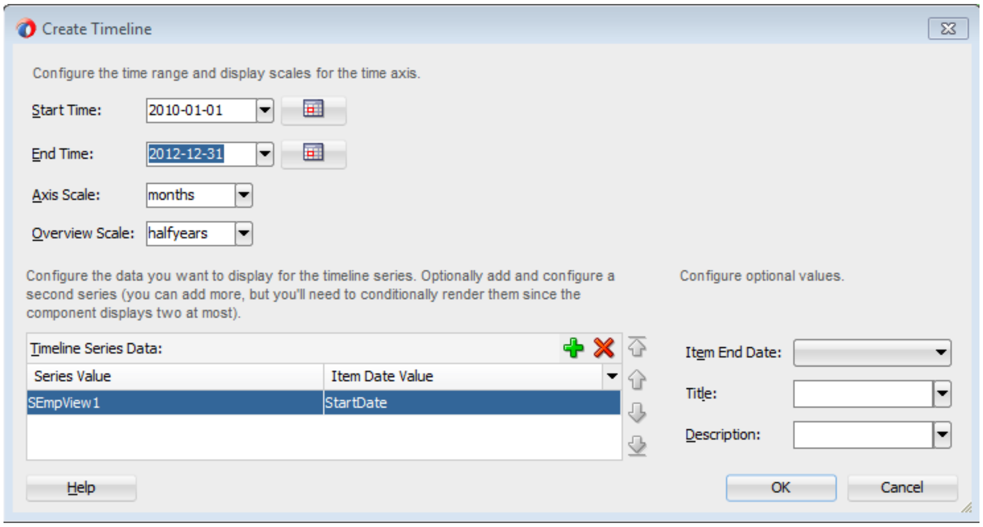
> **Note:**
>
> You can add and configure more series, but you will need to conditionally render them since the component displays two at most.

5. You can also configure the following values that are optional:

   - **Item End Date**: Specify the time-related attribute in the data collection that represents the end date for the timeline item. Use the dropdown list to display a list of available choices.
   - **Title**: Enter text or use the dropdown to specify the attribute in the data collection that represents the title for the timeline item.
   - **Description**: Enter text or use the dropdown to specify the attribute in the data collection that represents the description for the timeline item.

   For example, in Figure 42-15 you can specify all the values.

Figure 42-15 Create Timeline Dialog



Create Timeline

Configure the time range and display scales for the time axis.

Start Time: 2010-01-01

End Time: 2012-12-31

Axis Scale: months

Overview Scale: halfyears

Configure the data you want to display for the timeline series. Optionally add and configure a second series (you can add more, but you'll need to conditionally render them since the component displays two at most).

Timeline Series Data:

Series Value	Item Date Value
SEmpView1	StartDate

Configure optional values.

Item End Date:

Title:

Description:

6. Click **OK** to complete the definition of the data binding, or **Cancel** to end the creation of the timeline.

What Happens When You Use Data Controls to Create a Timeline

When you use ADF data controls to create a timeline, JDeveloper:

- Defines the bindings for the timeline in the page definition file of the JSF page, and
- Inserts code in the JSF page for the DVT timeline components.

The example below shows the bindings defined in the page definition file of the JSF page for the Summit ADF DVT sample timeline in Figure 42-12.

```
<bindings>
  <tree IterBinding="SEmpView1Iterator" id="SEmpView1" ChangeEventPolicy="ppr">
    <nodeDefinition DefName="model.SEmpView" Name="SEmpView1">
      <AttrNames>
        <Item Value="DeptId"/>
        <Item Value="FirstName"/>
        <Item Value="Id"/>
        <Item Value="LastName"/>
        <Item Value="StartDate"/>
      </AttrNames>
    </nodeDefinition>
  </tree>
</bindings>
```

The example below shows the code inserted in the JSF page for the example timeline in Figure 42-12.

```
<dvt:timeline id="tl1" startTime="2010-01-01" endTime="2011-12-31"
itemSelection="multiple"
              binding="#{extEditor.component}" summary="Timeline Component Demo">
    <dvt:timeSeries id="ts1" var="evt" value="#{timeline.model}">
        <dvt:timeItem id="ti1" value="#{evt.date}" title="#{evt.description}"
description="#{evt.date}"
                      thumbnail="/resources/images/timeline/employment.png"/>
    </dvt:timeSeries>
    <dvt:timeAxis id="ta1" scale="weeks" zoomOrder="quarters months weeks days"/>
    <dvt:timelineOverview id="ov1">
        <dvt:timeAxis id="ta2" scale="years"/>
    </dvt:timelineOverview>
</dvt:timeline>
```

What You May Need to Know About Using Data Controls to Create a Dual Timeline

When you use the Data Controls panel to create a dual timeline, you can configure the same or different data collections to configure the two timeline series and specify the timeline item display attributes and overview marker for each one. JDeveloper automatically lists any data collection from the application module that includes at least one qualifying date-based attribute from which to choose in the Create Timeline wizard.

The example below shows the sample code for the dual timeline displayed in Figure 42-13.

```
<dvt:timeline id="tl1" startTime="2000-01-01" endTime="2011-12-31"
                       inlineStyle="width:1024px;height:500px"
itemSelection="single" currentTime="2010-04-01">
    <dvt:timeSeries id="ts1" var="evt" value="#{timeline.firstModel}">
        <dvt:timeItem id="ti1" value="#{evt.date}" group="#{evt.group}"
title="#{evt.description}"
                      description="#{evt.date}" thumbnail="/resources/images/
timeline/#{evt.type}.png">
            <f:facet name="overviewItem">
                <dvt:marker id="m1" shape="circle" fillColor="#ff0000"/>
            </f:facet>
        </dvt:timeItem>
    </dvt:timeSeries>
    <dvt:timeSeries id="ts2" var="evt" value="#{timeline.secondModel}">
        <dvt:timeItem id="ti2" value="#{evt.date}" title="#{evt.description}"
                      description="#{evt.date}" thumbnail="/resources/images/
timeline/#{evt.type}.png">
            <f:facet name="overviewItem">
                <dvt:marker id="m2" shape="circle" fillColor="#0000ff"/>
            </f:facet>
        </dvt:timeItem>
   </dvt:timeSeries>
   <dvt:timeAxis id="ta1" scale="quarters" zoomOrder="quarters months weeks
days"/>
   <dvt:timelineOverview id="ov1">
       <dvt:timeAxis id="ta2" scale="years"/>
   </dvt:timelineOverview>
</dvt:timeline>
```

For information about the data requirements, tag structure, and options for customizing the look and behavior of the component, see the Using Timeline Components chapter in the *Developing Web User Interfaces with Oracle ADF Faces*.

43

Creating Databound Hierarchy Viewer, Treemap, and Sunburst Components

This chapter describes how to create a hierarchy viewer, treemap, or sunburst from data modeled with ADF Business Components, using ADF data controls and ADF Faces components in a Fusion web application. Specifically, it describes how you can use ADF Data Visualization `hierarchyViewer`, `treemap`, and `sunburst` components to create hierarchy viewers, treemaps, and sunbursts that visually represent business data. It describes how to use ADF data controls to create these components with data-first development.

If you are designing your page using simple UI-first development, then you can add the hierarchy viewer, treemap, or sunburst to your page and configure the data bindings later. For information about the data requirements, tag structure, and options for customizing the look and behavior of the hierarchy viewer, treemap, and sunburst components, see the Using Hierarchy Viewer Components and Using Treemap and Sunburst Components chapters in *Developing Web User Interfaces with Oracle ADF Faces*.

This chapter includes the following sections:

About ADF Data Visualization Components

ADF Data Visualization components provide extensive graphical and tabular capabilities for visually displaying and analyzing business data. Each component needs to be bound to data before it can be rendered since the appearance of the components is dictated by the data that is displayed.

The ADF hierarchy viewer component produces an interactive graphic that displays hierarchical data as a set of linked shapes. The shapes and links correspond to the elements and relationships in the data. For example, a hierarchy viewer component might be used to generate an organizational chart based on employee data. At runtime, end users can pan and zoom the graphic and expand, select, and navigate the management hierarchy that the graphic displays.

Use treemaps and sunbursts to display quantitative hierarchical data across two dimensions, represented visually by size and color. Treemaps and sunbursts use a shape called a node to reference the data in the hierarchy. For example, you can use a treemap or sunburst to display quarterly regional sales and to identify sales trends, using the size of the node to indicate each region's sales volume and the node's color to indicate whether that region's sales increased or decreased over the quarter. The appearance and content of the nodes are configurable at each level of the hierarchy.

The prefix `dvt:` occurs at the beginning of each data visualization component name indicating that the component belongs to the ADF Data Visualization Tools (DVT) tag library.

End User and Presentation Features

Visually compelling data visualization components enable end users to understand and analyze complex business data. The components are rich in features that provide out-of-the-box interactivity support. For detailed descriptions of the end user and presentation features for each component, see the following:

- Hierarchy Viewer components: End User and Presentation Features section in *Developing Web User Interfaces with Oracle ADF Faces*.
- Treemap and Sunburst components: End User and Presentation Features of Treemaps and Sunbursts section in *Developing Web User Interfaces with Oracle ADF Faces*.

Data Visualization Components Use Cases and Examples

For detailed descriptions of each data visualization use case and example, see the following:

- Hierarchy Viewer components: Hierarchy Viewer Use Cases and Examples section in *Developing Web User Interfaces with Oracle ADF Faces*.
- Treemap and Sunburst components: Treemap and Sunburst Use Cases and Examples section in *Developing Web User Interfaces with Oracle ADF Faces*.

Additional Functionality for Data Visualization Components

You may find it helpful to understand other Oracle ADF features before you data bind your data visualization components. Additionally, once you have added a data visualization component to your page, you may find that you need to add functionality such as validation and accessibility. Following are links to other functionality that data visualization components use:

- Partial page rendering: You may want a data visualization component to refresh to show new data based on an action taken on another component on the page. For more information, see the Rerendering Partial Page Content chapter in *Developing Web User Interfaces with Oracle ADF Faces*.
- Personalization: Users can change the way the data visualization components display at runtime, those values will not be retained once the user leaves the page unless you configure your application to allow user customization. For more information, see the Allowing User Customization on JSF Pages chapter in *Developing Web User Interfaces with Oracle ADF Faces*.
- Accessibility: You can make your data visualization components accessible. For more information, see the Developing Accessible ADF Faces Pages chapter in *Developing Web User Interfaces with Oracle ADF Faces*.
- Skins and styles: You can customize the appearance of data visualization components using an ADF skin that you apply to the application or by applying CSS style properties directly using a style-related property (`styleClass` or `inlineStyle`). For more information, see the Customizing the Appearance Using

Styles and Skins chapter in *Developing Web User Interfaces with Oracle ADF Faces*.

- Placeholder data controls: If you know the data visualization components on your page will eventually use ADF data binding, but you need to develop the pages before the data controls are ready, then you should consider using placeholder data controls, rather than manually binding the components. Using placeholder data controls will provide the same declarative development experience as using developed data controls. For more information, see Designing a Page Using Placeholder Data Controls.

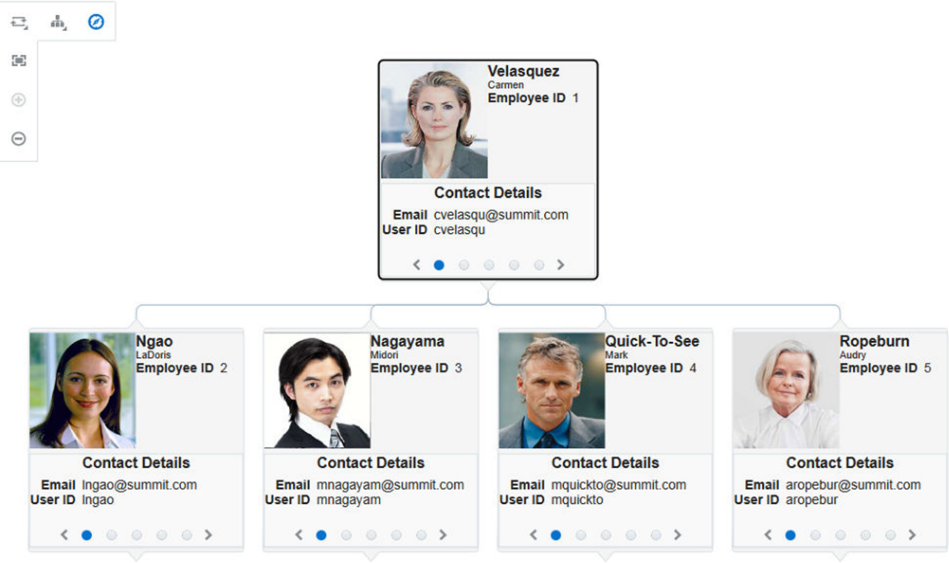
Creating Databound Hierarchy Viewers

A hierarchy viewer is an ADF Data Visualization component that visually displays data where parent-child relationships exist within the data. This component is useful where you want to display organization charts, network diagrams, social networks, or similar visual displays.

Hierarchy viewers use a shape called a node to reference the data in a hierarchy. The shape and content of the nodes are configurable, as well as the visual layout of the nodes. Nodes can display multiple views in a panel card.

Figure 43-1 shows a runtime view of a hierarchy viewer component that renders an organization chart.

Figure 43-1 Hierarchy Viewer Component Rendering an Organization Chart



Velasquez
Carmen
Employee ID 1
Contact Details
Email cvelasqu@summit.com
User ID cvelasqu

Ngao
LaDoris
Employee ID 2
Contact Details
Email lngao@summit.com
User ID lngao

Nagayama
Midori
Employee ID 3
Contact Details
Email mnagayam@summit.com
User ID mnagayam

Quick-To-See
Mark
Employee ID 4
Contact Details
Email mquickto@summit.com
User ID mquickto

Ropeburn
Audry
Employee ID 5
Contact Details
Email aropebur@summit.com
User ID aropebur

Each hierarchy viewer component (`dvt:hierarchyViewer`) that you create can include:

- One or more node elements (`dvt:node`)
- One or more link elements (`dvt:link`)

The optional panel card element (`dvt:panelCard`) can be used in conjunction with the hierarchy viewer component. The panel card provides a method to switch dynamically between multiple sets of content referenced by a node element using animation by, for example, horizontally sliding the content or flipping a node over.

For detailed information about hierarchy viewer end user and presentation features, use cases, tag structure, and adding special features to hierarchy viewers, see the Using Hierarchy Viewer Components chapter in *Developing Web User Interfaces with Oracle ADF Faces*.

How to Create a Hierarchy Viewer Using ADF Data Controls

Hierarchy viewers are based on data collections where a master-detail relationship exists between one or more detail collections and a master data collection. Using data controls in Oracle ADF, JDeveloper makes this a declarative task. You drag and drop a data collection from the Data Controls panel that generates one or more root nodes onto a JSF page.

Before you begin:

It may be helpful to have an understanding of databound hierarchy viewers. For more information, see Creating Databound Hierarchy Viewers.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Data Visualization Components.

You will need to complete these tasks:

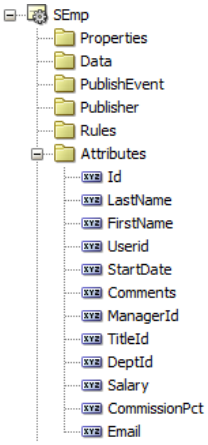
- Create an application module that contains instances of the view objects that you want in your data model for the hierarchy viewer, as described in Creating and Modifying an Application Module.

  For example, the data source for the hierarchy viewer in Figure 43-1 comes from a single table representing the employees in the Summit sample application for ADF DVT components. To set up a hierarchical relationship in the Summit ADF DVT sample application between the managers and employees using the `SEmp` table and to establish the highest level employee as the root of the hierarchy, do the following:

  1. Create an entity object based on the `SEmp` table.

     Figure 43-2 shows the `SEmp` entity object based on the `SEmp` table. Each row in the table includes both the employee's ID and the employee's manager ID. For help with creating entity objects, see Creating Entity Objects and Associations.

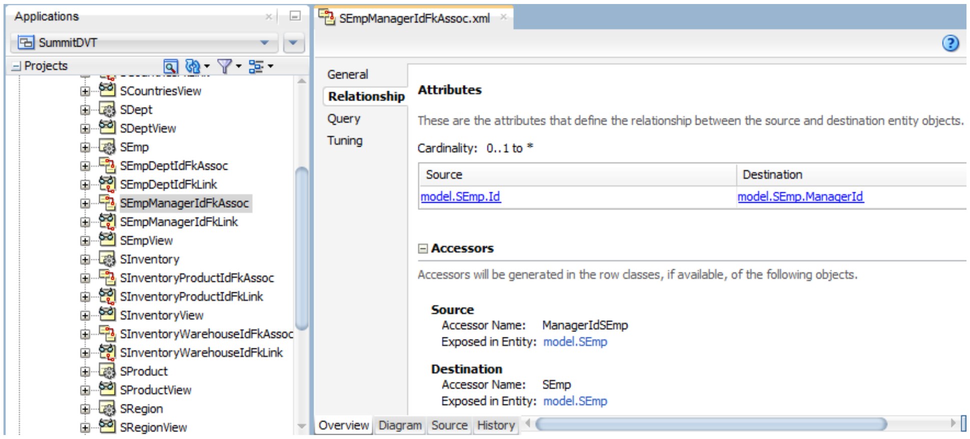
Figure 43-2 SEmp Entity Object in the Summit ADF DVT Sample Application



2. Create an association between `SEmp.Id` and `SEmp.ManagerID` to establish the relationship between the employee and manager.

   Figure 43-3 shows the `SEmpManagerIdFkAssoc` association in the Summit ADF DVT sample application. In this example, the association is automatically created when you create the `SEmp` entity object. For additional information about associations, see Creating Entity Objects and Associations.

Figure 43-3 Association Between Manager and Employees in the Summit ADF DVT Sample Application



Attributes

These are the attributes that define the relationship between the source and destination entity objects.

Cardinality: 0..1 to *

Source	Destination
model.SEmp.Id	model.SEmp.ManagerId

Accessors

Accessors will be generated in the row classes, if available, of the following objects.

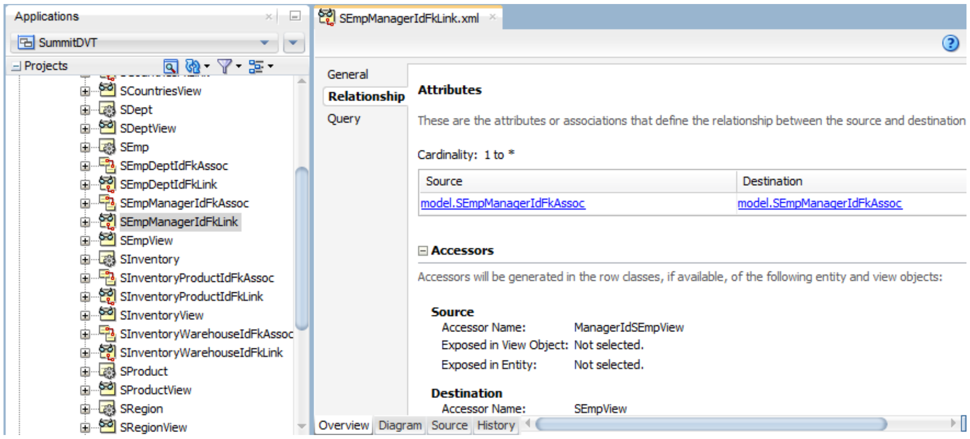
Source
Accessor Name: ManagerIdSEmp
Exposed in Entity: model.SEmp

Destination
Accessor Name: SEmp
Exposed in Entity: model.SEmp

3. To establish the hierarchy between the managers and employees, create a view link between the managers and employees based on the `SEmpManagerIdFkAssoc` association.

   Figure 43-4 shows the `SEmpManagerIDFKLink` view link in the Summit ADF DVT sample application. For additional information about establishing the master-detail relationship using view links, see How to Create a Master-Detail Hierarchy Based on Entity Associations.

Figure 43-4 View Link Between Managers and Employees in the Summit ADF DVT Sample Application



Applications

SummitDVT

Projects

- SCountriesView
- SDept
- SDeptView
- SEmp
- SEmpDeptIdFkAssoc
- SEmpDeptIdFkLink
- SEmpManagerIdFkAssoc
- SEmpManagerIdFkLink
- SEmpView
- SInventory
- SInventoryProductIdFkAssoc
- SInventoryProductIdFkLink
- SInventoryView
- SInventoryWarehouseIdFkAssoc
- SInventoryWarehouseIdFkLink
- SProduct
- SProductView
- SRegion
- SRegionView

SEmpManagerIdFkLink.xml

General
Relationship
Query

Attributes

These are the attributes or associations that define the relationship between the source and destination

Cardinality: 1 to *

Source	Destination
model.SEmpManagerIdFkAssoc	model.SEmpManagerIdFkAssoc

Accessors

Accessors will be generated in the row classes, if available, of the following entity and view objects:

Source

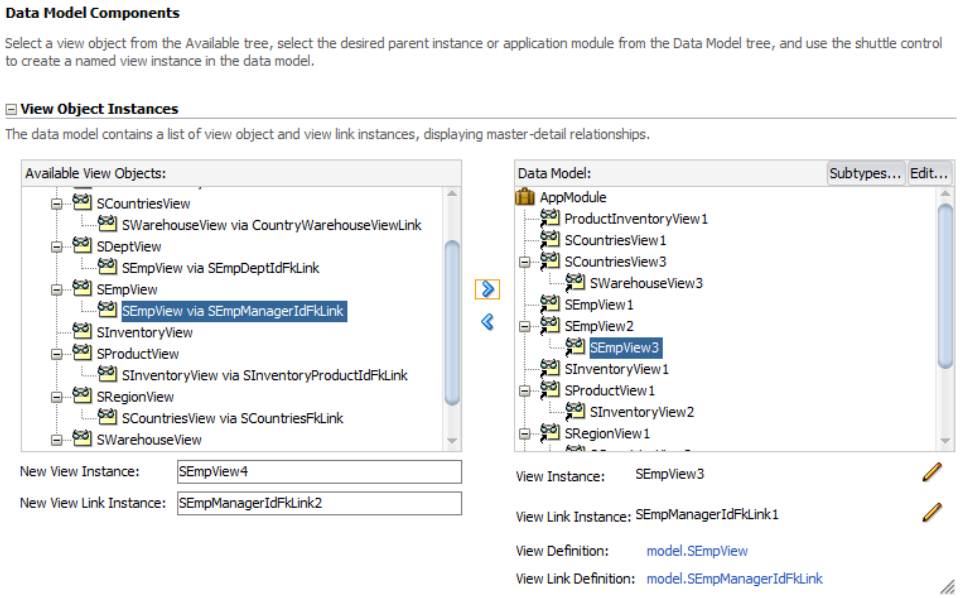
Accessor Name:	ManagerIdSEmpView
Exposed in View Object:	Not selected.
Exposed in Entity:	Not selected.

Destination

Accessor Name:	SEmpView

4. To use the SEmpManagerIDFKLink, create a new view object that ties the managers with the employees as shown in Figure 43-5.

Figure 43-5 Managers-Employees View Object in the Summit ADF DVT Sample Application



Data Model Components

Select a view object from the Available tree, select the desired parent instance or application module from the Data Model tree, and use the shuttle control to create a named view instance in the data model.

View Object Instances

The data model contains a list of view object and view link instances, displaying master-detail relationships.

Available View Objects:

- SCountriesView
  - SWarehouseView via CountryWarehouseViewLink
- SDeptView
  - SEmpView via SEmpDeptIdFkLink
- SEmpView
  - SEmpView via SEmpManagerIdFkLink
- SInventoryView
- SProductView
  - SInventoryView via SInventoryProductIdFkLink
- SRegionView
  - SCountriesView via SCountriesFkLink
- SWarehouseView

Data Model:

- AppModule
  - ProductInventoryView1
  - SCountriesView1
  - SCountriesView3
    - SWarehouseView3
  - SEmpView1
  - SEmpView2
    - SEmpView3
  - SInventoryView1
  - SProductView1
    - SInventoryView2
  - SRegionView1

New View Instance:	SEmpView4
New View Link Instance:	SEmpManagerIdFkLink2

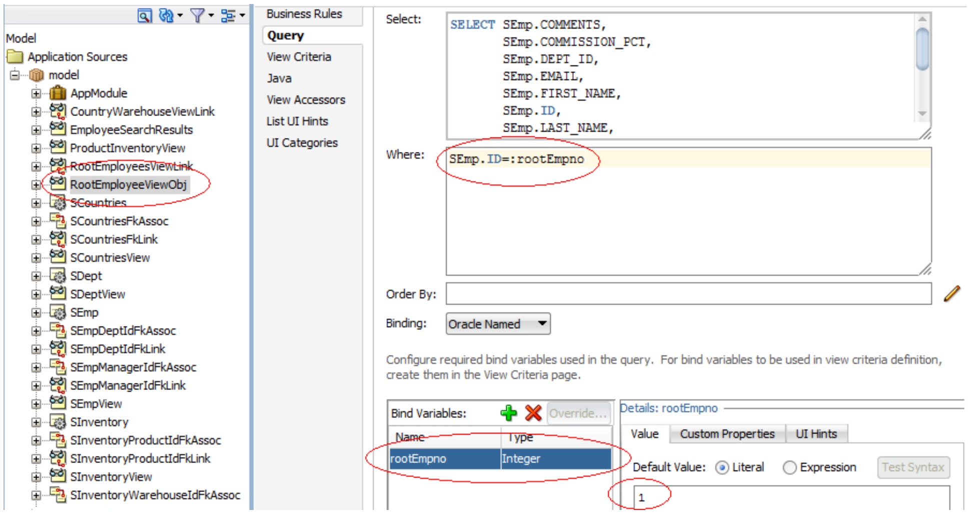
View Instance:	SEmpView3
View Link Instance:	SEmpManagerIdFkLink1
View Definition:	model.SEmpView
View Link Definition:	model.SEmpManagerIdFkLink

5. Create a view object that retrieves the root value from the data collection.

   For example, in the Summit ADF DVT sample application, Carmen Velasquez is the highest level manager and has a employee ID of 1. To create a view object based on the Summit ADF DVT sample application, create a view object that retrieves the employee whose ID is 1.

   Figure 43-6 shows the RootEmployeeViewObject for the Summit ADF DVT sample application. The query retrieves the employee whose ID is 1 and stores it in the rootEmpno bind variable. For more information about working with view objects and bind variables, see Working with Bind Variables.

Figure 43-6   RootEmployeeViewObject Sample for Hierarchy Viewer



Select:

```
SELECT SEmp.COMMENTS,
       SEmp.COMMISSION_PCT,
       SEmp.DEPT_ID,
       SEmp.EMAIL,
       SEmp.FIRST_NAME,
       SEmp.ID,
       SEmp.LAST_NAME,
```

Where:

```
SEmp.ID=:rootEmpno
```

Order By:

Binding: Oracle Named

Configure required bind variables used in the query. For bind variables to be used in view criteria definition, create them in the View Criteria page.

Bind Variables: Override...

Name	Type
rootEmpno	Integer

Details: rootEmpno

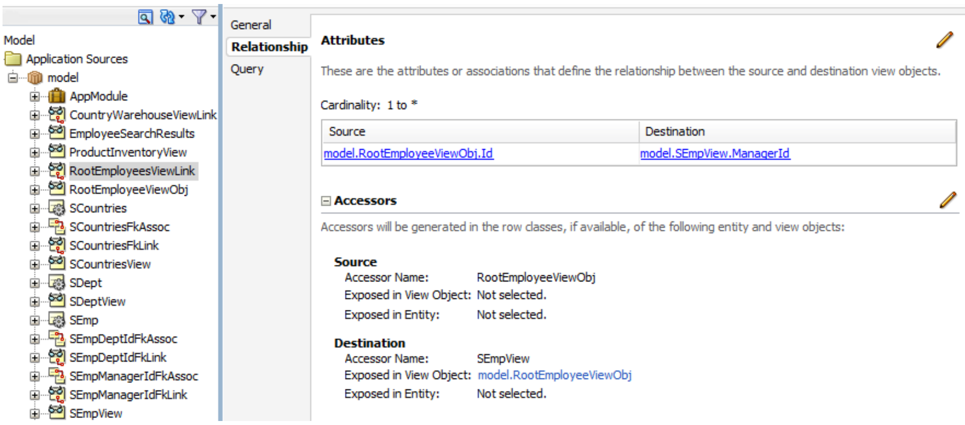
Value | Custom Properties | UI Hints

Default Value: ● Literal ○ Expression Test Syntax

1

6. To establish the hierarchy between the root employee and the lower-level managers and employees, add a view link between the `RootEmployeeViewObj` and the managers-employees view object created in Step 4.

Figure 43-7   RootEmployeesViewLink in the Summit ADF DVT Sample Application



Attributes

These are the attributes or associations that define the relationship between the source and destination view objects.

Cardinality: 1 to *

Source	Destination
model.RootEmployeeViewObj.Id	model.SEmpView.ManagerId

Accessors

Accessors will be generated in the row classes, if available, of the following entity and view objects:

Source

Accessor Name:	RootEmployeeViewObj
Exposed in View Object:	Not selected.
Exposed in Entity:	Not selected.

Destination

Accessor Name:	SEmpView
Exposed in View Object:	model.RootEmployeeViewObj
Exposed in Entity:	Not selected.

7. In the Data Controls panel, click **Refresh** to update the data controls.

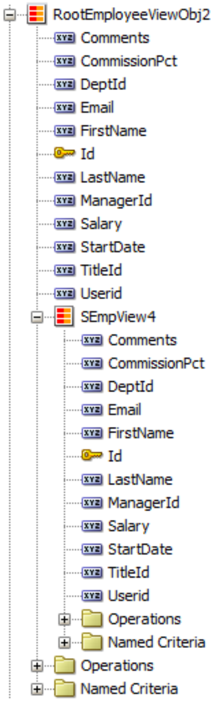
- Create a JSF page as described in the How to Create JSF Pages section of *Developing Web User Interfaces with Oracle ADF Faces*.

To create a hierarchy viewer using the Data Controls panel:

1. From the Data Controls panel, select a collection.

   Figure 43-8 shows an example where you could select the `RootEmployeeViewObj2` collection in the Data Controls panel to create a hierarchy viewer representing the personnel data in an organizational chart.

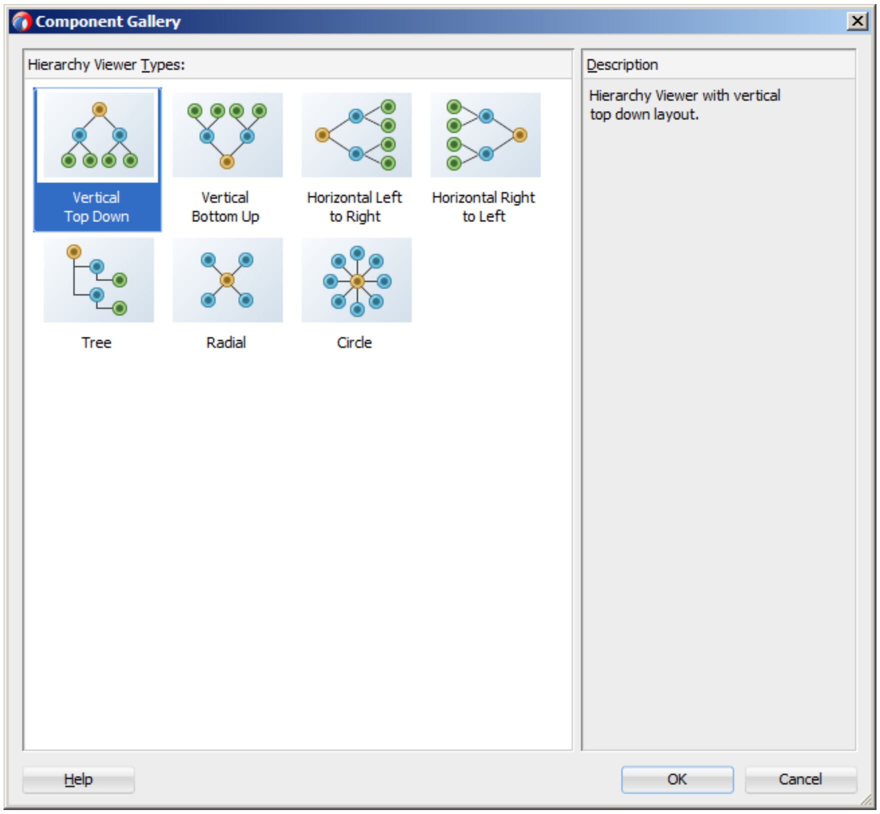
Figure 43-8 Data Collection for Personnel Organizational Chart



2. Drag the collection onto a JSF page and, from the context menu, choose **Hierarchy Viewer.**

3. From the Component Gallery, select the layout of the hierarchy viewer you want to create. Figure 43-9 shows the Component Gallery with the vertical top down layout selected.

Figure 43-9 Hierarchy Viewer Component Gallery



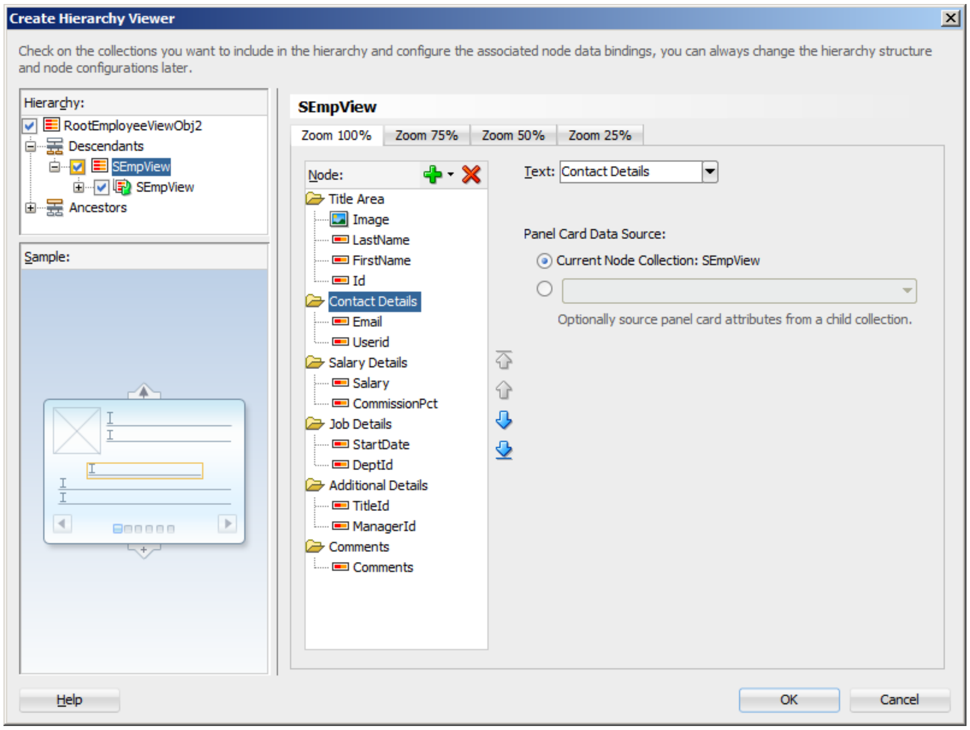
Hierarchy Viewer with vertical top down layout.

4. In the Create Hierarchy Viewer dialog, enter the following:
   - **Hierarchy**: Select the collections you want to include as nodes in the runtime diagram. By default, the root collection is selected. You can also choose to configure ancestors or descendants in the hierarchy.
   - **Node**: For each collection that you select in the Hierarchy list, configure the attributes in the Title Area, and the title and attributes in each panel card, using one or more of the zoom levels available to you. By default, the node for the 100% zoom level follows an algorithm that:
     - Assigns the first three attributes in the data collection to the **Title Area**.
     - Assigns the next two attributes to the first panel card.
     - Assigns the following two attributes to the second panel card.

     Select the panel card title or node attributes to configure one or more of the following for that element:
     - **Text**: Available for panel card titles. Not available for the **Title Area** element. Enter text for the panel card title in the hierarchy, or choose **Select Text Resource** from the dropdown list to open a dialog to select or add a text resource to use for the title. The text resource is a translatable string from an application resource bundle. If you need help, press F1 or click **Help**.
     - **Panel Card Data Source**: Available for panel cards. The data source indicates which data collection will be used for the panel card attributes. By default, the collection associated with the current hierarchy level will be used for attribute selection.

Figure 43-10 shows a Create Hierarchy Viewer dialog for a hierarchy viewer using data collections based on the Summit ADF schema. In this example, the **Panel Card Data Source** is the default `SEmpView` collection.

Figure 43-10 Hierarchy Viewer Configured With Default Panel Card Data Source



Create Hierarchy Viewer

Check on the collections you want to include in the hierarchy and configure the associated node data bindings, you can always change the hierarchy structure and node configurations later.

Hierarchy:
- ☑ RootEmployeeViewObj2
  - Descendants
    - ☑ SEmpView
      - ☑ SEmpView
  - Ancestors

Sample:

SEmpView

Zoom 100% | Zoom 75% | Zoom 50% | Zoom 25%

Node:
- Title Area
  - Image
  - LastName
  - FirstName
  - Id
- Contact Details
  - Email
  - Userid
- Salary Details
  - Salary
  - CommissionPct
- Job Details
  - StartDate
  - DeptId
- Additional Details
  - TitleId
  - ManagerId
- Comments
  - Comments

Text: Contact Details

Panel Card Data Source:

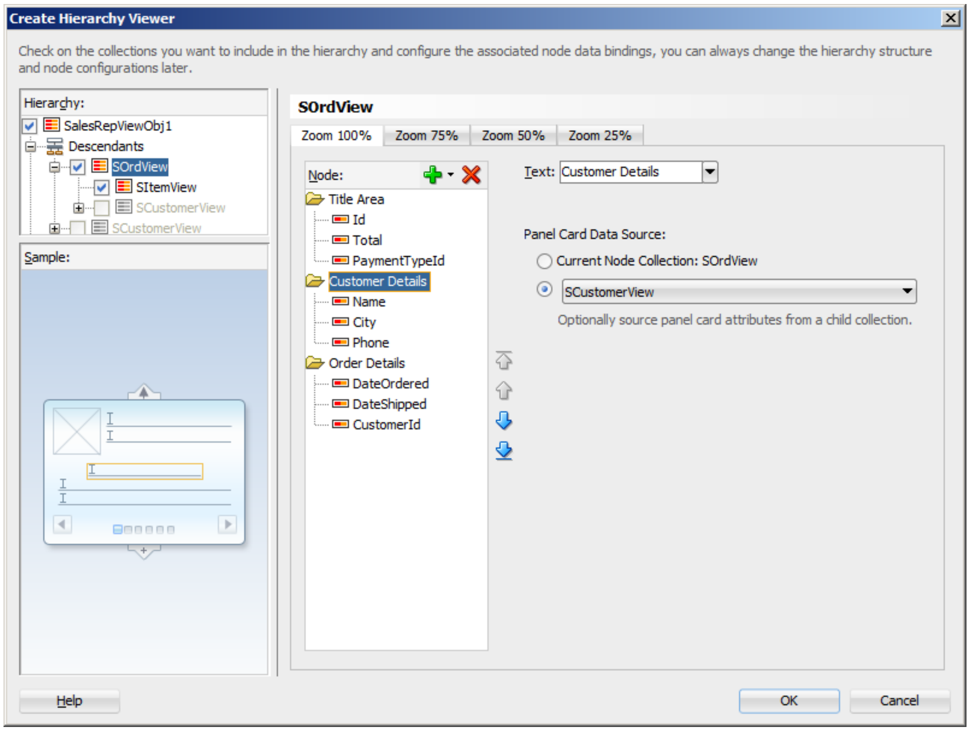
● Current Node Collection: SEmpView

○

Optionally source panel card attributes from a child collection.

To use an alternative child accessor for the panel card attributes, select an alternative child accessor from the dropdown list as shown in Figure 43-11. For additional information, see How to Configure an Alternate View Object for a Databound Panel Card.

Figure 43-11 Create Hierarchy Viewer Dialog Showing Alternate Panel Card Data Source



Create Hierarchy Viewer

Check on the collections you want to include in the hierarchy and configure the associated node data bindings, you can always change the hierarchy structure and node configurations later.

Text: Customer Details

Panel Card Data Source:

○ Current Node Collection: SOrdView

● SCustomerView

Optionally source panel card attributes from a child collection.

- **Component**: Available for attributes. Select the type of component to use to display the data in the node. Valid values are: **ADF Output Text w/Label**, **ADF Output Text**, **ADF Output Formatted Text w/Label**, and **ADF Output Formatted Text**. For more information about using these components, see the Using Output Components chapter in *Developing Web User Interfaces with Oracle ADF Faces*.
- **Attribute**: Available for attribute and image elements. From the dropdown list, select the attribute in the data collection that represents the value you wish to display in the hierarchy viewer node.
- **Label**: Available for attributes when you select the **Output Text w/ Label** or **ADF Output Formatted Text w/Label** component from the **Component** field's dropdown list. You can use the default **Use Attribute Name** to render the value as a string using the label from the `UIHints` for that attribute in the underlying `ViewObject`. You can also select **No Label** from the attribute's dropdown menu to render no label for the attribute, or choose **Select Text Resource** to open a Select Text Resource dialog to select or add a text resource to use for the label. The text resource is a translatable string from an application resource bundle.

Click the **New** icon to add a new panel card, attribute, or image to the node, relative to your selection in the node. After selecting an existing element, use the arrow icons (**Up**, **Down**, **Top**, **Bottom**) to reorder the element, or use the **Delete** icon to delete the selected element.

Use the 75%, 50%, and 25% pages to specify the expanded display of the hierarchy at each page level. Select **Add Zoom Level** to enable the zoom level for a page level. By default, the algorithms to assign node attributes and panel cards in the node are similar to the 100% zoom level.

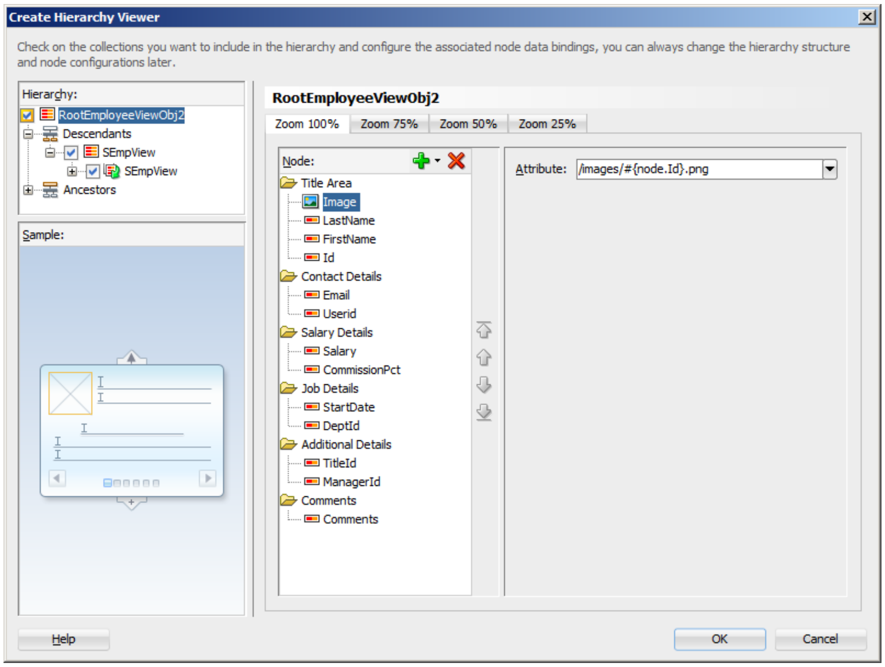
> **Note:**
>
> The hierarchy viewer component defines four zoom levels. You cannot modify these zoom levels or create new zoom levels. The default zoom level is 100%.

- **Sample**: A nonconfigurable display of the sample outline of the hierarchy viewer node. Areas such as Title Area and attributes are highlighted in the sample when selected in the node area.

5. Click **OK**.

Figure 43-12 shows a completed Create Hierarchy Viewer dialog for the root level of the hierarchy viewer using data from a data collection named `RootEmployeeViewObj2`.

Figure 43-12 Create Hierarchy Viewer Dialog



Create Hierarchy Viewer

Check on the collections you want to include in the hierarchy and configure the associated node data bindings, you can always change the hierarchy structure and node configurations later.

Hierarchy:
- ☑ RootEmployeeViewObj2
  - Descendants
    - ☑ SEmpView
      - ☑ SEmpView
  - Ancestors

Sample:

RootEmployeeViewObj2

Zoom 100% | Zoom 75% | Zoom 50% | Zoom 25%

Node:
- Title Area
  - Image
  - LastName
  - FirstName
  - Id
- Contact Details
  - Email
  - Userid
- Salary Details
  - Salary
  - CommissionPct
- Job Details
  - StartDate
  - DeptId
- Additional Details
  - TitleId
  - ManagerId
- Comments
  - Comments

Attribute: /images/#{node.Id}.png

After completing the Create Hierarchy Viewer dialog, you can use the Properties window to specify settings for the hierarchy viewer attributes, and you can also use the child tags associated with the hierarchy viewer tag to customize the hierarchy viewer further. For detailed information about hierarchy viewer end user and presentation features, use cases, tag structure, and adding special features to hierarchy viewers, see the Using Hierarchy Viewer Components chapter in *Developing Web User Interfaces with Oracle ADF Faces*.

When editing the hierarchy viewer in the visual editor, the display approximates the runtime display and functionality. You can rearrange attributes in the Structure window, or bind new attributes by dragging and dropping attributes from the Data Controls panel onto the Structure window.

What Happens When You Create a Databound Hierarchy Viewer

Creating a hierarchy viewer from Data Controls panel has the following effect:

- Creates the bindings for the hierarchy viewer in the page definition file of the JSF page
- Adds the necessary tags to the JSF page for the hierarchy viewer component

The example below displays bindings that JDeveloper generated for a hierarchy viewer component. The rules for populating the nodes of the master-detail tree are defined as a node definition. The example shows that two node definitions were generated. Each of these node definitions references a view object and associated attributes. The code example also references an accessor `SEmpView`.

```
<parameters/>
<executables>
  <variableIterator id="variables"/>
  <iterator Binds="RootEmployeeViewObj2" RangeSize="25"
            DataControl="AppModuleDataControl"
            id="RootEmployeeViewObj2Iterator"/>
</executables>
<bindings>
  <tree IterBinding="RootEmployeeViewObj2Iterator" id="RootEmployeeViewObj2">
    <nodeDefinition DefName="model.RootEmployeeViewObj"
                    Name="RootEmployeeViewObj20">
      <AttrNames>
        <Item Value="ManagerId"/>
        <Item Value="TitleId"/>
        <Item Value="Userid"/>
        <Item Value="Comments"/>
        <Item Value="DeptId"/>
        <Item Value="StartDate"/>
        <Item Value="Email"/>
        <Item Value="Salary"/>
        <Item Value="FirstName"/>
        <Item Value="Id"/>
        <Item Value="CommissionPct"/>
        <Item Value="LastName"/>
      </AttrNames>
      <Accessors>
        <Item Value="SEmpView">
          <Properties>
            <CustomProperties>
              <Property Name="hierarchyType" Value="children"/>
            </CustomProperties>
          </Properties>
        </Item>
      </Accessors>
    </nodeDefinition>
    <nodeDefinition DefName="model.SEmpView" Name="RootEmployeeViewObj21">
      <AttrNames>
        <Item Value="ManagerId"/>
        <Item Value="TitleId"/>
        <Item Value="Userid"/>
        <Item Value="Comments"/>
        <Item Value="DeptId"/>
        <Item Value="StartDate"/>
        <Item Value="Email"/>
        <Item Value="Salary"/>
```

```
          <Item Value="FirstName"/>
          <Item Value="Id"/>
          <Item Value="CommissionPct"/>
          <Item Value="LastName"/>
        </AttrNames>
        <Accessors>
          <Item Value="SEmpView">
            <Properties>
              <CustomProperties>
                <Property Name="hierarchyType" Value="children"/>
              </CustomProperties>
            </Properties>
          </Item>
        </Accessors>
      </nodeDefinition>
    </tree>
</bindings>
```

The example below shows the code generated on the JSF page that is associated with the page definition file in the above example. For brevity, the details in the facet elements named `zoom75`, `zoom50`, and `zoom25` and panel card `showDetailItem` components have been omitted. In addition, the details for the second node in the hierarchy viewer have also been omitted.

The example shows a hierarchy viewer component that references the `RootEmployeeViewObj2` tree binding. It includes a node `(dvt:node)` component that in turn includes a panel card component `(dvt:panelCard)`. The panel card component defines `slide_horz` as the effect to use when changing the display of content referenced by the node.

Once you create your hierarchy viewer, you can modify the layout of the component or add additional components, such as a panel card, using the Create Hierarchy dialog illustrated in Figure 43-12. To open the dialog, use the **Edit** icon in the Properties window for the `hierarchyViewer` component. You can also customize the layout of a hierarchy viewer component directly in the code, in the visual editor, or by setting values in the Properties window. You can add additional components, such as panel cards, using the Components window.

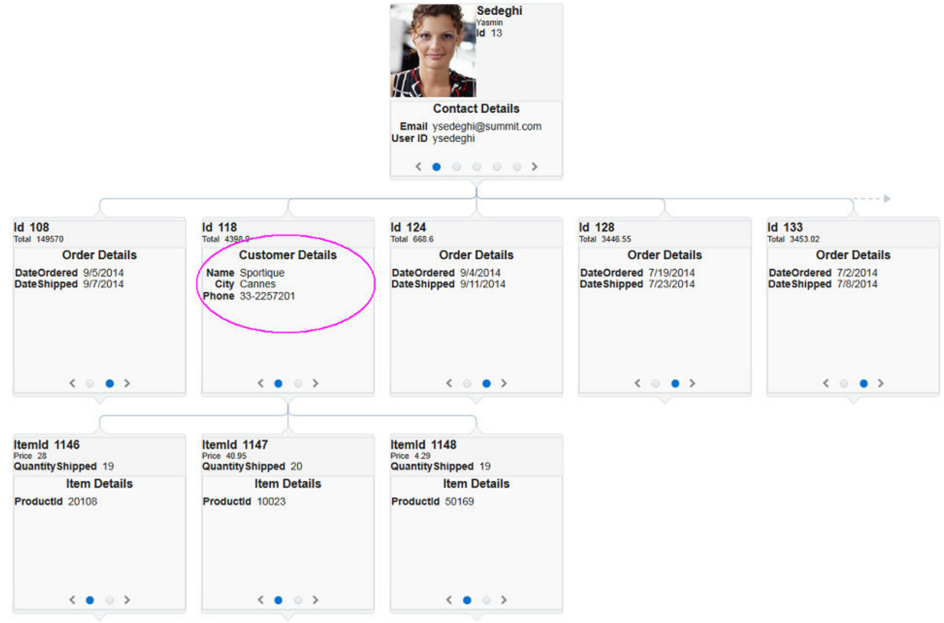
```
<dvt:hierarchyViewer id="hv1" var="node"
                     value="#{bindings.RootEmployeeViewObj2.treeModel}"

selectionListener="#{bindings.RootEmployeeViewObj2.treeModel.makeCurrent}"
                     detailWindow="none" layout="hier_vert_top"
                     levelFetchSize="#{bindings.RootEmployeeViewObj2.rangeSize}"
                     styleClass="AFStretchWidth" summary="Hierarchy Viewer Demo">
  <dvt:link linkType="orthogonalRounded" id="l1"/>
  <dvt:node type="model.RootEmployeeViewObj" width="233" height="233" id="n1">
    <f:facet name="zoom100">
      <af:panelGroupLayout styleClass="AFStretchWidth AFHVNodeStretchHeight
                                       AFHVNodePadding"
                           layout="vertical" id="pgl1">
        <af:panelGroupLayout layout="horizontal" id="pgl2">
          <af:panelGroupLayout id="pgl7">
            <af:image source="/images/#{node.Id}.png" shortDesc="Employee Image"
                      id="i1" styleClass="AFHVNodeImageSize"/>
          </af:panelGroupLayout>
          <af:panelGroupLayout layout="vertical" id="pgl3">
            <af:outputText value="#{node.LastName}"
              shortDesc="#{bindings.RootEmployeeViewObj2.hints.LastName.tooltip}"
                           styleClass="AFHVNodeTitleTextStyle" id="ot1"/>
            <af:outputText value="#{node.FirstName}"

shortDesc="#{bindings.RootEmployeeViewObj2.hints.FirstName.tooltip}"
                           styleClass="AFHVNodeSubtitleTextStyle" id="ot2"/>
            <af:panelLabelAndMessage

label="#{bindings.RootEmployeeViewObj2.hints.Id.label}"
                           styleClass="AFHVNodeLabelStyle" id="plam1">
              <af:outputText value="#{node.Id}"

shortDesc="#{bindings.RootEmployeeViewObj2.hints.Id.tooltip}"
                             styleClass="AFHVNodeTextStyle" id="ot3">
                <af:convertNumber groupingUsed="false"

pattern="#{bindings.RootEmployeeViewObj2.hints.Id.format}"/>
              </af:outputText>
            </af:panelLabelAndMessage>
          </af:panelGroupLayout>
        </af:panelGroupLayout>
        <af:spacer height="5" id="s1"/>
        <dvt:panelCard effect="slide_horz" styleClass="AFHVNodePadding" id="pc1">
          <af:showDetailItem text="Contact Details" id="sdi1">
            <af:spacer height="2" id="s2"/>
            <af:panelFormLayout styleClass="AFStretchWidth AFHVNodeStretchHeight
                                            AFHVNodePadding"
                                id="pfl1">
              <af:panelLabelAndMessage
                       label="#{bindings.RootEmployeeViewObj2.hints.Email.label}"
                       styleClass="AFHVPanelCardLabelStyle" id="plam2">
                <af:outputText value="#{node.Email}"
                 shortDesc="#{bindings.RootEmployeeViewObj2.hints.Email.tooltip}"
                             styleClass="AFHVPanelCardTextStyle" id="ot4"/>
              </af:panelLabelAndMessage>
              <af:panelLabelAndMessage

label="#{bindings.RootEmployeeViewObj2.hints.Userid.label}"
                       styleClass="AFHVPanelCardLabelStyle" id="plam3">
                <af:outputText value="#{node.Userid}"

shortDesc="#{bindings.RootEmployeeViewObj2.hints.Userid.tooltip}"
                             styleClass="AFHVPanelCardTextStyle" id="ot5"/>
              </af:panelLabelAndMessage>
            </af:panelFormLayout>
          </af:showDetailItem>
            ... remaining showDetailItem panel card details omitted
        </dvt:panelCard>
      </af:panelGroupLayout>
    </f:facet>
      ... remaining zoom levels omitted
  </dvt:node>
  <dvt:node type="model.SEmpView" width="233" height="233" id="n2">
    ... second node details are similar to first node and are omitted
  </dvt:node>
</dvt:hierarchyViewer>
```

How to Configure an Alternate View Object for a Databound Panel Card

You can specify an alternate view object as the data source for a panel card when you create or edit a databound hierarchy viewer. For example, you may have a data collection that has a master-detail relationship between sales representatives and orders, but you want your panel card to display details about the customer who placed the order.

Figure 43-13 shows a portion of the runtime view of a hierarchy viewer configured to display the orders and order details for sales representatives using the Summit ADF schema. In this example, the hierarchy viewer is configured to use an alternate view object for the Customer Details panel card.

Figure 43-13   Hierarchy Viewer Panel Card Configured With Alternate View Object



Sedeghi
Yasmin
Id 13
Contact Details
Email ysedeghi@summit.com
User ID ysedeghi

Id 108
Total 149570
Order Details
DateOrdered 9/5/2014
DateShipped 9/7/2014

Id 118
Total 4398.9
Customer Details
Name Sportique
City Cannes
Phone 33-2257201

Id 124
Total 668.6
Order Details
DateOrdered 9/4/2014
DateShipped 9/11/2014

Id 128
Total 3446.55
Order Details
DateOrdered 7/19/2014
DateShipped 7/23/2014

Id 133
Total 3453.02
Order Details
DateOrdered 7/2/2014
DateShipped 7/8/2014

ItemId 1146
Price 28
QuantityShipped 19
Item Details
ProductId 20108

ItemId 1147
Price 40.95
QuantityShipped 20
Item Details
ProductId 10023

ItemId 1148
Price 4.29
QuantityShipped 19
Item Details
ProductId 50169

The alternate view object must be a child of the parent collection. You can establish this parent-child relationship by creating a view link between the parent and child collection.

Before you begin:

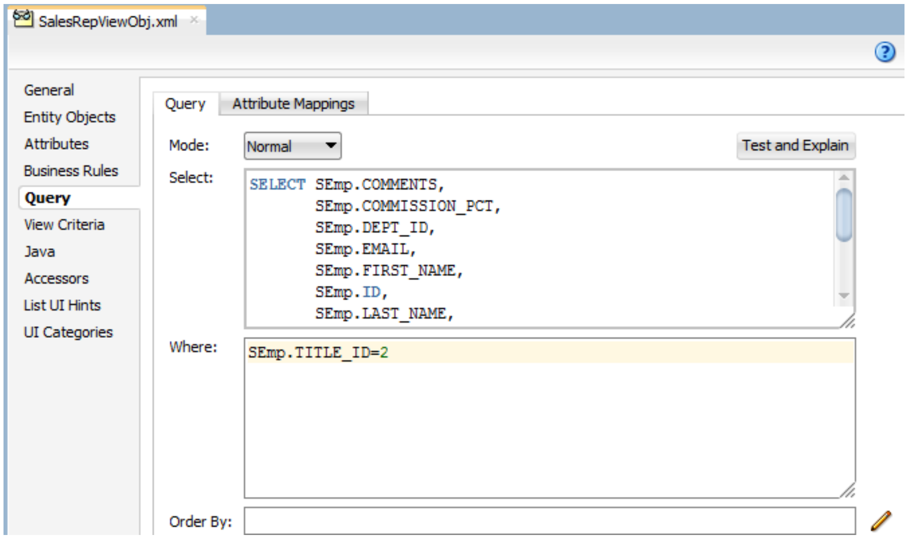
It may be helpful to have an understanding of databound hierarchy viewers. For more information, see Creating Databound Hierarchy Viewers.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Data Visualization Components.

Create an application module that contains instances of the entity and view objects that you want in your application, as described in Creating and Modifying an Application Module.

Create the data model for the hierarchy viewer. For example, the hierarchy viewer in Figure 43-13 uses a top level view object based on the `SEmp` entity in the Summit ADF DVT sample application. In this example, the view object retrieves all sales representatives in the Summit ADF employee database by retrieving all employee records with a `TITLE_ID` of `2`. Figure 43-14 shows the `SalesRepViewObj` view object.

Figure 43-14 Sales Representatives View Object Using the Summit ADF Schema



Mode: Normal

Select:

```
SELECT  SEmp.COMMENTS,
        SEmp.COMMISSION_PCT,
        SEmp.DEPT_ID,
        SEmp.EMAIL,
        SEmp.FIRST_NAME,
        SEmp.ID,
        SEmp.LAST_NAME,
```

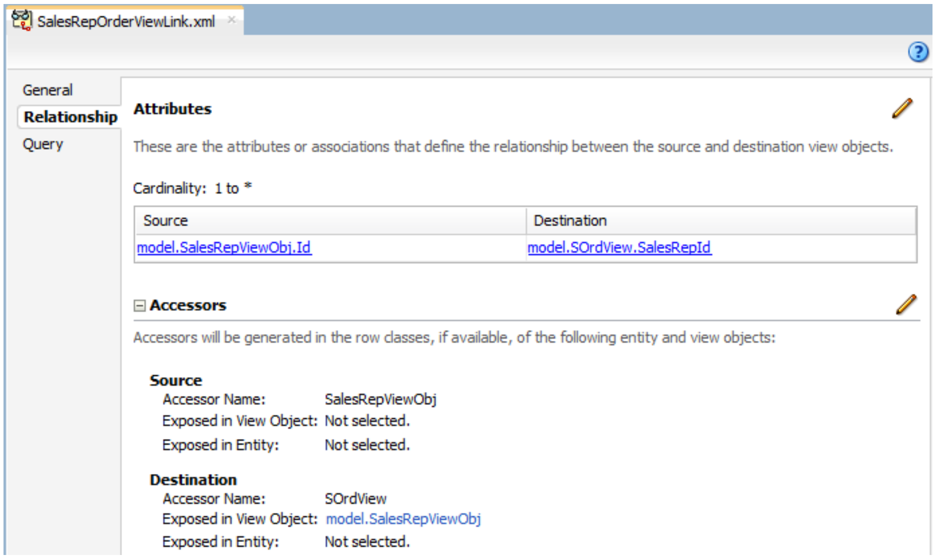
Where:

```
SEmp.TITLE_ID=2
```

For information about creating a view object, see How to Create an Entity-Based View Object.

The master-detail relationship between the Summit sales representatives and orders is established with a view link between `SalesRepViewObj` and `OrdersViewObj`. Figure 43-15 shows the relationship between the view objects.

Figure 43-15   View Link Between Summit Sales Representatives and Orders



SalesRepOrderViewLink.xml

Attributes

These are the attributes or associations that define the relationship between the source and destination view objects.

Cardinality: 1 to *

Source	Destination
model.SalesRepViewObj.Id	model.SOrdView.SalesRepId

Accessors

Accessors will be generated in the row classes, if available, of the following entity and view objects:

Source

Accessor Name:	SalesRepViewObj
Exposed in View Object:	Not selected.
Exposed in Entity:	Not selected.

Destination

Accessor Name:	SOrdView
Exposed in View Object:	model.SalesRepViewObj
Exposed in Entity:	Not selected.

For additional information about establishing the master-detail relationship using view links, see How to Create a Master-Detail Hierarchy Based on Entity Associations.

To configure an alternate view object for a databound panel card:

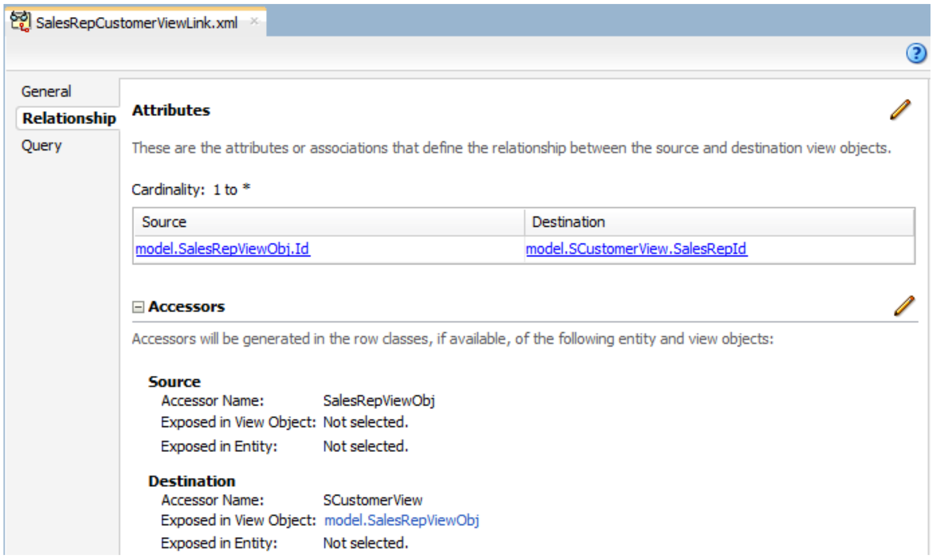
1. If you have not already done so, create the view object that will be the alternate data source for the panel card.

   For example, the hierarchy viewer in Figure 43-13 uses a view object based on the `SCustomer` entity object in the Summit ADF DVT sample application.

2. Create a view link between the parent collection and the alternate view object.

   Figure 43-16 shows the view link between the sales representatives and customers in the Summit ADF DVT sample application. For help with creating view links, see How to Create a Master-Detail Hierarchy Based on Entity Associations.

Figure 43-16 View Link Between Sales Representatives and Customers in the Summit ADF DVT Sample Application



Attributes

These are the attributes or associations that define the relationship between the source and destination view objects.

Cardinality: 1 to *

Source	Destination
model.SalesRepViewObj.Id	model.SCustomerView.SalesRepId

⊟ **Accessors**

Accessors will be generated in the row classes, if available, of the following entity and view objects:

Source

Accessor Name:	SalesRepViewObj
Exposed in View Object:	Not selected.
Exposed in Entity:	Not selected.

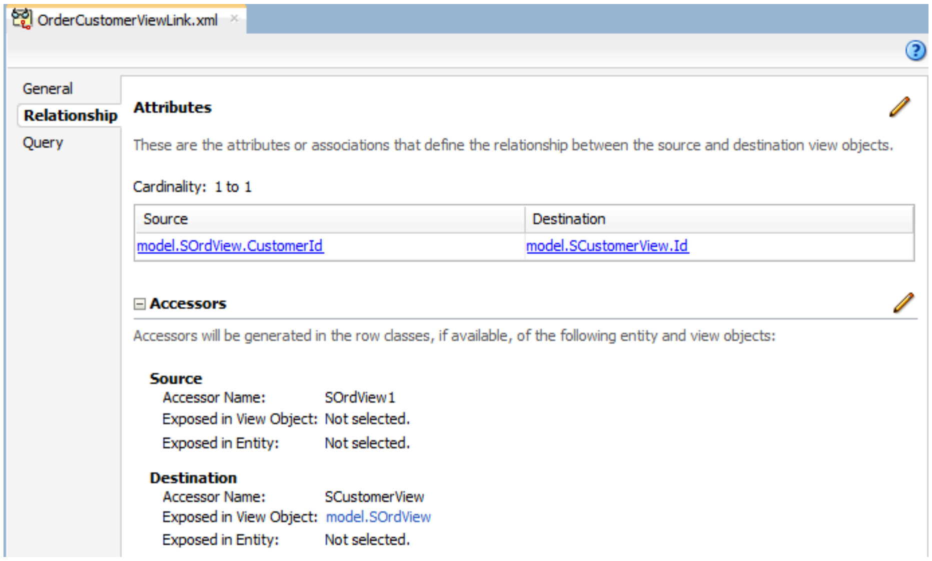
Destination

Accessor Name:	SCustomerView
Exposed in View Object:	model.SalesRepViewObj
Exposed in Entity:	Not selected.

3. If needed, create a view link to establish the hierarchy between the parent collection and the alternate child collection.

   For example, to establish the hierarchy between orders and customers in the Summit ADF DVT sample application, create a view link between the `SOrdView` and `SCustomerView` collections. Figure 43-17 shows the sample view link.

Figure 43-17 View Link Between Orders and Customers in the Summit ADF DVT Sample Application



Attributes

These are the attributes or associations that define the relationship between the source and destination view objects.

Cardinality: 1 to 1

Source	Destination
model.SOrdView.CustomerId	model.SCustomerView.Id

⊟ **Accessors**

Accessors will be generated in the row classes, if available, of the following entity and view objects:

Source

Accessor Name:	SOrdView1
Exposed in View Object:	Not selected.
Exposed in Entity:	Not selected.

Destination

Accessor Name:	SCustomerView
Exposed in View Object:	model.SOrdView
Exposed in Entity:	Not selected.

What Happens When You Use an Alternate View Object for a Hierarchy Viewer Panel Card

When you configure your hierarchy viewer to use an alternate view object for the panel card data source, JDeveloper adds the bindings for the hierarchy viewer and alternate source in the page definition file of the JSF page.

The example below shows the sample bindings for the hierarchy viewer displayed in Figure 43-13. The entries for the alternate source are highlighted. Note that the alternate data source is defined as one of the hierarchy viewer's node definitions.

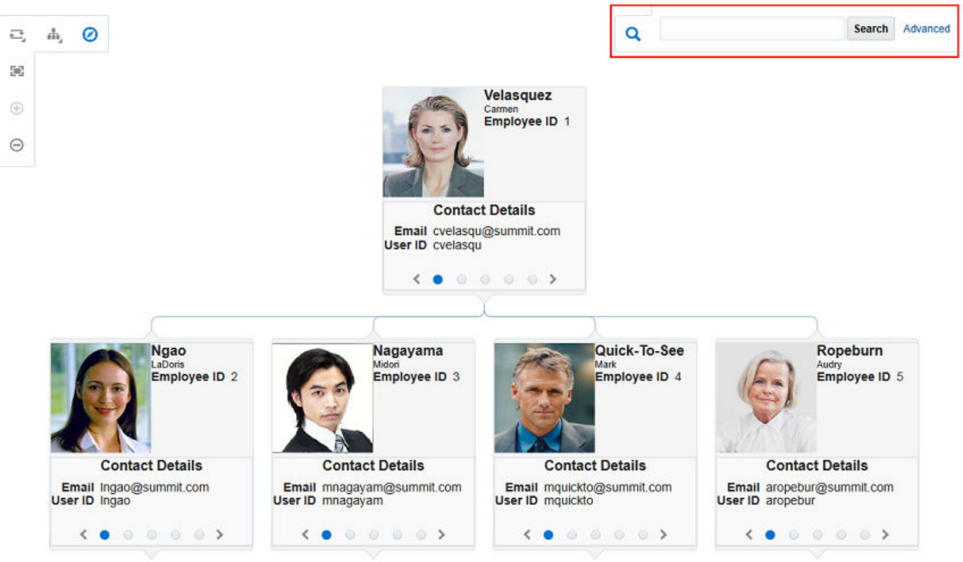
```
<bindings>
  <tree IterBinding="SalesRepViewObj2Iterator" id="SalesRepViewObj2">
    <nodeDefinition DefName="model.SalesRepViewObj" Name="SalesRepViewObj20">
      <AttrNames>
        <Item Value="ManagerId"/>
        <Item Value="TitleId"/>
        <Item Value="Userid"/>
        <Item Value="Comments"/>
        <Item Value="DeptId"/>
        <Item Value="StartDate"/>
        <Item Value="Email"/>
        <Item Value="Salary"/>
        <Item Value="FirstName"/>
        <Item Value="Id"/>
        <Item Value="CommissionPct"/>
        <Item Value="LastName"/>
      </AttrNames>
      <Accessors>
        <Item Value="SOrdView">
          <Properties>
            <CustomProperties>
              <Property Name="hierarchyType" Value="children"/>
            </CustomProperties>
          </Properties>
        </Item>
      </Accessors>
    </nodeDefinition>
    <nodeDefinition DefName="model.SOrdView" Name="SalesRepViewObj21">
      <AttrNames>
        <Item Value="DateShipped"/>
        <Item Value="DateOrdered"/>
        <Item Value="Id"/>
        <Item Value="Total"/>
      </AttrNames>
      <Accessors>
        <Item Value="SItemView">
          <Properties>
            <CustomProperties>
              <Property Name="hierarchyType" Value="children"/>
            </CustomProperties>
          </Properties>
        </Item>
        <Item Value="SCustomerView"/>
      </Accessors>
    </nodeDefinition>
    <nodeDefinition DefName="model.SCustomerView" Name="SalesRepViewObj22">
      <AttrNames>
        <Item Value="Name"/>
        <Item Value="Phone"/>
        <Item Value="City"/>
      </AttrNames>
    </nodeDefinition>
    <nodeDefinition DefName="model.SItemView" Name="SalesRepViewObj23">
      <AttrNames>
        <Item Value="QuantityShipped"/>
        <Item Value="ItemId"/>
        <Item Value="Quantity"/>
        <Item Value="Price"/>
        <Item Value="OrdId"/>
        <Item Value="ProductId"/>
      </AttrNames>
    </nodeDefinition>
  </tree>
</bindings>
```

How to Create a Databound Search in a Hierarchy Viewer

The search function in a hierarchy viewer is based on the searchable attributes or columns of the data collection that is the basis of the hierarchy viewer data model. Using a query results collection defined in data controls in Oracle ADF, JDeveloper makes this a declarative task. You drag and drop an **ExecuteWithParams** operation into an existing hierarchy viewer component on the page.

Figure 43-18 shows the Summit ADF DVT employee hierarchy viewer configured to search for employees by last name. In this example, the user can enter an employee's last name in the Search panel or specify a pattern match using `%` or `_`.

Figure 43-18 Hierarchy Viewer With Search Panel



Velasquez
Carmen
Employee ID 1
Contact Details
Email cvelasqu@summit.com
User ID cvelasqu

Ngao
LaDoris
Employee ID 2
Contact Details
Email lngao@summit.com
User ID lngao

Nagayama
Midori
Employee ID 3
Contact Details
Email mnagayam@summit.com
User ID mnagayam

Quick-To-See
Mark
Employee ID 4
Contact Details
Email mquickto@summit.com
User ID mquickto

Ropeburn
Audry
Employee ID 5
Contact Details
Email aropebur@summit.com
User ID aropebur

Before you begin:

It may be helpful to have an understanding of databound hierarchy viewers. For more information, see Creating Databound Hierarchy Viewers.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Data Visualization Components.

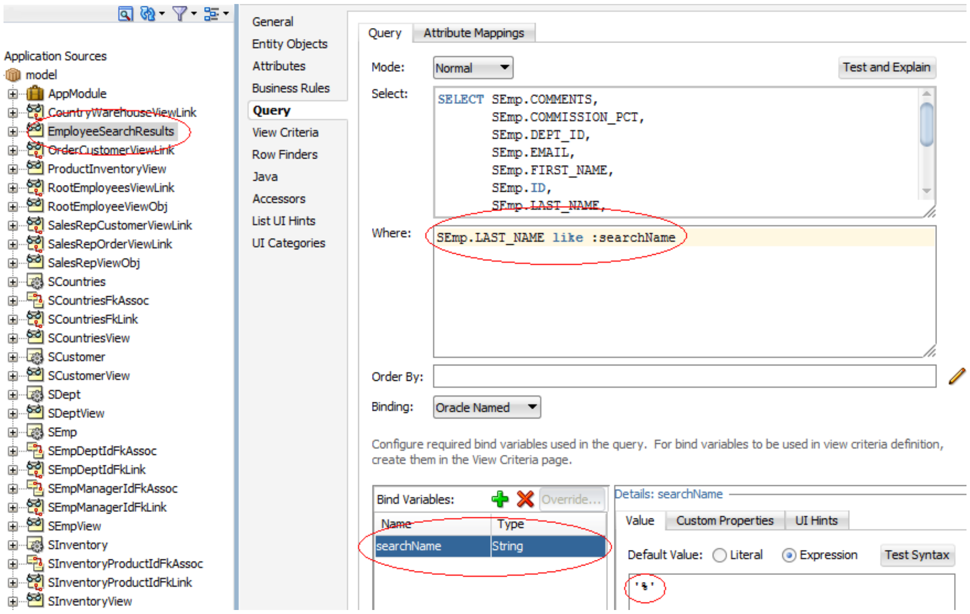
You will need to complete these tasks:

1. Create an application module that contains instances of the view objects that you want in your data model, as described in Creating and Modifying an Application Module.
2. Create a JSF page as described in the How to Create JSF Pages section of *Developing Web User Interfaces with Oracle ADF Faces*.
3. Create a databound hierarchy viewer component as described in How to Create a Hierarchy Viewer Using ADF Data Controls.
4. Verify the query that retrieves the root node in the hierarchy viewer.

   For example, Figure 43-6 shows retrieving the root node by the `Id` column and storing it in the `rootEmpno` bind variable.
5. Create a view object that performs the search.

   For example, Figure 43-19 shows the `EmployeesSearchResults` view object that performs the search based on the `LastName` column in the data collection with a default value of `%` for matching any value, and a comparison value of `like` to enable pattern matching.

Figure 43-19 EmployeeSearchResults View Object



Mode: Normal

Select:

```
SELECT SEmp.COMMENTS,
       SEmp.COMMISSION_PCT,
       SEmp.DEPT_ID,
       SEmp.EMAIL,
       SEmp.FIRST_NAME,
       SEmp.ID,
       SEmp.LAST_NAME,
```

Where:

```
SEmp.LAST_NAME like :searchName
```

Order By:

Binding: Oracle Named

Configure required bind variables used in the query. For bind variables to be used in view criteria definition, create them in the View Criteria page.

Bind Variables:

Name	Type
searchName	String

Details: searchName

Default Value: ○ Literal ● Expression

```
'%'
```

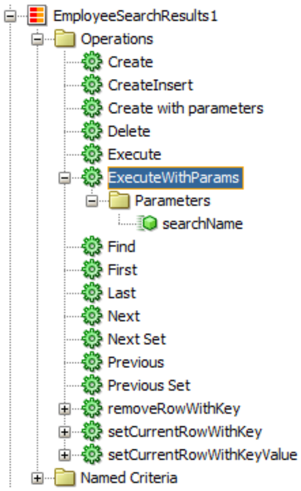
For information about creating a view object, see How to Create an Entity-Based View Object.

To create a databound search with a hierarchy viewer:

1. From the Data Controls panel, select the collection that corresponds to the query results and expand the **Operations** node to display the **ExecuteWithParams** operation.

Figure 43-20 shows the expanded `EmployeeSearchResults1` operations node in the Summit ADF DVT sample application.

Figure 43-20 ExecuteWithParams Operation for Hierarchy Viewer Search Example



2. Drag the **ExecuteWithParams** operation and drop it onto the hierarchy viewer in the visual editor or onto the component in the Structure window.

3. In the Create Hierarchy Viewer Search dialog, use the **Add** icon to specify the list of results to display in the Search Results panel, and specify the following for each result:

   a. **Display Label**: Select the values for the headers of the nodes in the hierarchy. If you select `<default>`, then the text for the header is automatically retrieved from the data binding.

   b. **Value Binding:** Select the columns in the data collection to use for nodes in the tree for the hierarchy viewer.

   c. **Component to Use**: Select the type of component to display in the node. The default is the ADF Output Text component.

   After selecting an existing field, use the arrow icons (**Up**, **Down**, **Top**, **Bottom**) to reorder the results or use the **Delete** icon to delete that result.

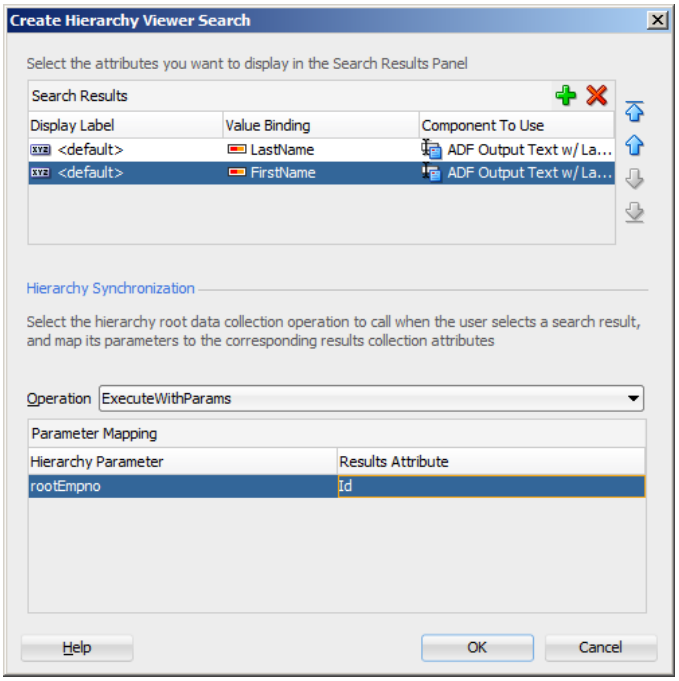
4. In the **Operation** dropdown list, select the hierarchy root data collection to use when a search result is selected. Valid values include:

   - **removeRowWithKey**: Uses the row key as a `String` converted from the value specified by the input field to remove the data object in the bound data collection.
   - **setCurrentRowWithKey**: Sets the row key as a `String` converted from the value specified by the input field. The row key is used to set the currency of the data object in the bound data collection.
   - **setCurrentRowWithKeyValue**: Sets the current object on the iterator, given a key's value.
   - **ExecuteWithParams**: Sets the values to the named bind variables passed as parameters.

5. In the **Parameter Mapping** table, use the dropdown list in the **Results Attribute** column to select the results collection attribute to map to the parameter displayed in the **Hierarchy Parameter** column.

Figure 43-21 shows the Create Hierarchy Viewer Search dialog that appears if you create a hierarchy viewer using data from a data collection named `EmployeesSearchResults1`.

Figure 43-21 Create Hierarchy Viewer Search Dialog



Create Hierarchy Viewer Search

Select the attributes you want to display in the Search Results Panel

Search Results

Display Label	Value Binding	Component To Use
<default>	LastName	ADF Output Text w/ La...
<default>	FirstName	ADF Output Text w/ La...

Hierarchy Synchronization

Select the hierarchy root data collection operation to call when the user selects a search result, and map its parameters to the corresponding results collection attributes

Operation: ExecuteWithParams

Parameter Mapping

Hierarchy Parameter	Results Attribute
rootEmpno	Id

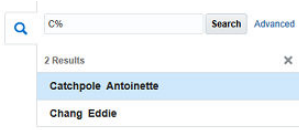
The example below shows the code on the JSF page after you click OK and the `dvt:search` is added to the page as a child of the hierarchy viewer component.

```
<dvt:search value="#{bindings.searchName.inputValue}"
            actionListener="#{bindings.ExecuteWithParams.execute}" id="s19">
  <f:facet name="end">
    <af:link text="Advanced" styleClass="AFHVAdvancedSearchLinkStyle" id="l1"/>
  </f:facet>
  <dvt:searchResults id="sr1"
                     value="#{bindings.EmployeeSearchResults.collectionModel}"
                     emptyText="#{bindings.EmployeeSearchResults.viewable ?
                               'No data to display.' : 'Access Denied.'}"
                     fetchSize="#{bindings.EmployeeSearchResults.rangeSize}"
                     resultListener="#{bindings.ExecuteWithParams1.execute}"
                     var="resultRow">
    <af:setPropertyListener type="action" from="#{resultRow.Id}"
                            to="#{bindings.rootEmpno.inputValue}"/>
    <f:facet name="content">
      <af:panelGroupLayout layout="horizontal" id="pgl27">
        <af:panelLabelAndMessage

label="#{bindings.ExecuteWithParams.hints.LastName.label}"
                        id="plam29">
          <af:outputText value="#{resultRow.LastName}"
```

```
shortDesc="#{bindings.ExecuteWithParams.hints.LastName.tooltip}"
                  id="ot45"/>
        </af:panelLabelAndMessage>
        <af:panelLabelAndMessage

label="#{bindings.ExecuteWithParams.hints.FirstName.label}"
                       id="plam30">
          <af:outputText value="#{resultRow.FirstName}"

shortDesc="#{bindings.ExecuteWithParams.hints.FirstName.tooltip}"
                 id="ot46">
            <af:convertNumber groupingUsed="false"

pattern="#{bindings.ExecuteWithParams.hints.FirstName.format}"/>
          </af:outputText>
        </af:panelLabelAndMessage>
      </af:panelGroupLayout>
    </f:facet>
  </dvt:searchResults>
</dvt:search>
```

At runtime, the search results are displayed in a table by `LastName` and `FirstName`. Figure 43-22 shows the search results panel if a user enters `C%` to search for all employees whose last names begin with `C`.

Figure 43-22   Hierarchy Viewer Search Results Panel



C%

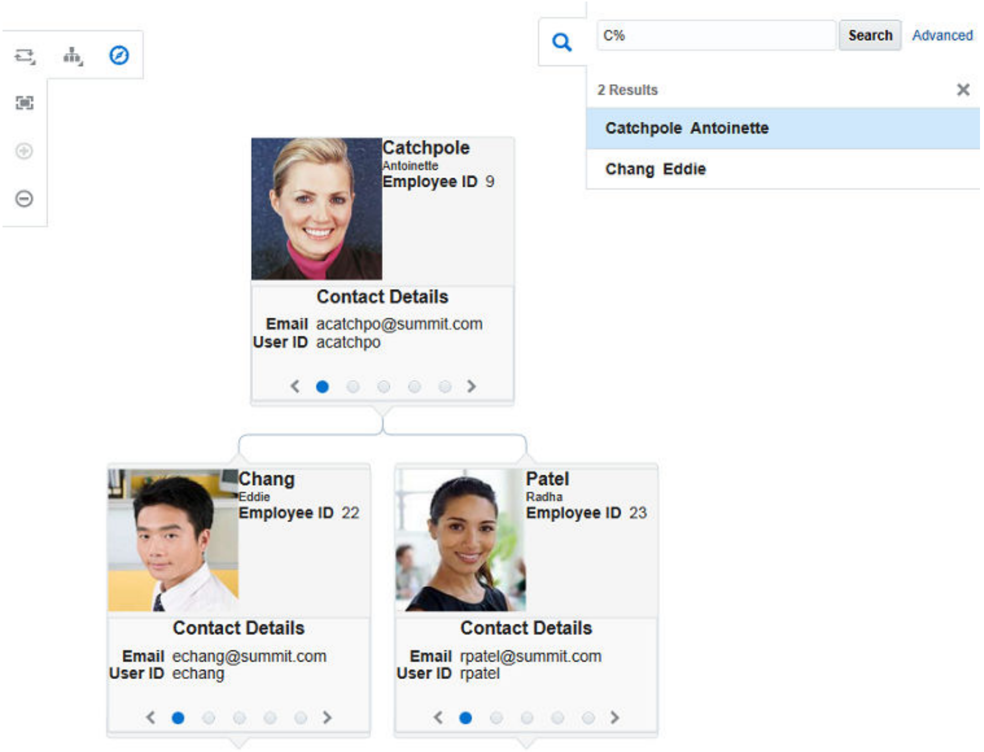
2 Results

Catchpole Antoinette

Chang Eddie

The user can choose one of the search results to display the associated hierarchy viewer node and any children if they exist. Figure 43-23 shows the hierarchy viewer node if the user chooses `Catchpole` from the search results list.

Figure 43-23   Hierarchy Viewer Showing Nodes After Search Selection



C%

2 Results

Catchpole Antoinette

Chang Eddie

Catchpole
Antoinette
Employee ID 9

Contact Details

Email acatchpo@summit.com
User ID acatchpo

Chang
Eddie
Employee ID 22

Contact Details

Email echang@summit.com
User ID echang

Patel
Radha
Employee ID 23

Contact Details

Email rpatel@summit.com
User ID rpatel

Creating Databound Treemaps and Sunbursts

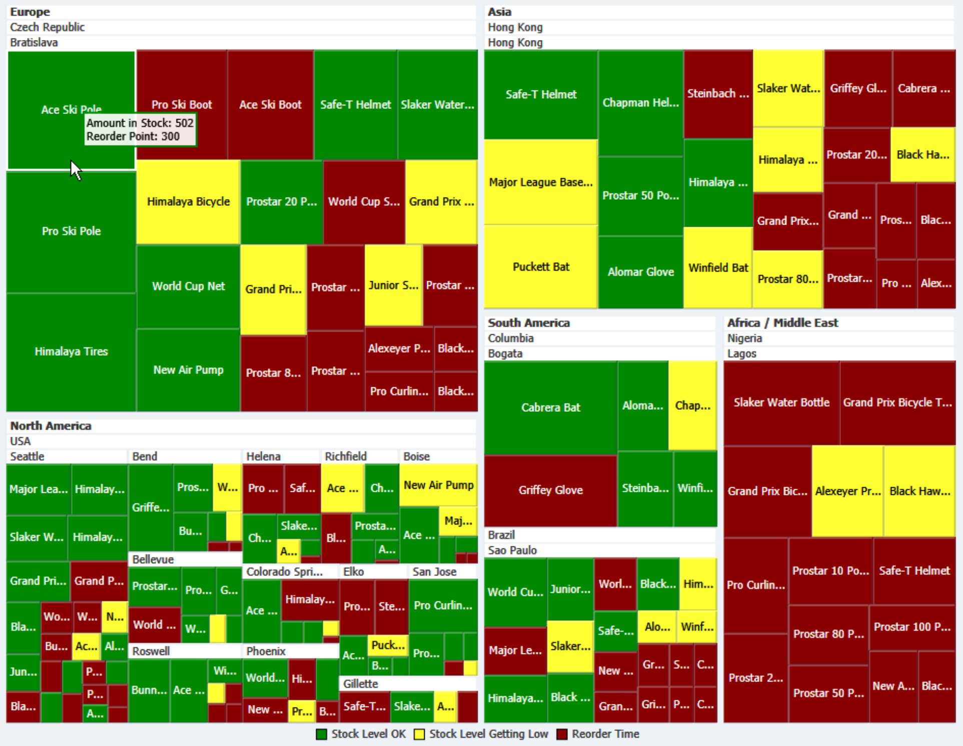
Treemaps and sunbursts are ADF Data Visualization components that display quantitative hierarchical data across two dimensions, represented visually by size and color. This is used to identify data trends easily and quickly.

For example, you can use a treemap or sunburst to display quarterly regional sales and to identify sales trends, using the size of the node to indicate each region's sales volume and the node's color to indicate whether that region's sales increased or decreased over the quarter.

Treemaps and sunbursts use a shape called a `node` to reference the data in the hierarchy. Treemaps display nodes as a set of nested rectangles. Each branch of the tree is given a rectangle, which is then tiled with smaller rectangles representing sub-branches.

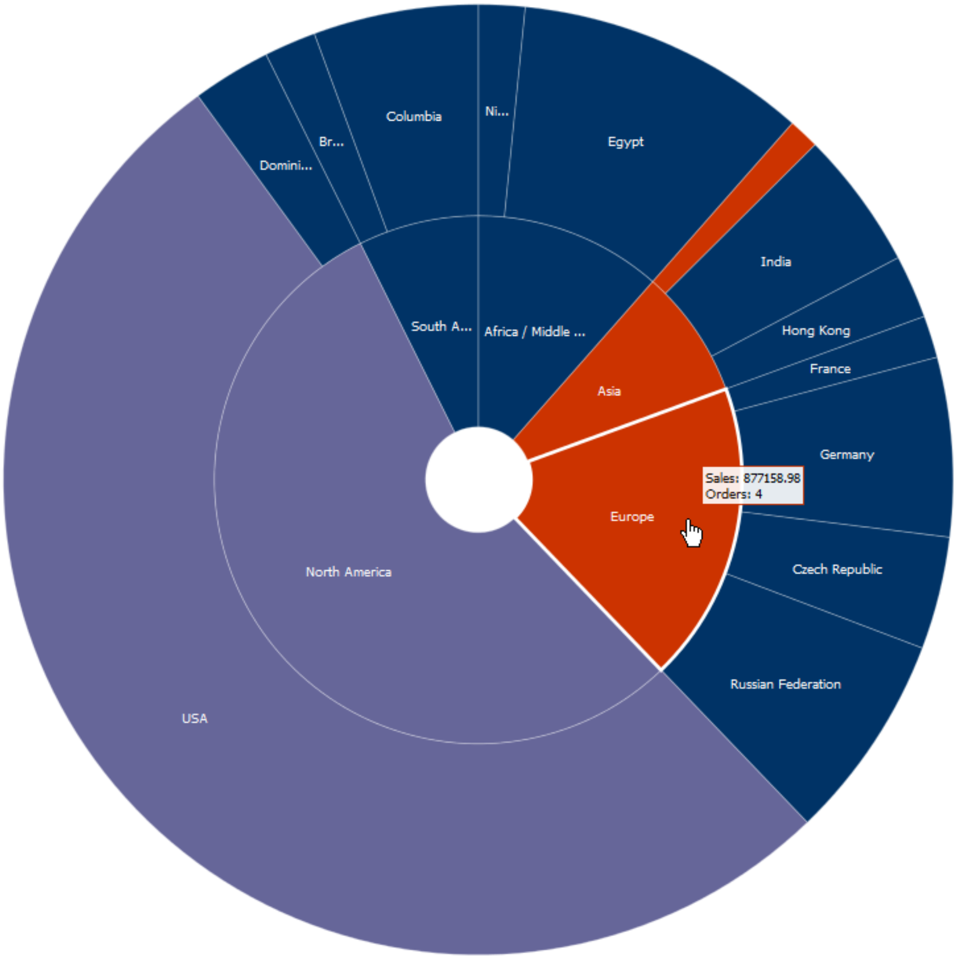
Figure 43-24 shows a treemap displaying the products available at warehouses in the Summit ADF DVT sample application. In this example, the node size represents the amount in stock for each product, and node color represents inventory status.

Figure 43-24 Treemap Showing Product Availability and Inventory Levels



Sunbursts display the nodes in a radial rather than a rectangular layout, with the top of the hierarchy at the center and deeper levels farther away from the center. Figure 43-25 shows a sunburst configured to show sales by region and country, with color used to indicate the number of orders per region and country.

Figure 43-25 Sunburst Showing Product Availability and Inventory Levels



The shape and content of the nodes are configurable, as well as the visual layout of the nodes. For detailed information about treemap and sunburst end user and presentation features, use cases, tag structure, and adding special features, see the Using Treemap and Sunburst Components chapter in *Developing Web User Interfaces with Oracle ADF Faces*.

How to Create Treemaps and Sunbursts Using ADF Data Controls

Treemaps and sunbursts are based on data collections where a master-detail relationship exists between one or more detail collections and a master data collection. Treemaps and sunbursts also require that the following attributes be set in JDeveloper:

- `value`: the size of the node
- `fillColor`: the color of the node
- `label`: a text identifier for the node

The values for the `value` and `label` attributes must be stored in the treemap's or sunburst's data model or in classes and managed beans if you are using UI-first development. You can specify the `fillColor` values in the data model, classes, and managed beans, or declaratively in the Properties window.

In order to configure a treemap or sunburst successfully, ensure that the data adheres to the following rules:

- Each child node can have only one parent node.
- There can be no skipped levels.

Using data controls in Oracle ADF, JDeveloper makes treemap and sunburst creation a declarative task. You drag and drop a data collection from the Data Controls panel that generates one or more nodes onto a JSF page.

Before you begin:

It may be helpful to have an understanding of databound treemaps and sunbursts. For more information, see Creating Databound Treemaps and Sunbursts.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Data Visualization Components.

You will need to complete these tasks:

- Create an application module that contains instances of the view objects that you want in your data model, as described in Creating and Modifying an Application Module.

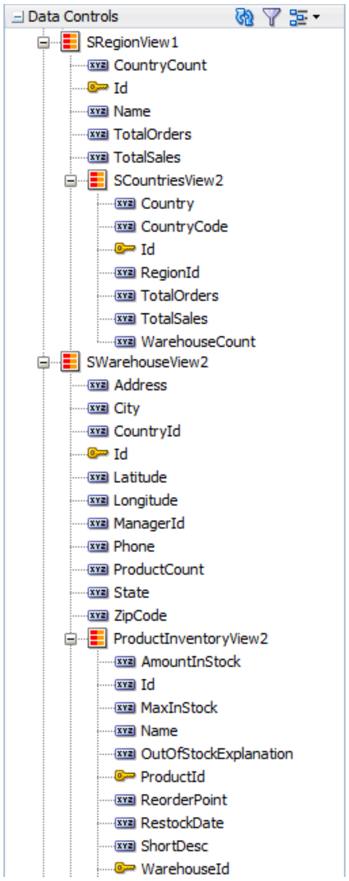
  If you need help working with master-detail relationships, see Displaying Master-Detail Data.
- Create a JSF page as described in the How to Create JSF Pages section of *Developing Web User Interfaces with Oracle ADF Faces*.

To create a treemap or sunburst using the Data Controls panel:

1. From the Data Controls panel, select a collection.

   Figure 43-26 shows the data collection for the treemap in Figure 43-24 and the sunburst in Figure 43-25.

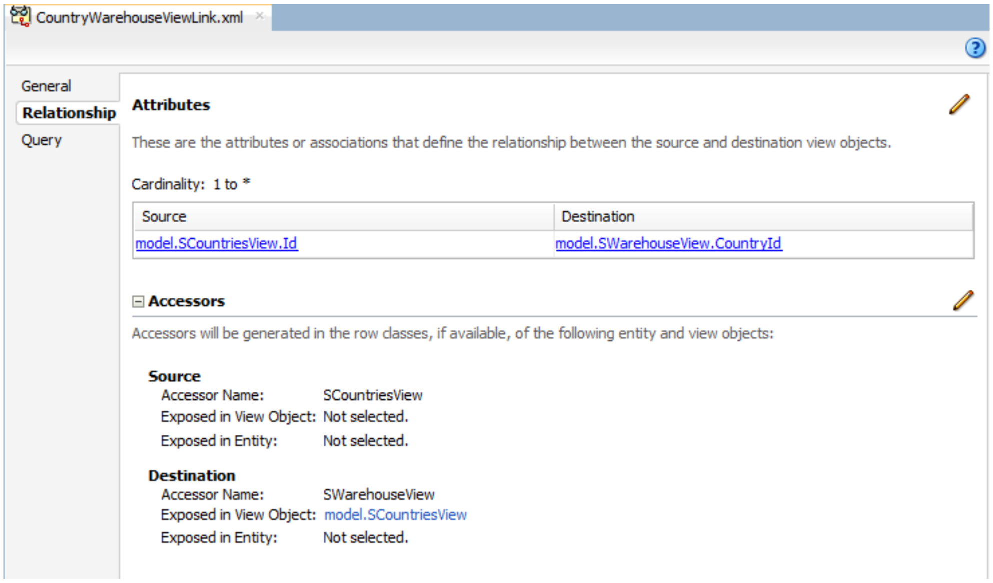
Figure 43-26   Sample Data Collection for Treemap and Sunburst



```
Data Controls
  SRegionView1
    CountryCount
    Id
    Name
    TotalOrders
    TotalSales
    SCountriesView2
      Country
      CountryCode
      Id
      RegionId
      TotalOrders
      TotalSales
      WarehouseCount
  SWarehouseView2
    Address
    City
    CountryId
    Id
    Latitude
    Longitude
    ManagerId
    Phone
    ProductCount
    State
    ZipCode
    ProductInventoryView2
      AmountInStock
      Id
      MaxInStock
      Name
      OutOfStockExplanation
      ProductId
      ReorderPoint
      RestockDate
      ShortDesc
      WarehouseId
```

In the Summit ADF DVT sample, the application uses a view link between the `SCountriesView2` collection and the `SWarehouseView2` collection to establish a hierarchy between the `SRegionView1` and `SWarehouseView2` data collections. Figure 43-27 shows the sample view link.

Figure 43-27 Sample View Link Between Country and Warehouse



Attributes

These are the attributes or associations that define the relationship between the source and destination view objects.

Cardinality: 1 to *

Source	Destination
model.SCountriesView.Id	model.SWarehouseView.CountryId

Accessors

Accessors will be generated in the row classes, if available, of the following entity and view objects:

Source
Accessor Name: SCountriesView
Exposed in View Object: Not selected.
Exposed in Entity: Not selected.

Destination
Accessor Name: SWarehouseView
Exposed in View Object: model.SCountriesView
Exposed in Entity: Not selected.

Table 43-1 shows the data collection and size, label, and color values used to create the treemap in Figure 43-24.

Table 43-1 Sample Values for Treemap Inventory Example

Collection	Size Value	Label Value	Color Value
`SRegionView1`	`CountryCount`	`Name`	`CountryCount`
`SCountriesView2`	`WarehouseCount`	`Country`	`WarehouseCount`
`SWarehouseView2`	`ProductCount`	`City`	`ProductCount`
`ProductInventoryView2`	`AmountInStock`	`Name`	Varies by stock level

The `CountryCount`, `WarehouseCount`, and `ProductCount` attributes are transient attributes that count the number of countries, warehouses, and products in the data collection. For example, the number of countries in a region is calculated by counting the number of unique `Id` values in the `SCountriesView` child collection that match the `RegionId` of the region:

```
SCountriesView.count('Id')
```

Table 43-2 shows the data collection and size, label, and color values used to create the sunburst in Figure 43-25.

Table 43-2 Sample Values for Sunburst Sales Example

Collection	Size Value	Label Value	Color Value
`SRegionView1`	`TotalSales`	`Name`	`TotalOrders`
`SCountriesView2`	`TotalSales`	`Country`	`TotalOrders`

The TotalSales and TotalOrders attributes are transient attributes that calculate the total sales volume and number of orders for each region and country in the data collection. For example, the total sales in a region is calculated by summing the TotalSales in the SCountriesView collection:

```
SCountriesView.sum ('TotalSales')
```

For information about adding transient attributes to view objects, see Adding Calculated and Transient Attributes to a View Object.

2. Drag the collection onto a JSF page and, from the context menu, choose **Treemap** or **Sunburst**.
3. In the Create Treemap dialog or Create Sunburst dialog, in the **Hierarchy** section, select the collections you want to include as nodes in the runtime treemap or sunburst. By default, the root collection is selected.
4. For each collection that you select in the Hierarchy list, enter a value for the following.
   - **Value**: From the dropdown list, select the attribute in the data collection that represents the size value for the sunburst. You can select one of the attributes provided or select **Expression Builder** to enter a JSF EL expression. For help with the Expression Builder dialog, press **F1** or click **Help**.

     For example, to create a treemap or sunburst using the data collection in Figure 43-26, select CountryCount for the **SRegionView1** value to represent the size value for the first node.
   - **Label**: From the dropdown list, select the attribute in the data collection that represents the label you wish to display for the sunburst node label. You can select one of the attributes provided or select **Expression Builder** to enter a JSF EL expression.

     For example, select Name for the **SRegionView1** node label.
   - Optionally, in the **View Area** section, click **New** to configure an attribute group for the node. Use attribute groups if you want the node's display to vary based on color or pattern. You can also specify the node's fill color or pattern in the node's fillColor and fillPattern properties after you create the treemap or sunburst.

     If you do not configure an attribute group to vary by color or specify a fillColor attribute on the node, the node will display in black. By default, no pattern will be displayed on the node.

     In the treemap and sunburst example, attribute groups are configured for each level in the hierarchy.

     To configure an attribute group, enter values for the following:
     - **Group by value**: From the dropdown list, select the attribute in the data collection to group by in the attribute group. You can select one of the attributes provided or select **Expression Builder** to enter a JSF EL expression.

       In the treemap example, the colors used are based on the count metric for each level in the hierarchy. For example, select **CountryCount** as the **Group by value** for the SRegionView1 collection, and the colors displayed on the **Regions** node will vary depending upon the number of countries in a given region.

In the sunburst example, the colors used are based on the number of orders for each level in the hierarchy. For example, select **TotalOrders** as the **Group by value** for the `SRegionView1` collection, and the colors displayed on the **Regions** node will vary depending upon the number of orders in a given region.

- **Area**: From the dropdown list, select **Color** if you want the attribute group to vary by color or select **Pattern** if you want the attribute group to vary by pattern. To vary the attribute group by both color and pattern, select **Select Multiple Attributes** and select both **Color** and **Pattern**. Click **OK**.

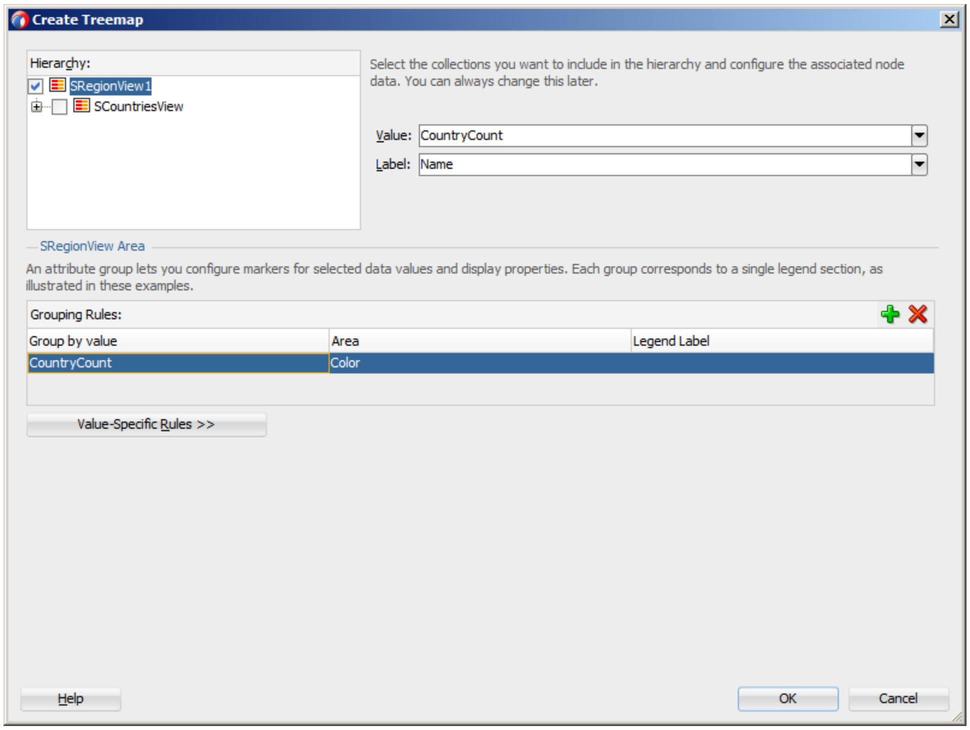
  In the treemap and sunburst examples, color is used for each level of the hierarchy.

- **Legend Label**: From the dropdown list, select the attribute in the data collection to display in the treemap's legend. You can select one of the attributes provided or select **Expression Builder** to enter a JSF EL expression.

  In the treemap and sunburst examples, this value is left blank.

Figure 43-28 shows the Create Treemap dialog completed for the first node in the Summit ADF DVT sample application.

Figure 43-28 Create Treemap Dialog for a Region Node



Hierarchy:
☑ SRegionView1
□ SCountriesView

Select the collections you want to include in the hierarchy and configure the associated node data. You can always change this later.

Value: CountryCount
Label: Name

SRegionView Area

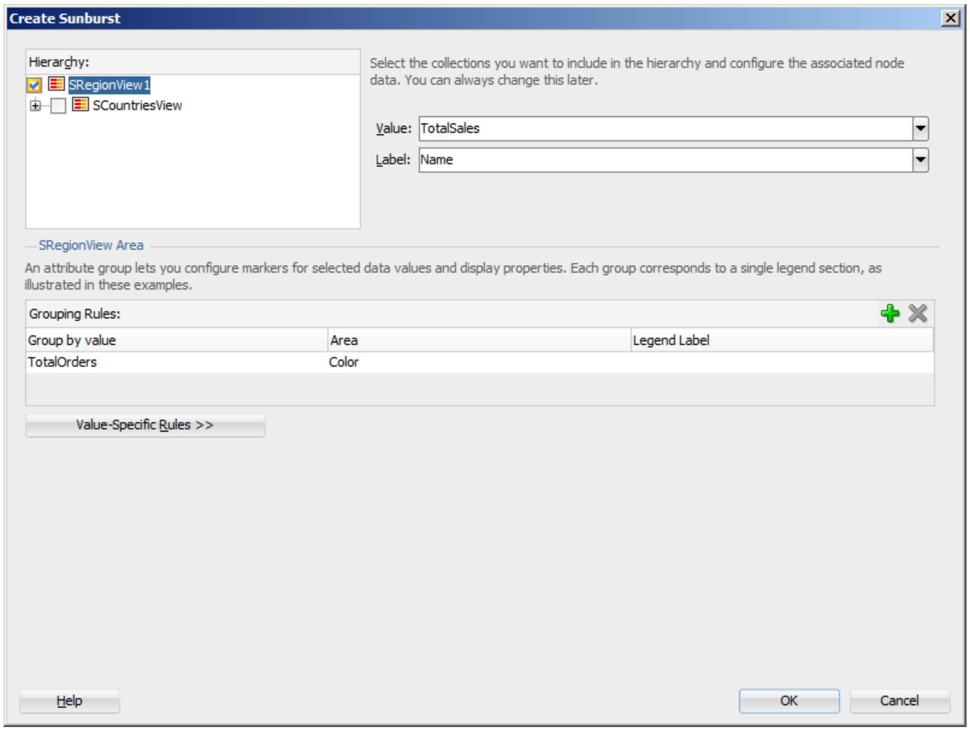
An attribute group lets you configure markers for selected data values and display properties. Each group corresponds to a single legend section, as illustrated in these examples.

Grouping Rules:

Group by value	Area	Legend Label
CountryCount	Color	

Figure 43-29 shows the Create Sunburst dialog completed for the first node in the Summit ADF DVT sample application.

Figure 43-29 Create Sunburst Dialog for a Region Node



Create Sunburst

Hierarchy:

☑ SRegionView1

□ SCountriesView

Select the collections you want to include in the hierarchy and configure the associated node data. You can always change this later.

Value: TotalSales

Label: Name

SRegionView Area

An attribute group lets you configure markers for selected data values and display properties. Each group corresponds to a single legend section, as illustrated in these examples.

Grouping Rules:

Group by value	Area	Legend Label
TotalOrders	Color	

- Optionally, click **Value-Specific Rules** to expand the attribute group dialog to specify a match or exception rule. Use match rules to specify colors or patterns for simple true or false conditions or when you want to match a specific value. Use exception rules when you want to specify a color or pattern when the node's grouped-by value meets a specific condition.

  To specify a match rule, in the **Match Rules** section, click **New** and enter values for the following:

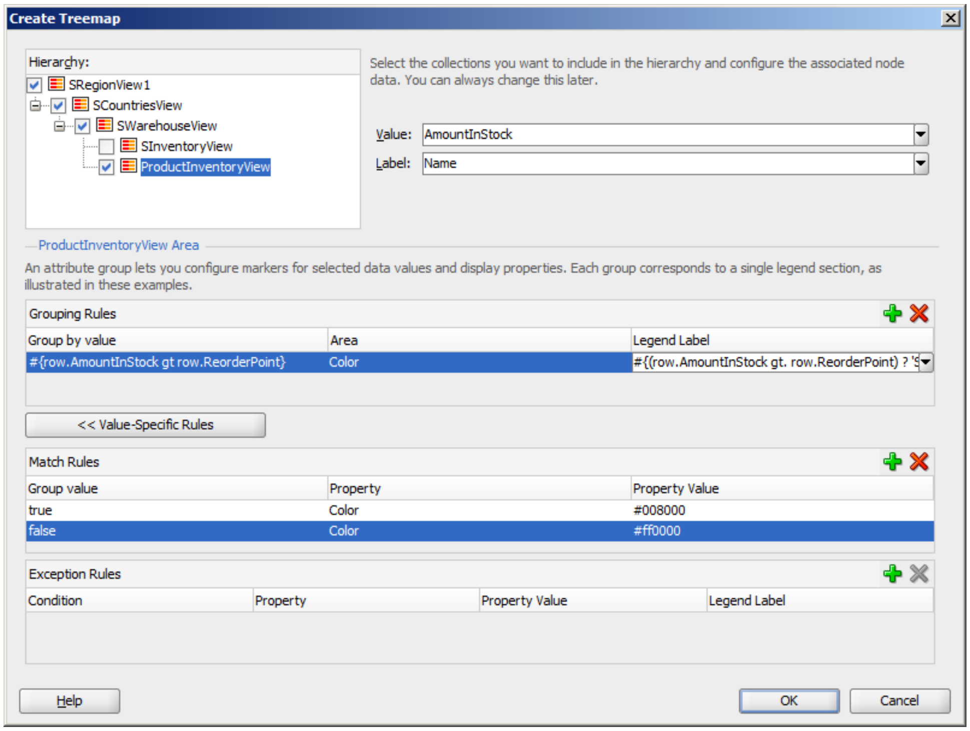
  – **Group Value**: Enter the category value for the match. This can be a string that represents a category or you can set this to `true` or `false`. If you set this to true or false, the **Group by value** field must contain an EL expression that evaluates to true or false as in the following example:`#{row.AmountInStock gt row.ReorderPoint}`.

  – **Property**: From the dropdown list, select **Color** if you want the node to vary by color or select **Pattern** if you want the attribute node to vary by pattern.

  – **Property Value**: From the dropdown list, select the color or pattern to display when the node's value matches the **Group Value**. For color, you can select one of the provided values or select **Custom Color** to enter a custom color in the Select Custom Color dialog. For pattern, you must select one of the provided values.

  Figure 43-30 shows the Create Treemap dialog for a data collection using match rules for the bottom level of the hierarchy.

Figure 43-30 Treemap Showing Match Rules for an Attribute Group



Create Treemap

Hierarchy:

- ☑ SRegionView1
  - ☑ SCountriesView
    - ☑ SWarehouseView
      - □ SInventoryView
      - ☑ ProductInventoryView

Select the collections you want to include in the hierarchy and configure the associated node data. You can always change this later.

Value: AmountInStock

Label: Name

ProductInventoryView Area

An attribute group lets you configure markers for selected data values and display properties. Each group corresponds to a single legend section, as illustrated in these examples.

Grouping Rules

Group by value	Area	Legend Label
#{row.AmountInStock gt row.ReorderPoint}	Color	#{(row.AmountInStock gt. row.ReorderPoint) ? 'S

<< Value-Specific Rules

Match Rules

Group value	Property	Property Value
true	Color	#008000
false	Color	#ff0000

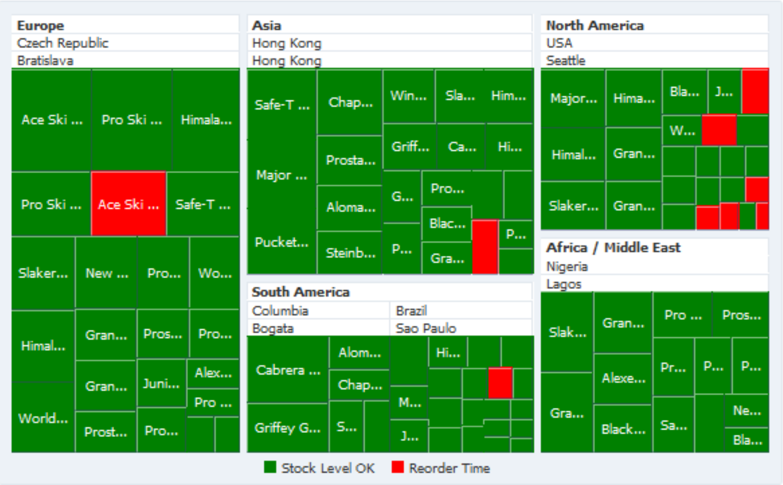
Exception Rules

Condition	Property	Property Value	Legend Label

In this example, the treemap nodes will display in green (RGB hexadecimal `#008000`) when the inventory levels are acceptable and in red (RBG hexadecimal `#ff0000`) when the inventory level is at or below the product's reorder level. The attribute group's **Label** field contains the details for the legend display. In this case, the field contains an expression that determines the legend label based on an item's amount in stock: `#{(row.AmountInStock gt row.ReorderPoint) ? 'Stock Level OK': 'Reorder Time'}`.

Figure 43-31 shows the runtime treemap.

Figure 43-31 Treemap Showing Match Rules at Runtime



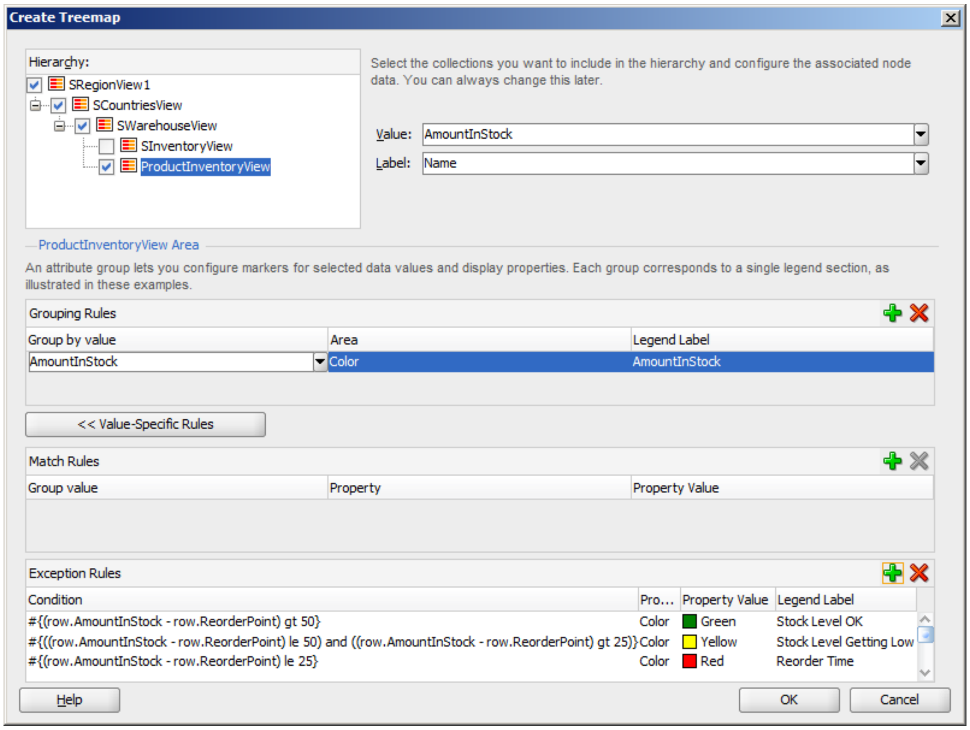
To specify an exception rule, in the **Exception Rules** section, click **New** and enter values for the following:

– **Condition**: Enter a JSF EL expression that evaluates to true or false. You can enter the expression directly in the **Condition** field or select **Expression Builder** to enter the JSF EL expression.

  In the Summit ADF DVT sample application, three exception rules are defined for the bottom level of the hierarchy. When the product's inventory level is well above the reorder level, the treemap node in Figure 43-24 and the sunburst node in Figure 43-25 display in green. The treemap and sunburst node display in yellow when the inventory level is close to the reorder point and display in red when the inventory level is at or very near the reorder point.

  Figure 43-32 shows the Create Treemap dialog for the Summit ADF DVT sample data collection using exception rules for the bottom level of the hierarchy.

Figure 43-32 Treemap Showing Exception Rules Creation Dialog



Create Treemap

Hierarchy:
- ☑ SRegionView1
  - ☑ SCountriesView
    - ☑ SWarehouseView
      - □ SInventoryView
      - ☑ ProductInventoryView

Select the collections you want to include in the hierarchy and configure the associated node data. You can always change this later.

Value: AmountInStock

Label: Name

ProductInventoryView Area

An attribute group lets you configure markers for selected data values and display properties. Each group corresponds to a single legend section, as illustrated in these examples.

Grouping Rules

Group by value	Area	Legend Label
AmountInStock	Color	AmountInStock

<< Value-Specific Rules

Match Rules

Group value	Property	Property Value

Exception Rules

Condition	Pro...	Property Value	Legend Label
#{(row.AmountInStock - row.ReorderPoint) gt 50}	Color	🟩 Green	Stock Level OK
#{((row.AmountInStock - row.ReorderPoint) le 50) and ((row.AmountInStock - row.ReorderPoint) gt 25)}	Color	🟨 Yellow	Stock Level Getting Low
#{(row.AmountInStock - row.ReorderPoint) le 25}	Color	🟥 Red	Reorder Time

> **Note:**
>
> In this example, the condition is referencing the `ReorderPoint` attribute from the data collection. To use this attribute, you must manually add it to the binding for the treemap after the treemap is created. For more information, see What Happens When You Create a Databound Treemap or Sunburst.

- **Property**: From the dropdown list, select **Color** if you want the node to vary by color. Select **Pattern** if you want the node to vary by pattern.
- **Property Value**: From the dropdown list, select the color or pattern to display when the node's value meets the condition you specified in the **Condition** field. For color, you can select one of the provided values or select **Custom Color** to enter a custom color in the Select Custom Color dialog. For pattern, you must select one of the provided values.
- **Legend Label**: From the dropdown list, select **Select Text Resource** to select a text resource to be used for the legend label. You can also enter text in this field or select **Expression Builder** to enter a JSF EL expression.

5. Click **OK**.

What Happens When You Create a Databound Treemap or Sunburst

Creating a treemap or sunburst from Data Controls panel has the following effect:

- Creates the bindings for the sunburst or treemap in the page definition file of the JSF page
- Adds the necessary tags to the JSF page for the `treemap` or `sunburst` component

The example below displays bindings that JDeveloper generated for a `treemap` component using the data collection in Figure 43-26.

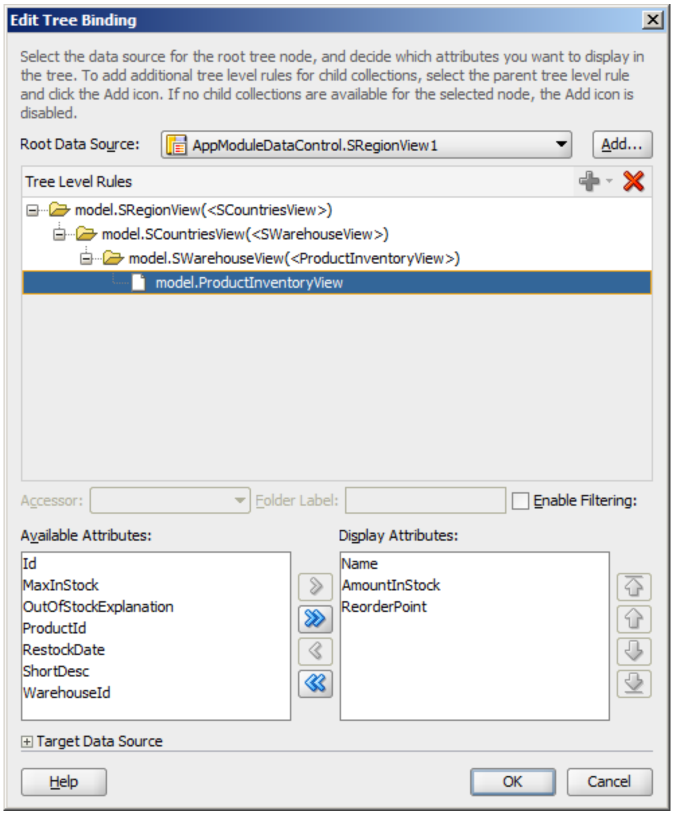
```
<executables>
  <variableIterator id="variables"/>
  <iterator Binds="SRegionView1" RangeSize="-1"
            DataControl="AppModuleDataControl" id="SRegionView1Iterator"/>
</executables>
<bindings>
  <tree IterBinding="SRegionView1Iterator" id="SRegionView1">
    <nodeDefinition DefName="model.SRegionView" Name="SRegionView10">
      <AttrNames>
        <Item Value="CountryCount"/>
        <Item Value="Name"/>
      </AttrNames>
      <Accessors>
        <Item Value="SCountriesView"/>
      </Accessors>
    </nodeDefinition>
    <nodeDefinition DefName="model.SCountriesView" Name="SRegionView11">
      <AttrNames>
        <Item Value="WarehouseCount"/>
        <Item Value="Country"/>
      </AttrNames>
      <Accessors>
        <Item Value="SWarehouseView"/>
      </Accessors>
    </nodeDefinition>
    <nodeDefinition DefName="model.SWarehouseView" Name="SRegionView12">
      <AttrNames>
        <Item Value="ProductCount"/>
        <Item Value="City"/>
      </AttrNames>
      <Accessors>
        <Item Value="ProductInventoryView"/>
      </Accessors>
    </nodeDefinition>
    <nodeDefinition DefName="model.ProductInventoryView" Name="SRegionView13">
      <AttrNames>
        <Item Value="Name"/>
        <Item Value="AmountInStock"/>
      </AttrNames>
    </nodeDefinition>
  </tree>
</bindings>
```

The rules for populating the nodes of the treemap or sunburst are defined in node definitions. The example shows that four node definitions were generated, one for each level of the hierarchy. Each of these node definitions references a view object and the attributes specified in the Create Treemap or Create Sunburst dialog. Each node definition also references an accessor for the next level of the hierarchy if the next level exists.

Attributes that were not specified in the **Value**, **Name**, or **Group By Value** fields during creation will not be included in the binding. If you need to reference another attribute in the data collection, you must add it manually to the binding. For example, in the Summit ADF DVT sample, the attribute exception rules calculate the colors based on the difference between the amount in stock (`AmountInStock`) and the reorder point (`ReorderPoint`) for a given product. For the EL expression to evaluate properly, you must add the `ReorderPoint` attribute to the binding definition for the treemap or sunburst.

To add the binding, in the Structure window, right-click the **dvt:treemap** or **dvt:sunburst** component and choose **Go to Binding**. Choose **Edit** in the **Binding** section to add the `ReorderPoint` attribute. Figure 43-33 shows the completed Edit Tree Binding dialog for the Summit ADF DVT treemap with `ReorderPoint` added to the display attributes.

Figure 43-33 Edit Tree Binding Dialog



Select the data source for the root tree node, and decide which attributes you want to display in the tree. To add additional tree level rules for child collections, select the parent tree level rule and click the Add icon. If no child collections are available for the selected node, the Add icon is disabled.

Root Data Source: AppModuleDataControl.SRegionView1

Tree Level Rules

```
model.SRegionView(<SCountriesView>)
  model.SCountriesView(<SWarehouseView>)
    model.SWarehouseView(<ProductInventoryView>)
      model.ProductInventoryView
```

Accessor: Folder Label: □ Enable Filtering:

Available Attributes: Id, MaxInStock, OutOfStockExplanation, ProductId, RestockDate, ShortDesc, WarehouseId

Display Attributes: Name, AmountInStock, ReorderPoint

⊞ Target Data Source

After you click **OK**, the binding will be updated with the added attribute. The example below shows the revised binding for the `SRegionView13` node definition.

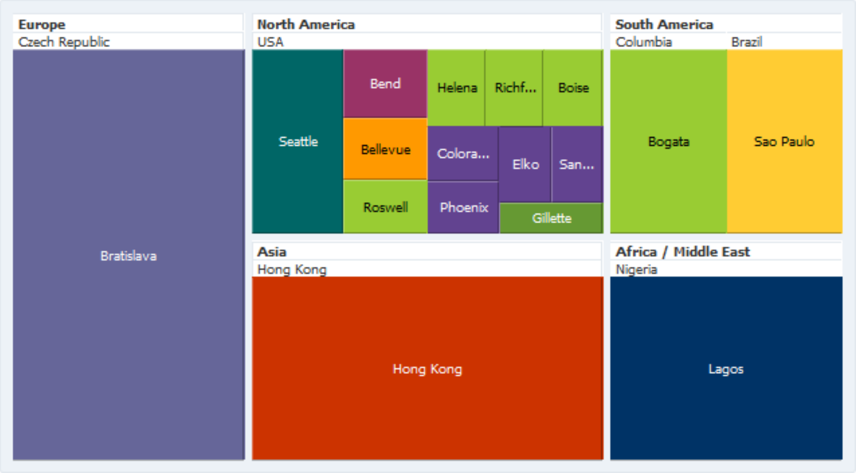
```
<nodeDefinition DefName="model.ProductInventoryView" Name="SRegionView13">
  <AttrNames>
    <Item Value="Name"/>
    <Item Value="AmountInStock"/>
    <Item Value="ReorderPoint"/>
  </AttrNames>
</nodeDefinition>
```

Once you create your treemap or sunburst, you can modify the value, label, and attribute group for each node using the Create Treemap or Create Sunburst dialog. To open the dialog, use the **Edit** icon in the Properties window for the `treemap` or `sunburst` component. You can also customize the values directly in the code or by setting values in the Properties window.

What Happens at Runtime: How Databound Sunbursts or Treemaps Are Initially Displayed

By default, a sunburst or treemap displays only the first two levels of the hierarchy and will not display a legend until you configure the `legendSource` attribute on the sunburst or treemap. Figure 43-34 shows the treemap that is displayed at runtime if you create the treemap using the data collection in Figure 43-26.

Figure 43-34   Databound Treemap Example Before Configuration



After completing the Create Treemap or Create Sunburst dialog, you can use the Properties window to configure the treemap or sunburst level display and `legendSource` attribute. You can also specify settings for other treemap or sunburst attributes and use the child tags associated with the sunburst and treemap tags to customize the components further.

The example below shows the code on the JSF page code for the treemap in Figure 43-24. In this example, the databound treemap is configured to display all levels in the hierarchy, a legend, a tooltip when the user hovers the mouse over a node, and a summary for accessibility. The code related to the additional configuration is highlighted in bold font.

```
<dvt:treemap id="t1" value="#{bindings.SRegionView1.treeModel}"
             var="row" displayLevelsChildren="3"
             legendSource="ag1" summary="Sample Treemap">
  <af:switcher facetName="#{row.hierTypeBinding.name}" id="s1">
    <f:facet name="SRegionView11">
      <dvt:treemapNode value="#{row.WarehouseCount}" label="#{row.Country}"
                       id="tn1">
        <dvt:attributeGroups id="ag2" value="#{row.WarehouseCount}"
type="color"/>
      </dvt:treemapNode>
    </f:facet>
    <f:facet name="SRegionView10">
      <dvt:treemapNode value="#{row.CountryCount}" label="#{row.Name}" id="tn2">
        <dvt:attributeGroups id="ag3" value="#{row.CountryCount}" type="color"/>
      </dvt:treemapNode>
    </f:facet>
    <f:facet name="SRegionView13">
      <dvt:treemapNode value="#{row.AmountInStock}" label="#{row.Name}" id="tn3"
                       shortDesc="Amount in Stock:
                                  #{row.AmountInStock}&lt;br/>Reorder Point:
                                  #{row.ReorderPoint}">
        <dvt:attributeGroups value="#{row.AmountInStock}" type="color" id="ag1">
          <dvt:attributeExceptionRule id="aer1"
                                      condition="#{(row.AmountInStock -
                                                    row.ReorderPoint) gt 50}"
                                      label="Stock OK">
            <f:attribute name="color" value="#008800"/>
          </dvt:attributeExceptionRule>
          <dvt:attributeExceptionRule id="aer2"
                condition="#{((row.AmountInStock - row.ReorderPoint) le 50) and
                             ((row.AmountInStock - row.ReorderPoint) gt 25)}"
                label="Stock Level Getting Low">
            <f:attribute name="color" value="#FFFF33"/>
          </dvt:attributeExceptionRule>
          <dvt:attributeExceptionRule id="aer3"
                condition="#{(row.AmountInStock - row.ReorderPoint) le 25}"
                label="Reorder Time">
            <f:attribute name="color" value="#880000"/>
          </dvt:attributeExceptionRule>
        </dvt:attributeGroups>
      </dvt:treemapNode>
    </f:facet>
    <f:facet name="SRegionView12">
      <dvt:treemapNode value="#{row.ProductCount}" label="#{row.City}" id="tn4">
        <dvt:attributeGroups id="ag4" value="#{row.ProductCount}" type="color"/>
      </dvt:treemapNode>
    </f:facet>
  </af:switcher>
</dvt:treemap>
```

The example below shows the code on the JSF page code for the sunburst in Figure 43-25. In this example, the databound sunburst is configured with a summary and a tooltip that displays when the user hovers the mouse over a node.

```
<dvt:sunburst id="t1" value="#{bindings.SRegionView1.collectionModel}"
              var="row" summary="Sunburst Demo">
  <af:switcher facetName="#{row.hierTypeBinding.name}" id="s1">
    <f:facet name="SRegionView10">
     <dvt:sunburstNode value="#{row.TotalSales}" label="#{row.Name}" id="sn1"
          shortDesc="Sales: #{row.TotalSales}&lt;br/>Orders:
#{row.TotalOrders}">
       <dvt:attributeGroups value="#{row.TotalOrders}" type="color" id="ag2"/>
     </dvt:sunburstNode>
   </f:facet>
   <f:facet name="SRegionView11">
     <dvt:sunburstNode value="#{row.TotalSales}" label="#{row.Country}" id="sn2"
          shortDesc="Sales: #{row.TotalSales}&lt;br/>Orders:
#{row.TotalOrders}">
       <dvt:attributeGroups value="#{row.TotalOrders}" type="color" id="ag1"/>
     </dvt:sunburstNode>
   </f:facet>
 </af:switcher>
</dvt:sunburst>
```

For information about configuring the sunburst or treemap level display, `legendSource` attribute, tooltips, additional attributes, or child tags, see the Using Treemap and Sunburst Components chapter in *Developing Web User Interfaces with Oracle ADF Faces*.

44

Creating Databound Diagram Components

This chapter describes how to how to create a diagram from data modeled with ADF Business Components, using ADF data controls and ADF Faces components in a Fusion web application. Specifically, it describes how you can use ADF Data Visualization `diagram` components to create diagrams that visually represent business data. It describes how to use ADF data controls to create these components with data-first development.
If you are designing your page using simple UI-first development, then you can add the diagram to your page and configure the data bindings later. For information about the data requirements, tag structure, and options for customizing the look and behavior of the diagram component, see Using Diagram Components in *Developing Web User Interfaces with Oracle ADF Faces*.

This chapter includes the following sections:

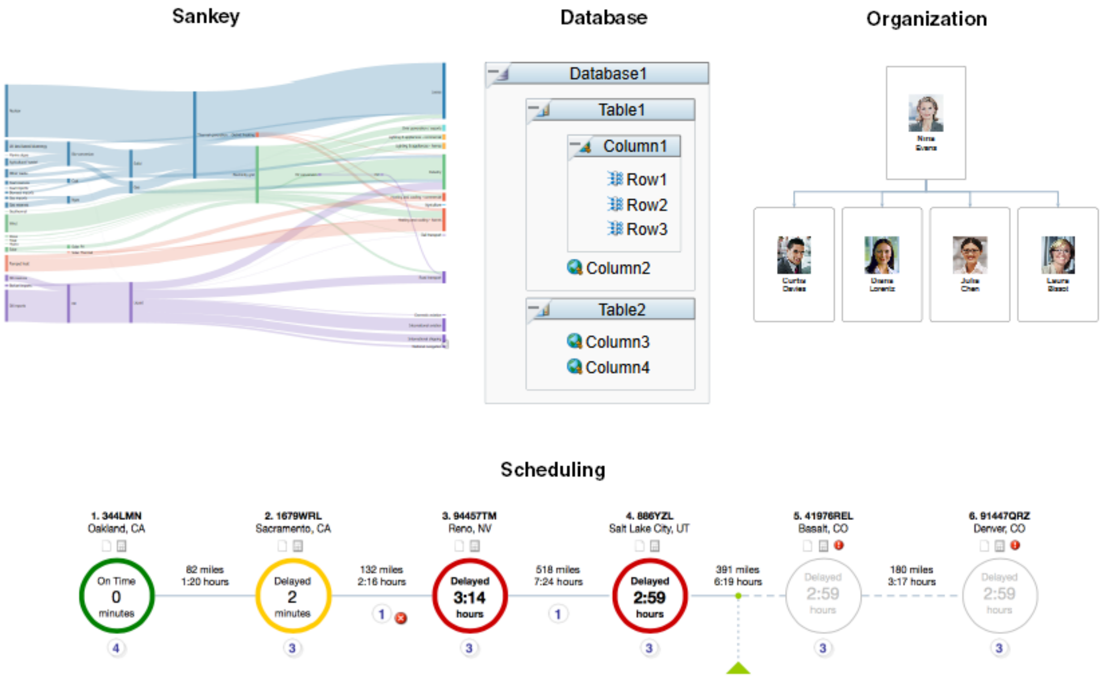
About ADF Data Visualization Diagram Components

The DVT `diagram` component is a versatile tool that produces an interactive component that you can use to model, represent, and visualize information using a shape called a node to represent data and links to represent relationships between the nodes. Use diagrams when you want to highlight both the data objects and the relationship between them.

ADF Data Visualization components provide extensive graphical and tabular capabilities for visually displaying and analyzing business data. Each component needs to be bound to data before it can be rendered since the appearance of the components is dictated by the data that is displayed.

Figure 44-1 shows four runtime images of diagram components: a Sankey flow diagram, database table design, employee organization chart, and a scheduling diagram. In these examples, the nodes are the processes, database objects, employees, and scheduled stops. The links show the relationship between each process in the Sankey flow diagram, employee in the organization chart, and sequence of stops in the scheduling diagram.

Figure 44-1 Diagram Component Examples



The prefix `dvt:` occurs at the beginning of each data visualization component name indicating that the component belongs to the ADF Data Visualization Tools (DVT) tag library.

End User and Presentation Features

Visually compelling data visualization components enable end users to understand and analyze complex business data. The components are rich in features that provide out-of-the-box interactivity support. For detailed descriptions of the end user and presentation features for the DVT diagram component, see End User and Presentation Features in *Developing Web User Interfaces with Oracle ADF Faces*.

Data Visualization Components Use Cases and Examples

For detailed descriptions of Diagram component use cases and examples, see Diagram Use Cases and Examples in *Developing Web User Interfaces with Oracle ADF Faces*.

Additional Functionality for Data Visualization Components

You may find it helpful to understand other Oracle ADF features before you data bind your data visualization components. Additionally, once you have added a data visualization component to your page, you may find that you need to add functionality such as validation and accessibility. Following are links to other functionality that data visualization components use:

- Partial page rendering: You may want a data visualization component to refresh to show new data based on an action taken on another component on the page. For more information, see Rerendering Partial Page Content in *Developing Web User Interfaces with Oracle ADF Faces*.

- Personalization: Users can change the way the data visualization components display at runtime, those values will not be retained once the user leaves the page unless you configure your application to allow user customization. For more information, see Allowing User Customization on JSF Pages in *Developing Web User Interfaces with Oracle ADF Faces*.
- Accessibility: You can make your data visualization components accessible. For more information, see Developing Accessible ADF Faces Pages in *Developing Web User Interfaces with Oracle ADF Faces*.
- Skins and styles: You can customize the appearance of data visualization components using an ADF skin that you apply to the application or by applying CSS style properties directly using a style-related property (`styleClass` or `inlineStyle`). For more information, see Customizing the Appearance Using Styles and Skins in *Developing Web User Interfaces with Oracle ADF Faces*.
- Placeholder data controls: If you know the data visualization components on your page will eventually use ADF data binding, but you need to develop the pages before the data controls are ready, then you should consider using placeholder data controls, rather than manually binding the components. Using placeholder data controls will provide the same declarative development experience as using developed data controls. For more information, see Designing a Page Using Placeholder Data Controls.

Creating Databound Diagram Components

Databound DVT diagram components require a collection of node objects to represent the data and, optionally, a collection of link objects to represent the relationships between the nodes.

You can use objects that implement a List interface, such as a `java.util.ArrayList` object, or a collection model, such as a `org.apache.myfaces.trinidad.model.CollectionModel`.

Databound diagrams also require a client layout configuration defined in a JavaScript method that you add to the application as a feature. Client layouts specify how to lay out the nodes and links on a page. The diagram component includes a default layout that you can use and modify as needed. The default `forceDirectedLayout` positions the diagram nodes so that all the links are of more or less equal length, and there are as few crossing links as possible.

> **Note:**
>
> You can also provide your own client layout configuration in a JavaScript object that you add as an application feature. For additional information, see Using the Diagram Layout Framework in *Developing Web User Interfaces with Oracle ADF Faces*.

The Create Diagram wizard provides declarative support for creating the diagram and binding it to data. In the five wizard pages you can:

1. Configure the diagram's client layout.

   You can choose the default client `forceDirectedLayout` feature, specify your own layout, or choose no layout.

2. Specify the node and link data, including:

   - The collections to use for the node and link data
   - The attribute that specifies the node's unique identifier (id)
   - Optionally, the attribute in the node collection that contains the container id to identify child nodes that are contained by the parent node
   - The attributes in the link collection that specify the starting and ending node ids for a link

3. Optionally, configure the display characteristics of the node's label and marker.

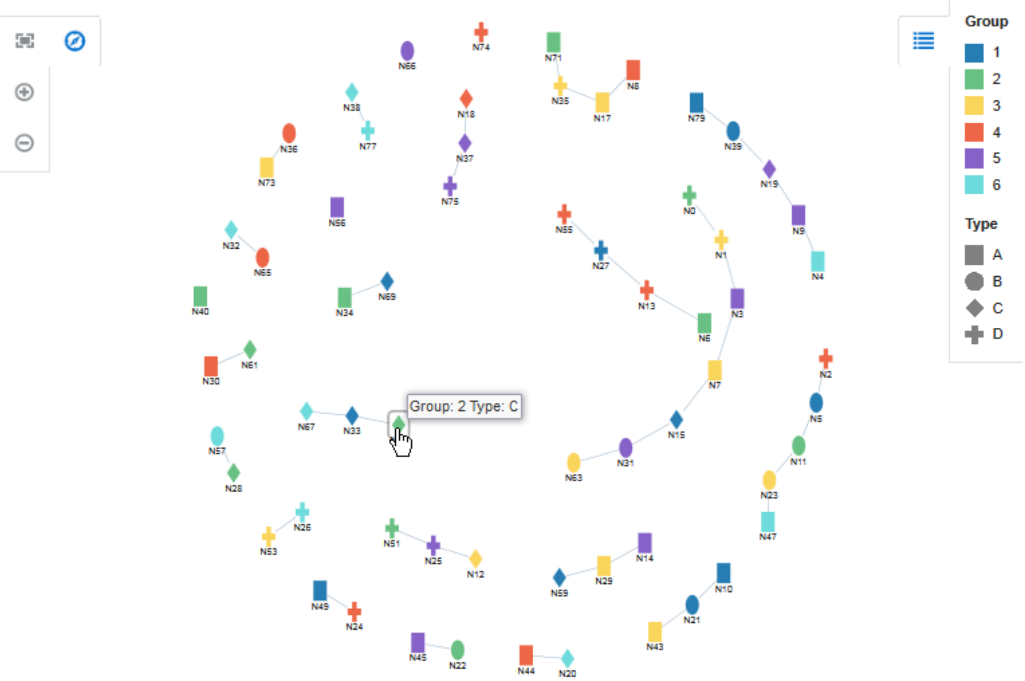
   Diagram nodes use the `dvt:marker` component to configure many of the node's display characteristics, including color, pattern, shape, border style, and color.

4. Optionally, configure attribute groups to group nodes by color, shape, pattern, opacity, scale X, or scale Y according to a specified attribute.

5. Optionally, configure the display characteristics of the link's label, style, color, width, beginning connector, and end connector.

Figure 44-2 shows a diagram configured to use the default client layout. In this example, the diagram is configured to use attribute groups to group the nodes by the values in the node's `NodeGroup` and `NodeType` attributes.

Figure 44-2 Diagram Example With Attribute Groups



After you complete the Create Diagram wizard and the diagram is added to your page, you can further customize the diagram using the Properties window. For additional information, see Using Diagram Components in *Developing Web User Interfaces with Oracle ADF Faces*.

> **Note:**
>
> The ADF Data Visualization Tools (DVT) component collection includes other components that you can also use to model organization charts, time lines, etc.
>
> For example, you can also use the DVT hierarchy viewer component to display an organization chart bound to employee data. Hierarchy viewers also use nodes and links to display data, but the component automatically creates the links based on the values in the node's data collection, and you do not need to supply a separate data collection for the links.
>
> For additional information about other DVT components and typical use cases, see Introduction to Data Visualization Components in *Developing Web User Interfaces with Oracle ADF Faces*.

How to Create a Diagram Using ADF Data Controls

To create a DVT diagram using data controls, bind the `dvt:diagram` component to a collection that contains the node data and a collection that contains the links that represent the relationships between the nodes. JDeveloper allows you to do this declaratively by dragging and dropping a collection from the Data Controls panel.

> **Tip:**
>
> You can also create the diagram by dragging the diagram from the Components window.

Before you begin:

It may be helpful to have an understanding of databound diagrams. For more information, see About ADF Data Visualization Diagram Components.

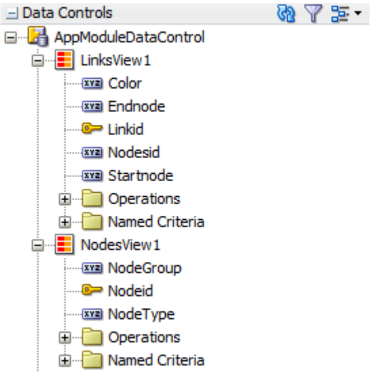
You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Data Visualization Components.

You will need to complete these tasks:

- Create an application module that contains instances of the view objects that you want in your data model for the diagram, as described in Creating and Modifying an Application Module.

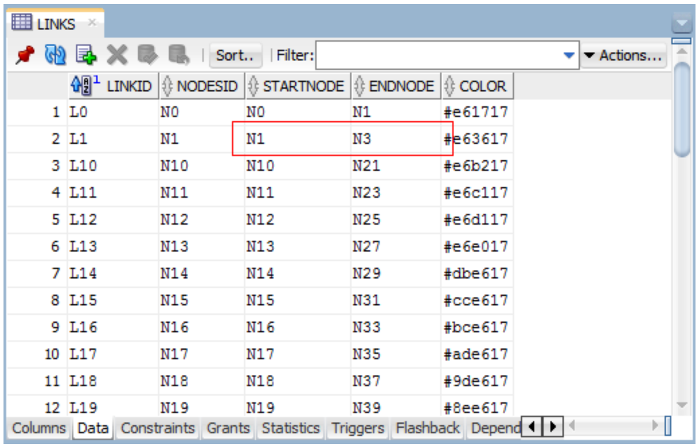
  Figure 44-3 shows the data control for the diagram displayed in Figure 44-2. In this example, `NodesView1` contains the data collection for the diagram nodes, and `LinksView1` contains the data collection for the diagram links.

Figure 44-3 Diagram Node and Link Data Collections Example



The `Startnode` and `Endnode` attributes in the `LinksView1` data collection contain references to the `Nodeid` attribute in the `NodesView1` data collection. Figure 44-4 shows a portion of the link collection's data in the Object Viewer.

Figure 44-4 Diagram Sample Link Data



	LINKID	NODESID	STARTNODE	ENDNODE	COLOR
1	L0	N0	N0	N1	#e61717
2	L1	N1	N1	N3	#e63617
3	L10	N10	N10	N21	#e6b217
4	L11	N11	N11	N23	#e6c117
5	L12	N12	N12	N25	#e6d117
6	L13	N13	N13	N27	#e6e017
7	L14	N14	N14	N29	#dbe617
8	L15	N15	N15	N31	#cce617
9	L16	N16	N16	N33	#bce617
10	L17	N17	N17	N35	#ade617
11	L18	N18	N18	N37	#9de617
12	L19	N19	N19	N39	#8ee617

- Create a JSF page as described in the How to Create JSF Pages section of *Developing Web User Interfaces with Oracle ADF Faces*.

To create a databound diagram:

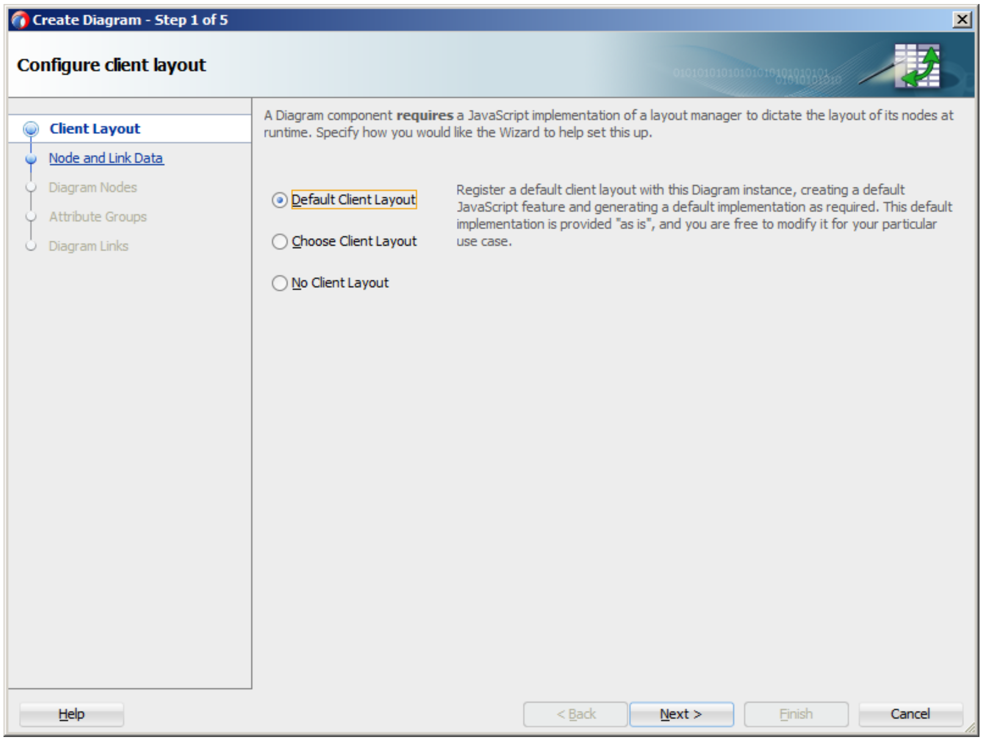
1. From the Data Controls panel, select the data collection that represents the nodes to use for your diagram.

   For example, to use the data collection shown in this section, select `NodesView1`.

2. Drag the collection onto a JSF page and, from the menu, choose **Diagram**.
3. In the Create Diagram - Configure Client Layout dialog, select your client layout.

   You can choose the default client layout, provide your own layout, or choose no layout. To choose the default layout, click **Default Client Layou**t as shown in Figure 44-5.

Figure 44-5 Configure Client Layout Dialog Showing Default Client Layout Selection



Configure client layout

A Diagram component **requires** a JavaScript implementation of a layout manager to dictate the layout of its nodes at runtime. Specify how you would like the Wizard to help set this up.

● Default Client Layout

○ Choose Client Layout

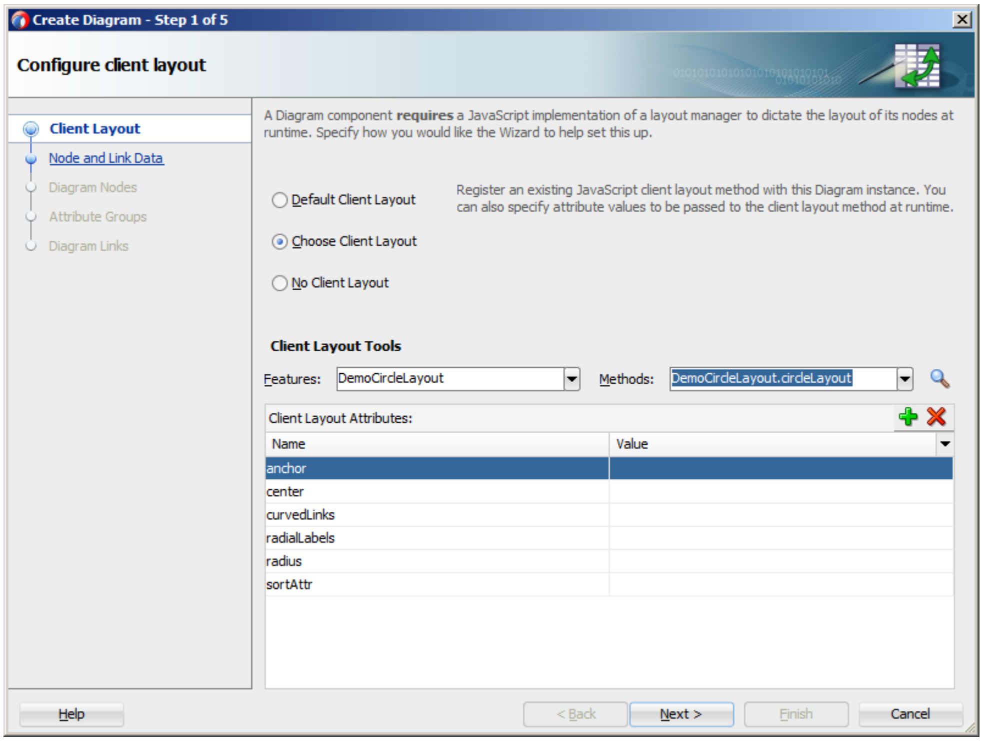
○ No Client Layout

Register a default client layout with this Diagram instance, creating a default JavaScript feature and generating a default implementation as required. This default implementation is provided "as is", and you are free to modify it for your particular use case.

To specify a different layout, click **Choose Client Layout** and select the feature and method that you want to use from the **Features** and **Methods** dropdown menus. After you choose the layout, the layout's optional attributes appear in the dialog, and you can enter values for the attributes as desired.

Figure 44-6 shows the completed Create Diagram - Configure client layout dialog for a custom circle layout. In this example, the selected feature name is `DemoCircleLayout`, and the selected method is `circleLayout`. Optional layout attributes include `anchor`, `center`, `curvedLinks`, `radialLabels`, `radius`, and `sortAttr`.

Figure 44-6 Configure Client Layout Dialog Showing Custom Layout Selection



Create Diagram - Step 1 of 5

Configure client layout

- Client Layout
- Node and Link Data
- Diagram Nodes
- Attribute Groups
- Diagram Links

A Diagram component **requires** a JavaScript implementation of a layout manager to dictate the layout of its nodes at runtime. Specify how you would like the Wizard to help set this up.

○ Default Client Layout

● Choose Client Layout

○ No Client Layout

Register an existing JavaScript client layout method with this Diagram instance. You can also specify attribute values to be passed to the client layout method at runtime.

Client Layout Tools

Features: DemoCircleLayout

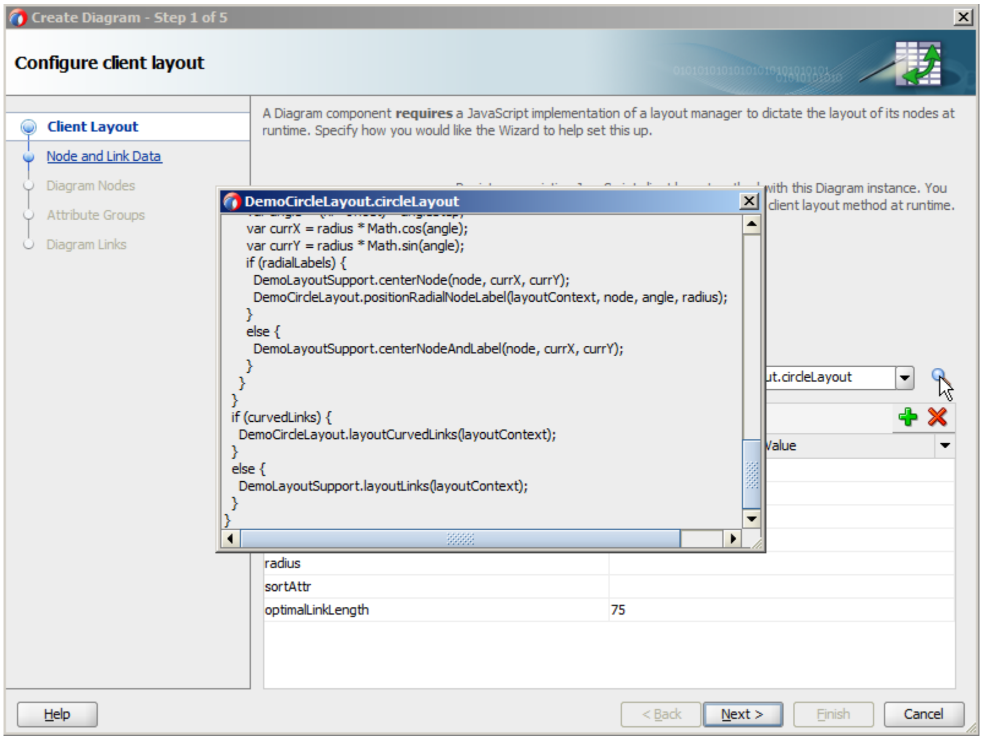
Methods: DemoCircleLayout.circleLayout

Client Layout Attributes:

Name	Value
anchor	
center	
curvedLinks	
radialLabels	
radius	
sortAttr	

If you want to look at the code source for the method you selected, click **Search** to display a popup containing the source code. Figure 44-7 shows the popup that displays the `DemoCircleLayout.circleLayout` method.

Figure 44-7 Configure Client Layout Dialog Showing Method Source Code Popup



A Diagram component **requires** a JavaScript implementation of a layout manager to dictate the layout of its nodes at runtime. Specify how you would like the Wizard to help set this up.

with this Diagram instance. You
client layout method at runtime.

ut.circleLayout

DemoCircleLayout.circleLayout

```
      var currX = radius * Math.cos(angle);
      var currY = radius * Math.sin(angle);
      if (radialLabels) {
        DemoLayoutSupport.centerNode(node, currX, currY);
        DemoCircleLayout.positionRadialNodeLabel(layoutContext, node, angle, radius);
      }
      else {
        DemoLayoutSupport.centerNodeAndLabel(node, currX, currY);
      }
    }
  }
  if (curvedLinks) {
    DemoCircleLayout.layoutCurvedLinks(layoutContext);
  }
  else {
    DemoLayoutSupport.layoutLinks(layoutContext);
  }
}
```

	Value
radius	
sortAttr	
optimalLinkLength	75

4. In the Create Diagram - Specify Node and Link Data dialog, enter values for the following to identify the data sources and link connections:

   - **Collection**: Verify that the correct data collection is displayed for the node collection.

     To change the data collection, click **Add Data Source** and choose another data collection from the list.

   - **Node Id**: Enter a value or use the dropdown menu to select the attribute in the node's data collection that represents the unique identifier of the node in the diagram.

     Alternatively, you can enter an EL expression that returns the value or select **Expression Builder** from the dropdown menu to launch the Expression Builder dialog. For help with the Expression Builder dialog, press **F1** or click **Help**.

   - **Container Id**: Enter or use the dropdown list to select the attribute in the node's data collection that represents the parent node for child nodes in the diagram.

     Alternatively, you can enter an EL expression that returns the value or select **Expression Builder** from the dropdown menu to launch the Expression Builder dialog.

   - **Collection** (Link): Enter the data collection that represents the links, or click **Add Data Source** to choose the data collection from the list.

   - **Start Node Id**: Enter or use the dropdown list to select the attribute in the link's data collection that represents the source node in diagram.

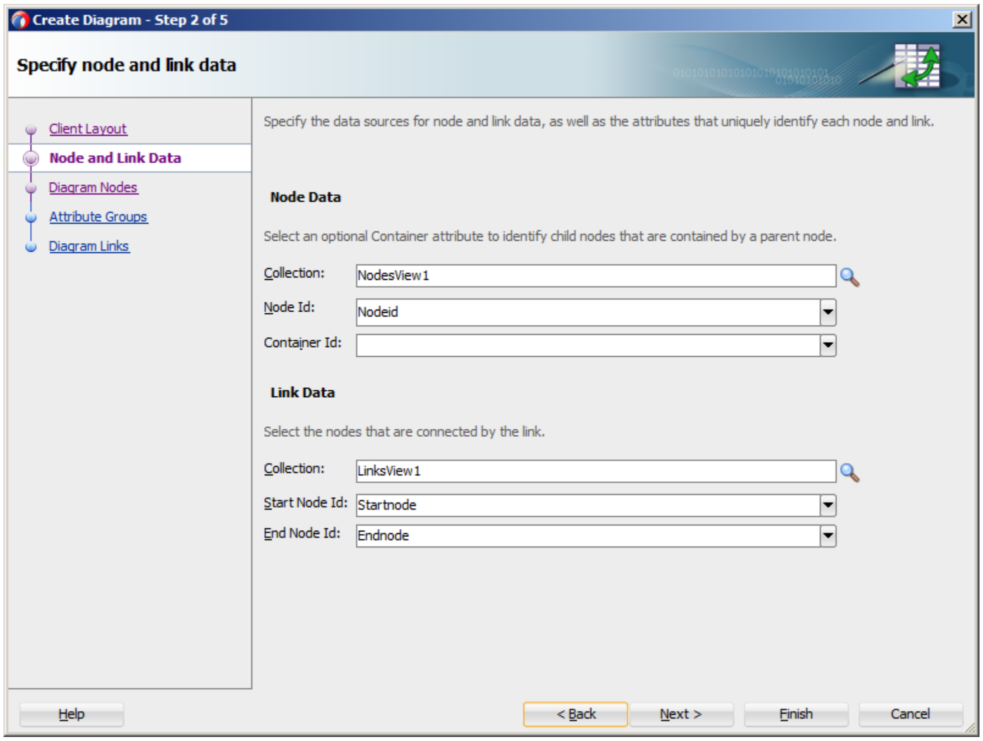
Alternatively, you can enter an EL expression that returns the value or select **Expression Builder** from the dropdown menu to launch the Expression Builder dialog.

- **End Node Id**: Enter or use the dropdown list to select the attribute in the link's data collection that represents the target node in diagram.

  Alternatively, you can enter an EL expression that returns the value or select **Expression Builder** from the dropdown menu to launch the Expression Builder dialog.

Figure 44-8 shows the completed dialog for a diagram using the `NodesView1` and `LinksView1` data collections. In this example, the **Node Id** value is `Nodeid`, and the link's **Start Node Id** and **End Node Id** values are set to the `Startnode` and `Endnode` attributes in the `LinksView1` data collection. In this example, the parent container has no children, and the **Container ID** field is not used.

Figure 44-8 Specify Node and Link Data Dialog Example



Create Diagram - Step 2 of 5

Specify node and link data

Specify the data sources for node and link data, as well as the attributes that uniquely identify each node and link.

Node Data

Select an optional Container attribute to identify child nodes that are contained by a parent node.

Collection: NodesView1
Node Id: Nodeid
Container Id:

Link Data

Select the nodes that are connected by the link.

Collection: LinksView1
Start Node Id: Startnode
End Node Id: Endnode

5. In the Create Diagram - Configure Diagram Nodes dialog, optionally enter values for the following to configure the display characteristics of the node's label and marker:

   - **Node Label**: From the dropdown list, select the attribute in the data collection to use for the node's label. You can select one of the attributes provided or select **Expression Builder** to enter a JSF EL expression.

     Optionally, you can enter values for the label's width, height, and opacity. For help with the dialog, press **F1** or click **Help**.

> **Tip:**
>
> If you're not sure yet which values make sense for your diagram, you can create the diagram and modify the values after creation.

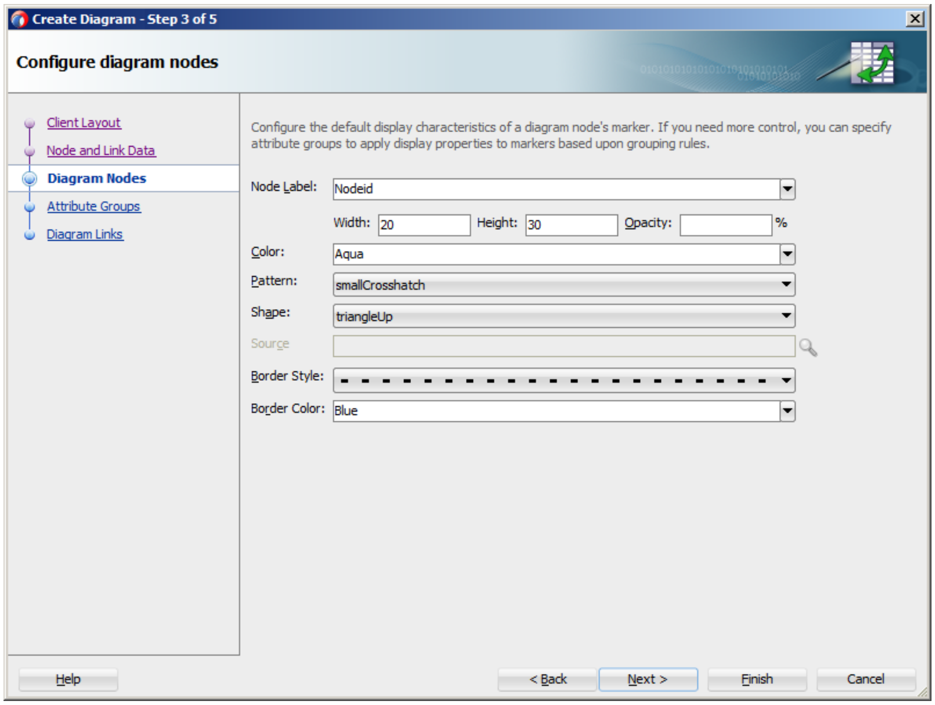
- **Color**: From the dropdown list, select a node color from the provided colors or select **Custom Colo**r to enter the Color Picker dialog. The default node color is `black`.
- **Pattern**: From the dropdown list, select a node pattern. Valid values for pattern include a choice of twelve prebuilt patterns, such as small and large checkers, diamonds, and triangles. If you specified a node color, the pattern will display in the color you chose. The default node pattern is `none`.
- **Shape**: From the dropdown list, select a shape to use for the node. Valid values include: `circle`, `human`, `diamond`, `square`, `triangleDown`, `triangleUp`, and `plus`. The default node shape is `circle`.

> **Note:**
>
> When you specify the node's shape, color, or pattern explicitly, the same value will be used for each node in the diagram. If you want to vary the color, pattern, or shape according to some measure, configure an attribute group in the next page of the Create Diagram wizard instead.

- **Source**: To use a custom node shape, enter the relative path to the SVG file to use for the node image, or use the Search icon to open the Select SVG File dialog to navigate to the image source.
- **Border Style**: From the dropdown list, select a border style for all nodes in the diagram. Valid values include `solid`, `dash`, and `dot`. The default style is a solid line.
- **Border Color**: From the dropdown list, select a border color for all nodes in the diagram or select **Custom Color** to enter the Color Picker dialog. The default border color is `black`.

  Figure 44-9 shows a completed Create Diagram - Configure Diagram Nodes dialog. In this example, the node label is configured with a relative width of 20 and relative height of 30. The node's shape is a triangle pointing up with an aqua-colored small crosshatch pattern.

Figure 44-9   Configure Diagram Nodes Dialog Example



Create Diagram - Step 3 of 5

Configure diagram nodes

Configure the default display characteristics of a diagram node's marker. If you need more control, you can specify attribute groups to apply display properties to markers based upon grouping rules.

Node Label: Nodeid

Width: 20 Height: 30 Opacity: %

Color: Aqua

Pattern: smallCrosshatch

Shape: triangleUp

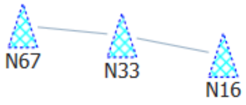
Source

Border Style: -

Border Color: Blue

[Figure 44-10](#) shows the runtime view of three diagram nodes after creation.

Figure 44-10   Diagram Nodes With Optional Display Attributes



6. (Optional) In the Create Diagram - Configure Attribute Groups, to configure one or more attribute groups for the diagram nodes, click **New** in the **Grouping Rules** section and enter values for the following:

   - **Group by value**: From the dropdown list, select the attribute in the data collection to group by in the attribute group. You can select one of the attributes provided or select **Expression Builder** to enter a JSF EL expression.
   - **Property**: From the dropdown list, select the property that you want to group by for display. Available options include `Color`, `Opacity`, `Pattern`, `Scale X`, `Scale Y`, and `Shape`.

     To vary the attribute group by more than one property, select **Select Multiple Attributes**. In the Select Multiple Attributes, select the desired properties, and click **OK**.
   - **Legend Label**: From the dropdown list, select the attribute in the data collection to display in the diagram legend. You can select one of the attributes provided or select **Expression Builder** to enter a JSF EL expression.

- **Section Label**: To use a text resource for the label, select **Select Text Resource** to associate a text resource for the label. If you wish to access data stored in objects, or reference and invoke methods using an EL Expression, select **Expression Builder** to launch the Expression Builder dialog.

  > **Note:**
  >
  > The section label describes the legend content of a sub-section of the legend and is rendered in the legend area. In Figure 44-2, for example, Group and Type are section labels.

- Optionally, click **Value-Specific Rules** to expand the attribute group dialog to specify a match or exception rule. Use match rules to specify colors or patterns for simple true or false conditions or when you want to match a specific value. Use exception rules when you want to specify a color or pattern when the grouped-by value meets a specific condition.

  To specify a match rule, in the **Match Rules** section, click **New** and enter values for the following:

  – **Group Value**: Enter the category value for the match. This can be a string that represents a category or you can set this to `true` or `false`. If you set this to true or false, the **Group by value** field must contain an EL expression that evaluates to true or false as in the following example:`#{row.AmountInStock gt row.ReorderPoint}`.

  – **Property**: From the dropdown list, select the property that you want to group by for display. Available options include `Color`, `Opacity`, `Pattern`, `Scale X`, `Scale Y`, and `Shape`.

  – **Property Value**: From the dropdown list, select the color or pattern to display when the node's value matches the **Group Value**. For color, you can select one of the provided values or select **Custom Color** to enter a custom color in the Select Custom Color dialog. For pattern, you must select one of the provided values.

  To specify an exception rule, in the **Exception Rules** section, click **New** and enter values for the following:

  – **Condition**: Enter a JSF EL expression that evaluates to true or false. You can enter the expression directly in the **Condition** field or select **Expression Builder** to enter the JSF EL expression.

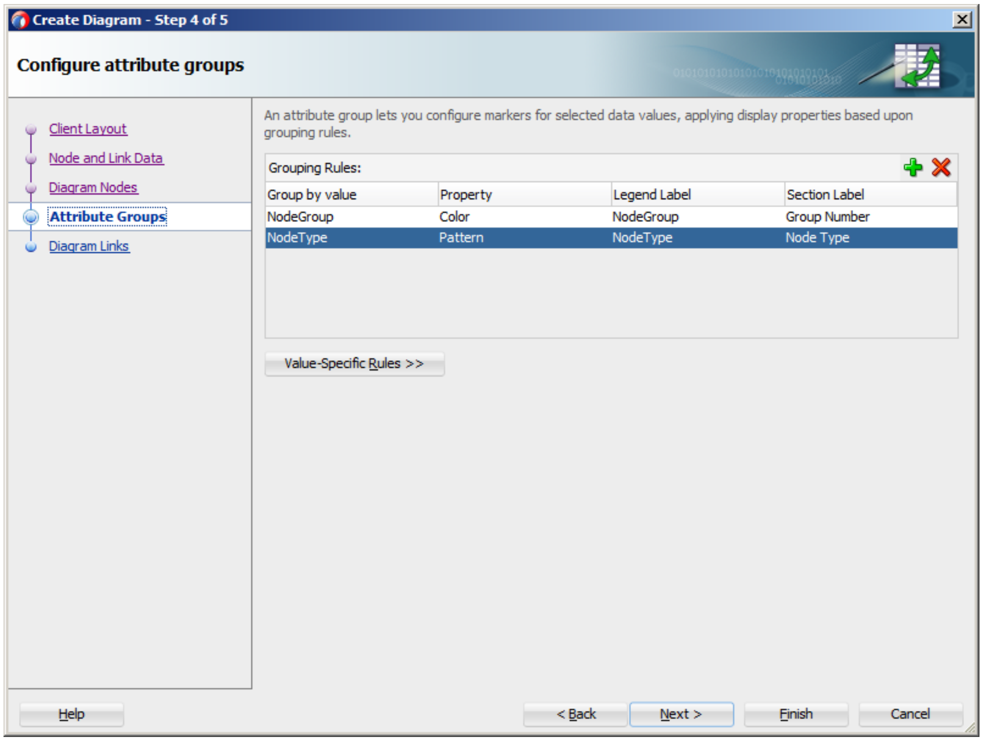
  – **Property**: From the dropdown list, select the property that you want to group by for display. Available options include `Color`, `Opacity`, `Pattern`, `Scale X`, `Scale Y`, and `Shape`.

  – **Property Value**: From the dropdown list, select the color or pattern to display when the node's value meets the condition you specified in the **Condition** field. For color, you can select one of the provided values or select **Custom Color** to enter a custom color in the Select Custom Color dialog. For pattern, you must select one of the provided values.

  – **Legend Label**: From the dropdown list, select **Select Text Resource** to select a text resource to be used for the legend label. You can also enter text in this field or select **Expression Builder** to enter a JSF EL expression.

Figure 44-11 shows the completed Create Diagram - Configure Attribute Groups dialog for the diagram in Figure 44-2. In this example, the diagram is configured for two attributes groups. The first attribute group varies the node's color depending upon the value in the `NodeGroup` attribute, and the second attribute group varies the node's shape depending upon the value in the `NodeType` attribute. The legend will display section labels for both groups.

Figure 44-11   Configure Attribute Groups Dialog Example



An attribute group lets you configure markers for selected data values, applying display properties based upon grouping rules.

Grouping Rules:

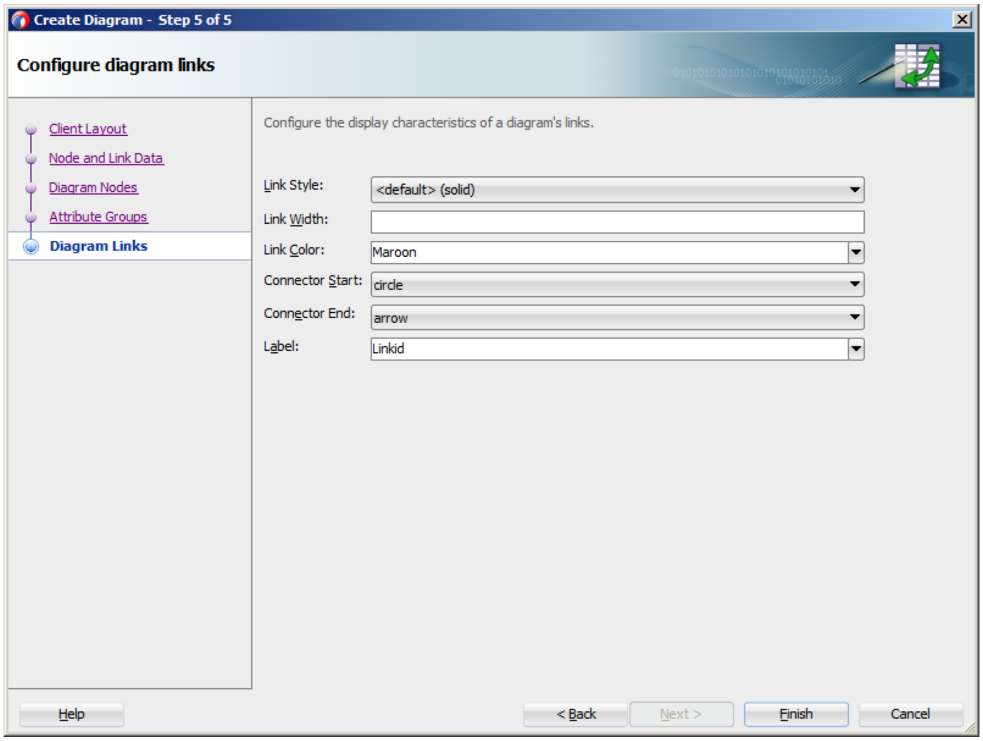
Group by value	Property	Legend Label	Section Label
NodeGroup	Color	NodeGroup	Group Number
NodeType	Pattern	NodeType	Node Type

7. In the Create Diagram - Configure Diagram Links dialog, enter values for the following to configure the display characteristics of the link label, style, and connector:

   - **Link Style**: From the dropdown list, select a border style for all nodes in the diagram. Valid values include `solid`, `dash`, `dot`, and `dashDot`. The default style is a solid line.
   - **Link Width**: Enter the width of the link in pixels. The default value is `1`.
   - **Link Color**: From the dropdown list, select a link color from the provided colors or select **Custom Colo**r to enter the Color Picker dialog. The default link color is `black`.
   - **Connector Start**: Enter a value for the link's start connector. Valid values include `none`, `arrowOpen`, `arrow`, `arrowConcave`, `circle`, `rectangle`, and `rectangleRounded`. The default value is `none`.
   - **Connector End**: Enter a value for the link's end connector. Valid values include `none`, `arrowOpen`, `arrow`, `arrowConcave`, `circle`, `rectangle`, and `rectangleRounded`. The default value is `none`.
   - **Label**: From the dropdown list, select the attribute in the data collection to use for the links's label. You can select one of the attributes provided or select **Expression Builder** to enter a JSF EL expression.

Figure 44-12 shows a completed Create Diagram - Configure Diagram Links dialog. In this example, the link is configured with a label and the default width and style. The link's color is maroon and will display a circle at the connector's start and an arrow at the connector's end.

Figure 44-12 Configure Dialog Links Dialog Example

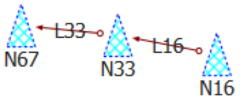


Configure the display characteristics of a diagram's links.

Link Style: <default> (solid)
Link Width:
Link Color: Maroon
Connector Start: circle
Connector End: arrow
Label: Linkid

Figure 44-13 shows the runtime view of three diagram nodes after creation.

Figure 44-13 Diagram Links With Optional Display Characteristics



8. When you have completed the Create Diagram wizard, click **Finish** to exit the wizard and add the diagram to the page.

What Happens When You Create a Databound Diagram

Creating a diagram from the Data Controls panel has the following effect:

- Creates the bindings for the diagram in the page definition file (*pageName*`PageDef.xml`) of the JSF page
- Adds the necessary tags to the JSF page for the `dvt:diagram` component
- If you chose the default client layout during creation, registers the layout with the ADF Faces runtime environment

Bindings for Diagram Components

The following code sample shows the bindings that JDeveloper generated for a `dvt:diagram` component using the data collection shown in Figure 44-3.

```
<executables>
  <variableIterator id="variables"/>
    <iterator Binds="NodesView1" RangeSize="-1"
DataControl="AppModuleDataControl" id="NodesView1Iterator"/>
    <iterator Binds="LinksView1" RangeSize="-1"
DataControl="AppModuleDataControl" id="LinksView1Iterator"/>
</executables>
<bindings>
  <tree IterBinding="NodesView1Iterator" id="NodesView1">
    <nodeDefinition DefName="model.NodesView" Name="NodesView10">
      <AttrNames>
        <Item Value="NodeGroup"/>
        <Item Value="NodeType"/>
        <Item Value="Nodeid"/>
      </AttrNames>
    </nodeDefinition>
  </tree>
  <tree IterBinding="LinksView1Iterator" id="LinksView1">
    <nodeDefinition DefName="model.LinksView" Name="LinksView10">
      <AttrNames>
        <Item Value="Endnode"/>
        <Item Value="Startnode"/>
      </AttrNames>
    </nodeDefinition>
  </tree>
</bindings>
```

The rules for populating the diagram are defined in a node definition. Each node definition references the node and links view objects and the attributes specified in the Create Diagram dialog. For additional information about the *pageName*`PageDef.xml` file, see pageNamePageDef.xml.

Code on the JSF Page for a Diagram Component

The following example shows the code that is generated on the JSF page for the diagram shown in Figure 44-2.

```
<dvt:diagram id="dg1" layout="DiagramSampleDiagramLayout">
  <dvt:diagramNodes var="node" id="dnodes1"
                    value="#{bindings.NodesView1.collectionModel}">
    <dvt:diagramNode label="#{node.Nodeid}" id="dn1" nodeId="#{node.Nodeid}">
      <f:facet name="zoom100">
        <dvt:marker width="20" id="m1" height="30">
          <dvt:attributeGroups id="ag1" label="#{node.NodeGroup}" type="color"
                               value="#{node.NodeGroup}"/>
          <dvt:attributeGroups id="ag2" label="#{node.NodeType}" type="shape"
                               value="#{node.NodeType}"/>
        </dvt:marker>
      </f:facet>
    </dvt:diagramNode>
  </dvt:diagramNodes>
  <dvt:diagramLinks var="link" value="#{bindings.LinksView1.collectionModel}"
                    id="dl1">
    <dvt:diagramLink startNode="#{link.Startnode}" id="dl2"
```

```
                     endNode="#{link.Endnode}"/>
  </dvt:diagramLinks>
  <dvt:clientLayout method="DiagramSampleDiagramLayout.forceDirectedLayout"
                    featureName="DiagramSampleDiagramLayout"
                    name="DiagramSampleDiagramLayout">
    <f:attribute name="optimalLinkLength" value="75"/>
  </dvt:clientLayout>
  <dvt:legend id="l1">
    <dvt:legendSection label="#{viewcontrollerBundle.GROUP}" source="dnodes1:ag1"
                       id="ls1"/>
    <dvt:legendSection label="#{viewcontrollerBundle.TYPE}" source="dnodes1:ag2"
                       id="ls2"/>
  </dvt:legend>
</dvt:diagram>
```

Default Client Layout Files and Location

When you create a diagram using the default client layout, JDeveloper registers the force directed layout with the ADF runtime environment as a JavaScript (JS) feature. The creation process automatically adds a JS file in the ViewController/src/js/layout directory with the following naming convention:

*ApplicationName*DiagramLayout.js

For example, if your application is named DiagramSample, the JS file is named DiagramSampleDiagramLayout.js. The content of the file contains the forceDirectedLayout method, and you can open the file in JDeveloper. You can also click **Edit Component Definition** in the Properties window to display the Edit Diagram - Configure Client Layout dialog and then click **Search** to display the method popup.

JDeveloper also creates the adf-js-features.xml file if it doesn't already exist and adds the *ApplicationName*DiagramLayout as the feature-name and the path to the JS file as the feature-class. The following example shows the adf-js-features.xml file for an application named DiagramSample.

```
<?xml version="1.0" encoding="UTF-8" ?>
<features xmlns="http://xmlns.oracle.com/adf/faces/feature">
  <feature>
    <feature-name>DiagramSampleDiagramLayout</feature-name>
    <feature-class>js/layout/DiagramSampleDiagramLayout.js</feature-class>
  </feature>
</features>
```

Modifying Diagram Properties and Layout

After you create your diagram, you can specify a different client layout or add additional elements, such as a label, using the Edit Diagram dialog. To open the dialog, use the **Edit** icon in the Properties window for the dvt:diagram component. You can also customize the diagram properties directly in the visual editor or by setting values in the Properties window.

For example, to add the tooltip shown in Figure 44-2 to your diagram, you can enter a value for the diagram node's shortDesc attribute in the Properties window or in the code editor.

```
<dvt:diagramNode label="#{node.Nodeid}" id="dn1" nodeId="#{node.Nodeid}"
                 shortDesc="Group: #{node.NodeGroup} Type:
#{node.NodeType}">
```

```
  <f:facet name="zoom100">
    ... contents omitted
  </f:facet>
</dvt:diagramNode>
```

For additional information and examples for customizing your diagram, see Using Diagram Components in *Developing Web User Interfaces with Oracle ADF Faces*.

45
Creating Databound Tag Cloud Components

This chapter describes how to how to create a tag cloud from data modeled with ADF Business Components, using ADF data controls and ADF Faces components in a Fusion web application. Specifically, it describes how you can use ADF Data Visualization `tagCloud` components to create tag clouds that visually represent business data. It describes how to use ADF data controls to create these components with data-first development.

If you are designing your page using simple UI-first development, then you can add the tag cloud to your page and configure the data bindings later. For information about the data requirements, tag structure, and options for customizing the look and behavior of the tag cloud component, see Using Tag Cloud Components in *Developing Web User Interfaces with Oracle ADF Faces*.

This chapter includes the following sections:

- About ADF Data Visualization Tag Cloud Components
- Creating Databound Tag Cloud Components

About ADF Data Visualization Tag Cloud Components

A tag cloud is a visual representation for text data, typically used to visualize free form text or tag metadata for a website. The importance or frequency of each tag is shown with font size or color. This visualization is useful for quickly identifying the most prominent terms in a set.

ADF Data Visualization components provide extensive graphical and tabular capabilities for visually displaying and analyzing business data. Each component must be bound to data before it can be rendered since the appearance of the components is dictated by the data that is displayed.

The prefix `dvt:` occurs at the beginning of each data visualization component name indicating that the component belongs to the ADF Data Visualization Tools (DVT) tag library.

Data Visualization Components Use Cases and Examples

For detailed descriptions of Tag Cloud component use cases and examples, see Tag Cloud Use Cases and Examples in *Developing Web User Interfaces with Oracle ADF Faces*.

End User and Presentation Features

Visually compelling data visualization components enable end users to understand and analyze complex business data. The components are rich in features that provide out-of-the-box interactivity support.

For detailed descriptions of the end user and presentation features for the DVT tag cloud component, see End User and Presentation Features of Tag Clouds in *Developing Web User Interfaces with Oracle ADF Faces*.

Additional Functionality for Data Visualization Components

You may find it helpful to understand other **Oracle ADF** features before you data bind your data visualization components. Additionally, once you have added a data visualization component to your page, you may find that you need to add functionality such as validation and accessibility. Following are links to other functionality that data visualization components use:

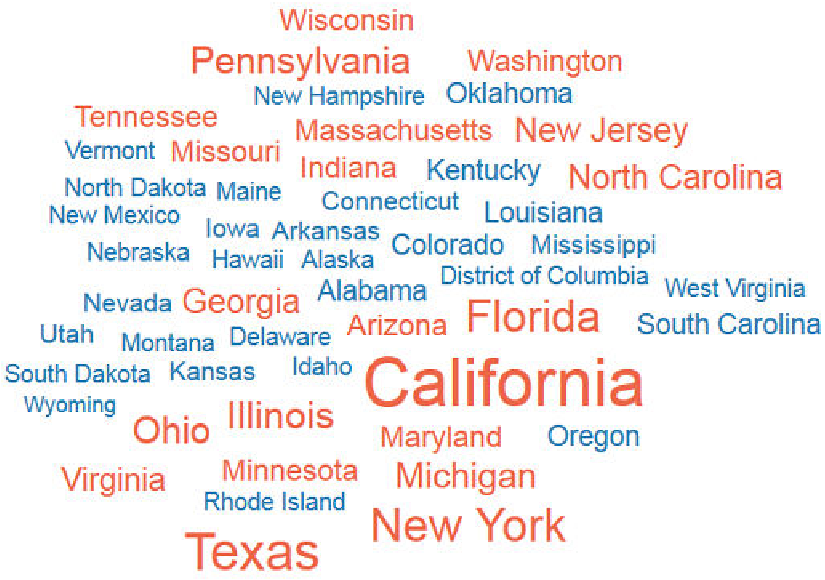
- Partial page rendering: You may want a data visualization component to refresh to show new data based on an action taken on another component on the page. For more information, see the Rerendering Partial Page Content chapter in *Developing Web User Interfaces with Oracle ADF Faces*.
- Personalization: Users can change the way the data visualization components display at runtime, those values will not be retained once the user leaves the page unless you configure your application to allow user customization. For more information, see the Allowing User Customization on JSF Pages chapter in *Developing Web User Interfaces with Oracle ADF Faces*.
- Accessibility: By default, data visualization components are accessible. You can make your application pages available to screen readers. For more information, see the Developing Accessible ADF Faces Pages chapter in *Developing Web User Interfaces with Oracle ADF Faces*.
- Skins and styles: You can customize the appearance of data visualization components using an ADF skin that you apply to the application or by applying CSS style properties directly using a style-related property (`styleClass` or `inlineStyle`). For more information, see the Customizing the Appearance Using Styles and Skins chapter in *Developing Web User Interfaces with Oracle ADF Faces*.
- Placeholder data controls: If you know the data visualization components on your page will eventually use ADF data binding, but you need to develop the pages before the data controls are ready, then you should consider using placeholder data controls, rather than manually binding the components. Using placeholder data controls will provide the same declarative development experience as using developed data controls. For more information, see Designing a Page Using Placeholder Data Controls.

Creating Databound Tag Cloud Components

The ADF Tag Cloud component displays tag data in either a rectangular container or in a free-form cloud. Tags represent the actual data and are stamped inside the cloud layout according to their weight, with heavier or more important tags in the middle.

Figure 45-1 shows a tag cloud configured to show all the states in the USA. States with a population of less than 5 million have blue tags and states with a population of 5 million or more have red tags. This tag cloud renders high density tags nearer to the center.

Figure 45-1   Tag Cloud Component showing High and Low Population States in USA



For detailed information about tag cloud end user and presentation features, use cases, tag structure, and adding special features to tag clouds, see the Using Tag Cloud Components chapter in *Developing Web User Interfaces with Oracle ADF Faces*.

How to Create a Databound Tag Cloud

A tag cloud is used to display tag information or website metadata in a free flow format. You can create a tag cloud by using the Data Controls panel.

Before you begin:

It may be helpful to have an understanding of databound tag clouds. For more information, see Creating Databound Tag Cloud Components.

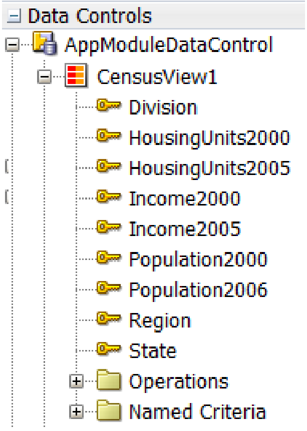
You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Additional Functionality for Data Visualization Components.

You will need to complete these tasks:

- Create an application module that contains instances of the view objects that you want in your data model for the tag cloud, as described in Creating and Modifying an Application Module.

  Figure 45-2 shows the data control for the tag cloud displayed in Creating Databound Tag Cloud Components. In this example, `CensusView1` contains the data collection for the tag cloud. Tags are represented visually as free-form text.. Each tag is sized and stamped according to the values in the tag's `Population2006` attribute.

Figure 45-2 Data Control for Tag Cloud Showing High and Low Population States in the USA



The values of the `Population2006` attribute are checked against Match Rules in the tag cloud. Tags with a value of more than 5 million are rendered in red and the other tags are rendered in blue.

- Create a JSF page as described in the How to Create JSF Pages section of *Developing Web User Interfaces with Oracle ADF Faces*.

To create a databound tag cloud:

1. From the Data Controls panel, select a collection.

   For example, to use the data control shown in this section, select `CensusView1`.

2. Drag the collection onto a JSF page and, from the menu, choose **Tag Cloud**.

3. In the Create Tag Cloud dialog, do the following to configure the tag cloud:

   - From the dropdown menu of the **Text Value** field, select an attribute to represent the text that will be shown in the rendered tag cloud.

     For example, to configure the tag cloud in the sample, select the `State` attribute.

   - To configure the tag sizes, select an attribute from the dropdown menu of the **Relative Size** field.

     For example, to configure the tag cloud in the sample, select the `Population2006` attribute

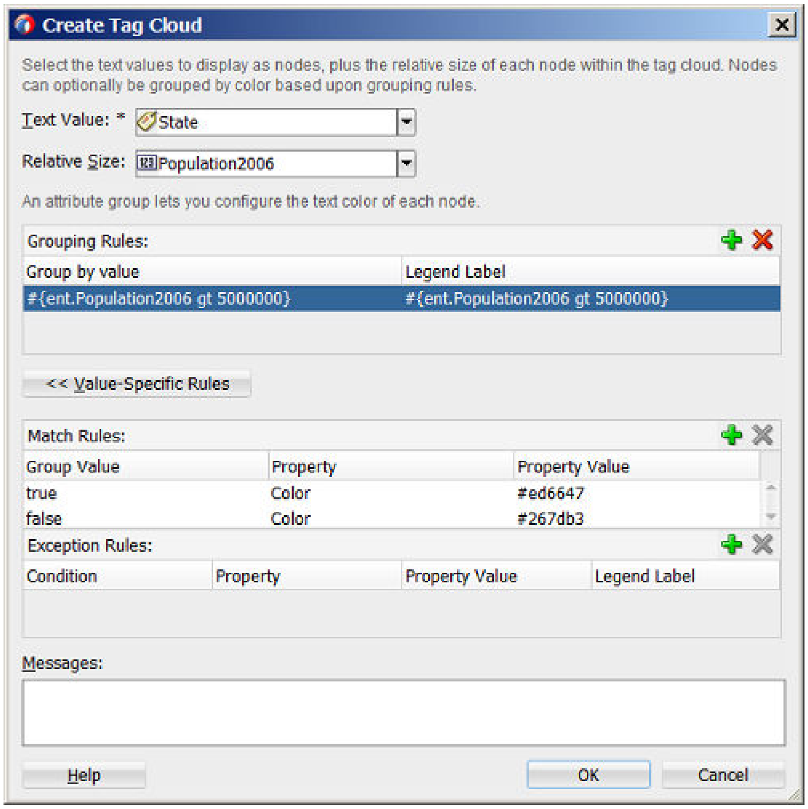
   - To configure tag colors, create a rule under **Grouping Rules** and then create appropriate **Value-Specific Rules**. Use **Match Rules** to specify colors for simple true or false conditions or when you want to match a specific value. Use **Exception Rules** when you want to specify a color when the tag's grouped-by value meets a specific condition.

     For example, to configure the tag cloud in the sample, create a new Grouping rule. Under both **Group by value** and **Legend Label**, provide `#{ent.Population2006 gt 5000000}`.

Then create two new Match Rules. For the first, provide a **Group Value** of `true` and a **Property Value** of `#ed6647`. For the second, provide a **Group Value** of `false` and a **Property Value** of `#267db3`.

Figure 45-3 shows the Create Tag Cloud dialog that generates the sample tag cloud using the `CensusView1` data control.

Figure 45-3 Create Tag Cloud dialog for CensusView1



Create Tag Cloud

Select the text values to display as nodes, plus the relative size of each node within the tag cloud. Nodes can optionally be grouped by color based upon grouping rules.

Text Value: * State

Relative Size: Population2006

An attribute group lets you configure the text color of each node.

Grouping Rules:

Group by value	Legend Label
#{ent.Population2006 gt 5000000}	#{ent.Population2006 gt 5000000}

<< Value-Specific Rules

Match Rules:

Group Value	Property	Property Value
true	Color	#ed6647
false	Color	#267db3

Exception Rules:

Condition	Property	Property Value	Legend Label

Messages:

4. Click **OK**.

After completing the data binding dialog, you can use the Property Inspector to specify settings for the tag cloud attributes and you can also use the child tags associated with the `dvt:tagCloud` tag to customize the tag cloud further. For more information about configuring tag cloud components, see the Using Tag Cloud Components section in the *Developing Web User Interfaces with Oracle ADF Faces*.

What Happens When You Create a Databound Tag Cloud

Dropping a tag cloud from the Data Controls panel has the following effect:

- Creates the bindings for the tag cloud and adds the bindings to the page definition file
- Adds the necessary code for the UI components to the JSF page

The data binding XML that JDeveloper generates represents the physical model of the tag cloud you create. The example below shows the bindings that JDeveloper generated in the page definition file where a tag cloud was created using data from the `State` and `Population2006` attributes in the `CensusView1` data collection.

```
<tree IterBinding="CensusView1Iterator" id="CensusView1">
    <nodeDefinition DefName="view.CensusView" Name="CensusView10">
        <AttrNames>
            <Item Value="State"/>
            <Item Value="Population2006"/>
        </AttrNames>
    </nodeDefinition>
</tree>
```

The following example shows the code that is generated on the JSF page for the tag cloud:

```
<dvt:tagCloud id="tc1" var="ent"
value="#{bindings.CensusView1.collectionModel}" layout="cloud">
    <dvt:tagCloudItem label="#{ent.State}" id="tci1"
value="#{ent.Population2006}">
        <dvt:attributeGroups value="#{ent.Population2006 gt 5000000}"
type="color"

label="#{ent.Population2006}"    id="ag1">
            <dvt:attributeMatchRule group="true" id="amr1">
                <f:attribute name="color" value="#ed6647"/>
            </dvt:attributeMatchRule>
            <dvt:attributeMatchRule group="false" id="amr2">
                <f:attribute name="color" value="#267db3"/>
            </dvt:attributeMatchRule>
        </dvt:attributeGroups>
    </dvt:tagCloudItem>
</dvt:tagCloud>
```

After you create your tag cloud, you can modify the binding or add additional rules using the Edit Tag Cloud dialog. To open the dialog, use the Edit icon in the Properties window for the tag cloud component. You can also customize the attributes of the tag cloud directly in the code, in the visual editor, or by setting values in the Properties window.

46
Using Contextual Events

This chapter describes how to create, publish, and subscribe to contextual events to facilitate communications between ADF Faces regions in the Fusion web application. It describes how to declaratively create the events, as well as programmatically use managed beans and JavaScript.
This chapter includes the following sections:

About Creating Contextual Events

The Contextual Events feature in Oracle ADF provides a powerful mechanism for passing parameters to share information among pages and regions. You can create contextual events by using managed beans, plain Java objects, and JavaScripts.

The contextual events framework provides the communications between regions within a page. The framework provides the page with the ability to map events that are produced and consumed by the various regions on the page. You can use JDeveloper to declaratively publish events using the page definition file. Similarly, you can declaratively subscribe to those events from the page definition file. You can pass parameters with the event and implement handlers to respond to an event. Other ways to create contextual events include using managed beans and using JavaScript. Note that contextual events do not require regions to be refreshed for the region to uptake parameters.

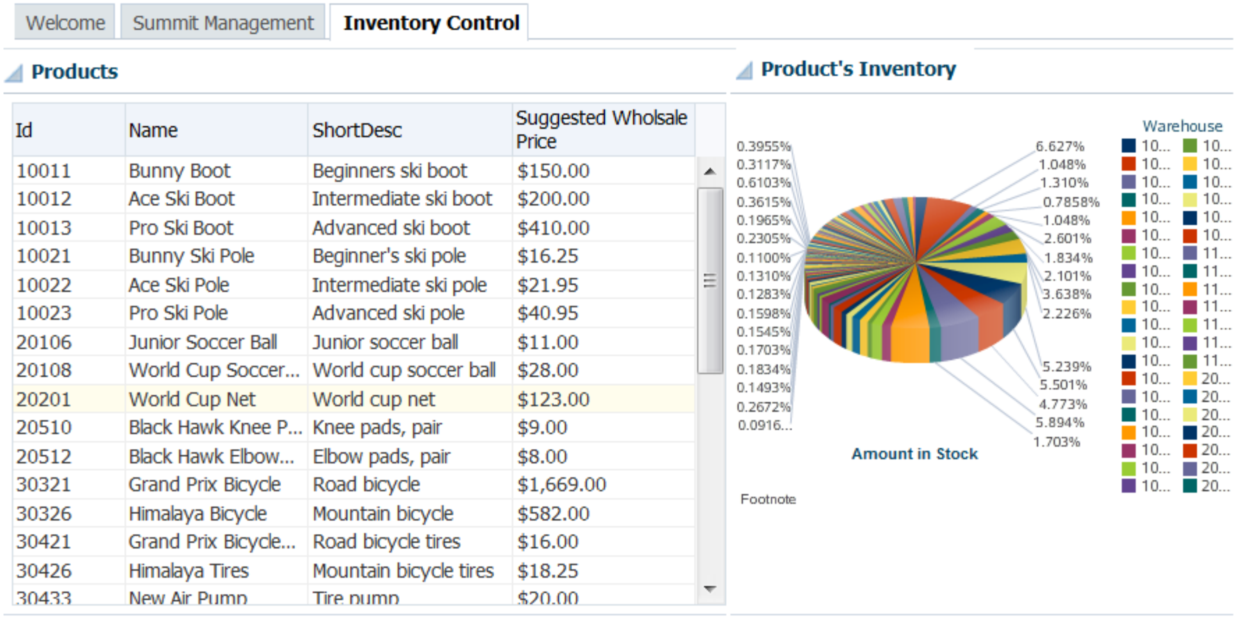
Contextual events are not the same as the business events that can be raised by **ADF Business Components** or the events raised by UI components. For a description of these types of events, see *Developing Web User Interfaces with Oracle ADF Faces*. Contextual events can be used, however, in association with UI events. In this case, an action listener that is invoked due to a UI event can, in turn, invoke a method action binding that then raises the event.

Often a page or a region within a page needs information from somewhere else on the page or from a different region. While you can pass parameters to obtain that information, doing so makes sense only when the parameters are well known and the inputs are EL-accessible to the page. Parameters are also useful when a task flow may need to be restarted if the parameter value changes.

However, suppose you have a task flow with multiple page fragments that contain various interesting values that could be used as input on one of the pages in the flow. If you were to use parameters to pass the value, the task flow would need to surface output parameters for the union of each of the interesting values on each and every fragment. Instead, for each fragment that contains the needed information, you can define a contextual event that will be raised when the page is submitted. The page or fragment that requires the information can then subscribe to the various events and receive the information through the event.

In the Summit sample application for ADF task flows, contextual events are used in the Inventory Control tab of the main page to display the inventory for a selected product. When the Inventory Control tab is selected by the user, it displays two regions side by side. One region contains the `showProducts.jsff` page fragment which contains a table of the products available. The other region contains the `showInventory.jsff` page fragment which contains a DVT chart showing the product inventory percentages for different warehouses. Figure 46-1 shows the Inventory Control tab displaying the products and inventory regions.

Figure 46-1 Inventory Control Tab



Products

Id	Name	ShortDesc	Suggested Wholsale Price
10011	Bunny Boot	Beginners ski boot	$150.00
10012	Ace Ski Boot	Intermediate ski boot	$200.00
10013	Pro Ski Boot	Advanced ski boot	$410.00
10021	Bunny Ski Pole	Beginner's ski pole	$16.25
10022	Ace Ski Pole	Intermediate ski pole	$21.95
10023	Pro Ski Pole	Advanced ski pole	$40.95
20106	Junior Soccer Ball	Junior soccer ball	$11.00
20108	World Cup Soccer...	World cup soccer ball	$28.00
20201	World Cup Net	World cup net	$123.00
20510	Black Hawk Knee P...	Knee pads, pair	$9.00
20512	Black Hawk Elbow...	Elbow pads, pair	$8.00
30321	Grand Prix Bicycle	Road bicycle	$1,669.00
30326	Himalaya Bicycle	Mountain bicycle	$582.00
30421	Grand Prix Bicycle...	Road bicycle tires	$16.00
30426	Himalaya Tires	Mountain bicycle tires	$18.25
30433	New Air Pump	Tire pump	$20.00

Product's Inventory

The products region contains the `show-products-task-flow` task flow which utilizes the `showProducts.jsff` page fragment. The inventory region contains the `product-inventory-task-flow` task flow which utilizes the `showInventory.jsff` page fragment.

The main page `index.jsff` is the parent page for both regions, which are located in a child of the `af:panelTabbed` layout component.

A contextual event is passed from the products region to the inventory region so that the `showInventory-task-flow` can display the DVT chart for the selected product. When you create a event for the producer component using the Properties window, a event is registered in the producer's page definition file. In the Summit ADF sample application, the `productSelected` event is registered in the `showProductsPageDef.xml` file, like this:

```
<tree IterBinding="ProductVO1Iterator" id="ProductVO1">
      <nodeDefinition DefName="oracle.summit.model.views.ProductVO"
         Name="ProductVO10">
        <AttrNames>
          <Item Value="Id"/>
          <Item Value="Name"/>
          <Item Value="ShortDesc"/>
          <Item Value="LongtextId"/>
          <Item Value="ImageId"/>
          <Item Value="SuggestedWhlslPrice"/>
          <Item Value="WhlslUnits"/>
        </AttrNames>
        <events xmlns="http://xmlns.oracle.com/adfm/contextualEvent">
          <event name="productSelected" eventType="Currency Change Event"
             customPayLoad="#{bindings.Id.inputValue}"/>
        </events>
      </nodeDefinition>
</tree>
```

When you use the overview editor for the page definition file to subscribe to events, an eventMap will be created to map events between producers and consumers.

In the Summit ADF sample application, the eventMap is created in the consumer's page definition file, like this:

```
<eventMap xmlns="http://xmlns.oracle.com/adfm/contextualEvent">
    <event name="productSelected">
      <producer region="*">
        <consumer handler="populateInventoryForProduct" region="">
        <parameters>
        <parameter name="productId" value="${payLoad}"/>
        </parameters>
        </consumer>
      </producer>
    </event>
</eventMap>
```

At runtime, when users enter the `show-products-task-flow-definition` and the `showProducts.jsff` page fragment, they are presented with a table of the products that are available. When the user selects a product from the list, a currency change event is invoked, resulting in the broadcast of the `productSelected` contextual event. In this example, a `payLoad` parameter containing the `productId` of the selected product is passed with the event.

This event is then consumed by the `product-inventory` task flow and its handler, the `populateInventoryForProduct()` method. This method uses the `payLoad` parameter to determine which product's inventory information to be displayed in the DVT component.

You will need to create the code for the event handler. For the Summit ADF sample application, the `populateInventoryForProduct()` handler was added to the `InventoryVOImpl.java` file because the InventoryVO **view object** contains the inventory data, as shown below:

```
public void populateInventoryForProduct( String productId ) {
          applyViewCriteria(getViewCriteria("InventoryVOCriteria"));
          setinputProductId( productId );
          executeQuery();
}
```

```
/**
 * Returns the variable value for Variable.
 * @return variable value for Variable
 */
public String getinputProductId() {
        return
(String)ensureVariableManager().getVariableValue("inputProductId");
}

/**
 * Sets <code>value</code> for variable Variable.
 * @param value value to bind as Variable
 */
public void setinputProductId(String value) {
        ensureVariableManager().setVariableValue("inputProductId", value);
}
```

Events are configured in the page definition file for the page or region that will raise the event (the producer). In order to associate the producer with the consumer that will do something based on the event, you create an event map in the page definition file of the consuming page or the parent page. (The parent page is the page that contains both regions). If the consuming page is in a dynamic region, the event map should be in the page definition file of the consuming page and the producer's attribute region set to "*". The attribute region is set to "*" because at design time, the framework cannot determine the relative path to the producer.

You can raise a contextual event for an action binding, a method action binding, a value attribute binding, or a range binding (table, tree, or list binding). You also can conditionally fire an event and conditionally handle an event using EL expressions.

For action and method action bindings, the event is raised when the action or method is executed. The `payLoad` contains the method return value. You can also raise a contextual event from an ADF Faces event such as clicking a button or selecting from a menu. An `eventBinding` is created in the page definition to define the event.

For a value attribute binding, the event is triggered by the binding container and raised after the attribute is set successfully. The `payLoad` is an instance of `DCBindingContainerValueChangeEvent`, which provides access to the new and old value, the producer iterator, the binding container, and the source. If the `payLoad` property is changed to point to a custom data object, then the `payLoad` reference will return the object instead. An example below shows a value change event inside an attribute value binding associated with an input component. The event, `valueChangeEvent`, will be dispatched when the user changes the value of `LAST_NAME` in the page.

```
<attributeValues IterBinding="DeptView1Iterator" id="Dname"
            xmlns="http://xmlns.oracle.com/adfm/jcuimodel">
    <events xmlns="http://xmlns.oracle.com/adfm/contextualEvent">
            <event name="valueChangeEvent"/>
    </events>
    <AttrNames xmlns="http://xmlns.oracle.com/adfm/uimodel">
            <Item Value="LAST_NAME"/>
    </AttrNames>
</attributeValues>
</bindings>
<eventMap xmlns="http://xmlns.oracle.com/adfm/contextualEvent">
    <event name="valueChangeEvent">
        <producer region="LAST_NAME">
            <consumer region="" handler="consumeEvent"/>
        </producer>
    </event>
</eventMap>
```

For a range binding (tree, table, list), the event is raised after the currency change has succeeded. The `payLoad` is an instance of `DCBindingContainerValueChangeEvent`, which provides access to the new and old value, the producer iterator, the binding container, and the source. The `eventMap` example above shows the event map for a currency change event associated with a range binding.

Value attribute binding and range binding contextual events may also be triggered by navigational changes. For example, if you create an event inside a tree table binding, the event will be dispatched when the user selects a different node of the tree in the page.

You use the **Contextual Events** section under the **Behavior** section in the Properties window to create and publish contextual events. The Contextual Events panel will only appear when you select eligible components or regions in the page, as shown in Figure 46-2.

Figure 46-2   Contextual Events Panel in the Properties Window

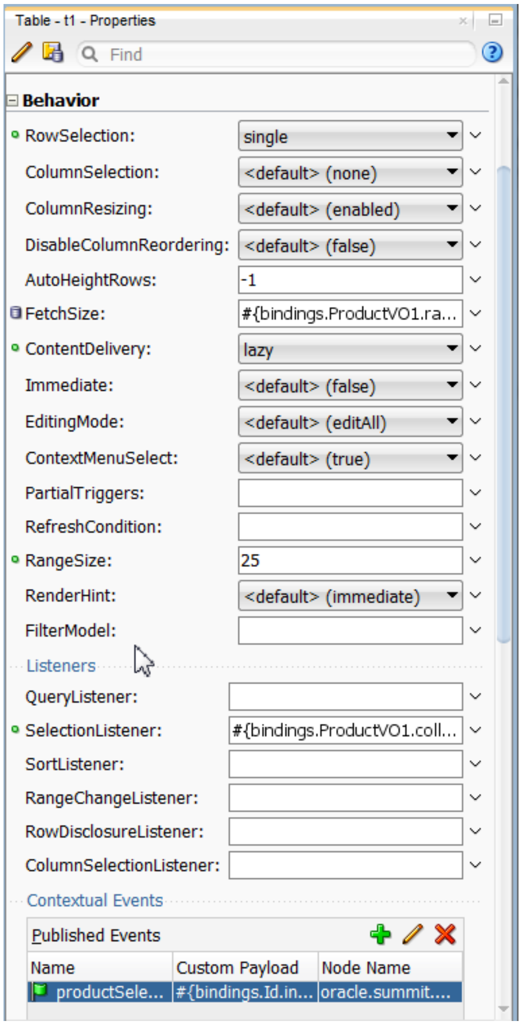


Table - t1 - Properties

Find

Behavior

RowSelection: single
ColumnSelection: <default> (none)
ColumnResizing: <default> (enabled)
DisableColumnReordering: <default> (false)
AutoHeightRows: -1
FetchSize: #{bindings.ProductVO1.ra...
ContentDelivery: lazy
Immediate: <default> (false)
EditingMode: <default> (editAll)
ContextMenuSelect: <default> (true)
PartialTriggers:
RefreshCondition:
RangeSize: 25
RenderHint: <default> (immediate)
FilterModel:

Listeners

QueryListener:
SelectionListener: #{bindings.ProductVO1.coll...
SortListener:
RangeChangeListener:
RowDisclosureListener:
ColumnSelectionListener:

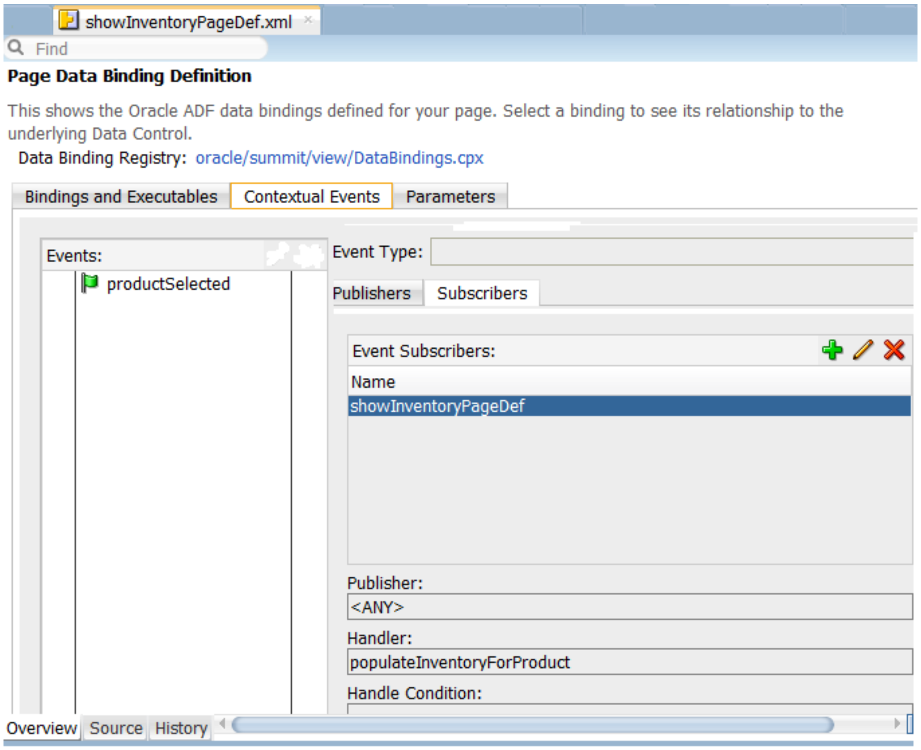
Contextual Events

Published Events

Name	Custom Payload	Node Name
productSele...	#{bindings.Id.in...	oracle.summit....

You subscribe to contextual events using the overview editor for page definition file's **Contextual Events** tab **Subscriber** section, as shown in Figure 46-3

Figure 46-3 Page Definition Contextual Events Tab



Page Data Binding Definition

This shows the Oracle ADF data bindings defined for your page. Select a binding to see its relationship to the underlying Data Control.

Data Binding Registry: oracle/summit/view/DataBindings.cpx

Events:
productSelected

Event Type:

Event Subscribers:

Name
showInventoryPageDef

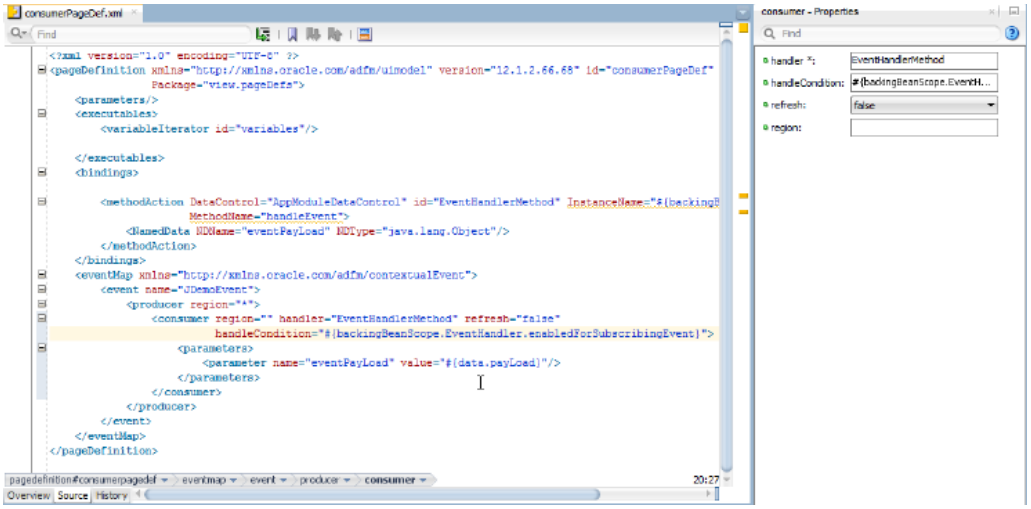
Publisher: <ANY>

Handler: populateInventoryForProduct

Handle Condition:

You can turn off automatic region refresh while consuming a contextual event by adding `refresh="false"` on the consumer element in the `eventMap` of the consuming region's page definition. Setting `refresh` to `false` is useful if you want to take charge of refresh for the event consuming region. The default for the refresh attribute is `true`. Additionally, you can turn off a subscriber region from consuming specific instances of contextual events using an EL binding on the `handleCondition` attribute also present in the consumer element inside the `eventMap` of the consuming region's page definition. The following page definition file source and Property window show a `methodAction` binding and an `eventMap` in a consuming region's page definition, where the addition of the `region` and `handleCondition` attributes control the consuming region's contextual event response.

Figure 46-4   Attributes of Consumer Element in a Consumer EventMap



```
<?xml version="1.0" encoding="UTF-8" ?>
<pageDefinition xmlns="http://xmlns.oracle.com/adfm/uimodel" version="12.1.2.66.68" id="consumerPageDef"
                Package="view.pageDefs">
    <parameters/>
    <executables>
        <variableIterator id="variables"/>

    </executables>
    <bindings>

        <methodAction DataControl="AppModuleDataControl" id="EventHandlerMethod" InstanceName="#{backingB
                      MethodName="handleEvent">
            <NamedData NDName="eventPayLoad" NDType="java.lang.Object"/>
        </methodAction>
    </bindings>
    <eventMap xmlns="http://xmlns.oracle.com/adfm/contextualEvent">
        <event name="JDemoEvent">
            <producer region="*">
                <consumer region="" handler="EventHandlerMethod" refresh="false"
                          handleCondition="#{backingBeanScope.EventHandler.enabledForSubscribingEvent}">
                    <parameters>
                        <parameter name="eventPayLoad" value="#{data.payLoad}"/>
                    </parameters>
                </consumer>
            </producer>
        </event>
    </eventMap>
</pageDefinition>
```

consumer - Properties

handler *: EventHandlerMethod

handleCondition: #{backingBeanScope.EventH...

refresh: false

region:

Contextual Events Use Cases and Examples

There are three kinds of communication patterns between a region and the parent page: parent to region, region to parent, and region to region. The contextual events framework is the most powerful communication implementation that works for all three of these scenarios. Also note that contextual events do not require a region to be refreshed in order to consume input parameters.

You should use contextual events to communicate between ADF regions. While it is possible to use ADF task flow parameters to communicate between regions, doing so may create direct dependencies between the regions. ADF task flow parameters may be fine for static regions, but using contextual events will allow you to implement independent communications between regions.

For example, a page may contain a region with a form for entering employee information. When the user updates the employee Id and presses the submit button, a value change contextual event is raised and the employee Id value is also passed as payLoad. On the same page, another region subscribes to and consumes the event and displays departmental information about the selected employee based on the payLoad information passed with the contextual event.

Additional Functionality for Contextual Events

You may find it helpful to understand other ADF features before you work with contextual events. Following are links to other functionality that may be of interest.

- You can also use ADF task flow parameters to communicate between ADF regions. For more information about using task flows and regions, see Using Task Flows as Regions.

Creating Contextual Events Declaratively

Contextual events in Oracle ADF has two parts, the publisher event (producer) and the handler event (consumer). In the process of creating contextual events declaratively, follow the different procedures to publish, subscribe, and consume an event.

You create contextual events by first creating and publishing the event on the producer based on a method action, action, value attribute, or list binding. On the consumer, you subscribe to the event and create a handler to process the event.

Typically, you create a parent page with regions that contain task flows and view activities. You create contextual events in one region to be published for consumer event handlers in the other region. For more information about using task flows and regions, see Using Task Flows as Regions.

> **Note:**
>
> You can also publish an action contextual event from code (for example, from within a managed bean). You can use `getBindingContainer().getEventDispatcher` which returns an instance of `EventDispatcher`. `EventDispatcher` has public APIs which you can use to programmatically raise contextual events.

How to Publish Contextual Events

You use the Properties window to create contextual events in the page where the contextual event will be produced. That is, the producer's page is the page where the contextual event is generated. The events can be published to either local or remote consumers. By default, all the events are published to local consumer. For instance, clicking a button on a page that would trigger a contextual event to be consumed by another component. The page that contains the button is the producer's page.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create contextual events. See Creating Contextual Events Declaratively.

You may also find it helpful to understand functionality that can be used with contextual events. See Additional Functionality for Contextual Events.

You will need to complete this task:

> Decide on the type of component you want to use to raise the contextual event. If you plan to use a method action binding, you must have already created the method to be dropped onto the page.

To create a contextual event:

1. In the page where you want to publish the contextual event, drag and drop a component from the Data Controls panel to the page.

   This is the component that will trigger the event. It must have a method action, action, value attribute, or list binding event.

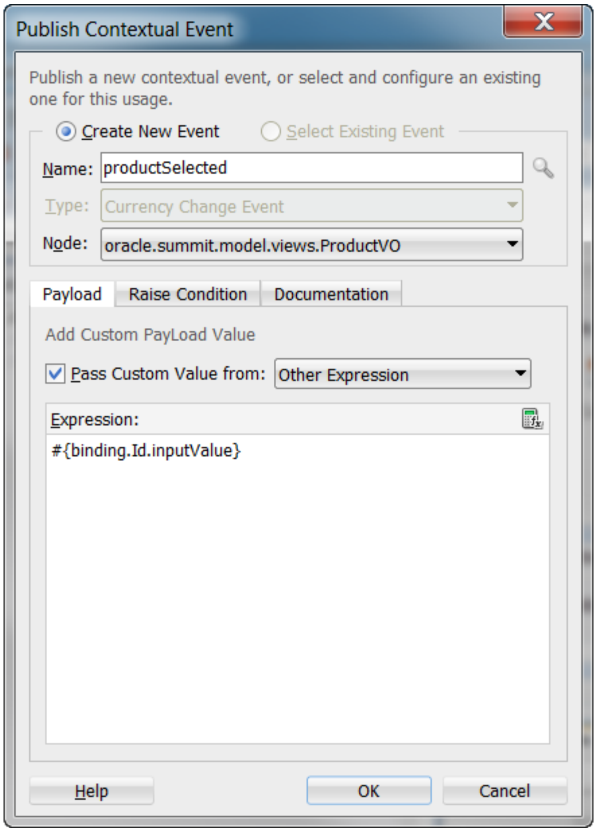
2. In the Properties window, expand the **Behavior** section.
3. In the **Publish Events** section, click the **Add** icon.
4. In the Publish Contextual Events dialog:
   a. Select **Create New Event**.
   b. Enter the name of the event.
   c. If your component is a table, tree, or tree table, a **Node** field appears for you to enter the node that will publish the event upon a change event.
   d. Select **Pass Custom Value From** if you want to pass payload data to the consumer.
   e. If you are passing payload data, select the type of data from the dropdown list. Normally, the parameter is {$payLoad}

      For instance, if you want to pass an attribute value from the producer page to the consumer page, you can select **Page Data** and select the attribute from the tree structure.
   f. You can conditionally raise the event by entering an EL expression in the **Raise Condition** tab.

      For instance, entering an expression such as `${bindings.LAST_NAME.inputValue == 'KING'}` will cause the event to be raised only if the customer's last name is KING.
   g. Click **OK**.

      The event is created on the page, but it is not ready for publishing until it is associated with the component binding.

Figure 46-5 Publish a Contextual Event



Publish Contextual Event

Publish a new contextual event, or select and configure an existing one for this usage.

● Create New Event ○ Select Existing Event

Name: productSelected

Type: Currency Change Event

Node: oracle.summit.model.views.ProductVO

Add Custom PayLoad Value

☑ Pass Custom Value from: Other Expression

Expression:

#{binding.Id.inputValue}

5. By default, all the events are dispatched to the local region. To dispatch the events to a remote region:

   a. In the Source view of the page definition file for the page that generates the event, select the event to see the event properties in the Property window.

   b. Set `dispatchMode` attribute to **remote**.

   > **Note:**
   >
   > If `dispatchMode` attribute is not set to **remote** value when a remote event is raised from a remote region, it might result in an exception, `java.io.NotSerializableException`. In this scenario, it is recommended not to set `dispatchMode` attribute to **local** value.

   c. For `customPayLoad` attribute, write an EL expression that references a serializable object.

   d. For `customPayLoadType` attribute, specify the type of custom payload of the object.

How to Subscribe to and Consume Contextual Events

You use the overview editor for page definition files to subscribe to contextual events. The overview editor displays the current region and its child regions. Only the regions that are shown can subscribe to contextual events.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create contextual events. For more information, see Creating Contextual Events Declaratively.

You may also find it helpful to understand functionality that can be used with contextual events. For more information, see Additional Functionality for Contextual Events.

You will need to complete this task:

> Create a contextual event in the page definition file, as described in How to Publish Contextual Events.

To subscribe and consume the event:

1. Create a handler to process the event and its payload data.

   In the Summit ADF sample application, the `PopulateInventoryForProduct` handler method was created in the `InventoryVO` view object by adding the code to the `InventoryVOImpl.java` file.

   The method appears in the `BackOfficeAppModuleDataControl` **application module**.

2. In the consuming page, add the components that will respond to the event.

   In the Summit ADF sample application, a DVT component is added to the page to display the inventory available in different warehouses.

   The DVT component will receive the data from the `PopulateInventoryForProduct` method.

3. In the Applications window, double-click the page definition file that will contain the event map.

   In the Summit ADF sample application, the `showInventoryPageDef.xml` page definition file will have the event map.

4. In the overview editor, click the **Bindings and Executables** tab and then click the **Add** icon for the **Bindings** section

5. In the Insert Item dialog, select **methodAction** and click **OK**.

6. In the Create Action Binding dialog:

   a. Expand the data collection where you have created your handler.

   b. Select the handler.

   c. Click **OK**.

7. In the overview editor **Contextual Events** tab, click **Subscribers** and click the **Add** icon in the **Event Subscribers** section.

8. In the Subscribe to Contextual Event dialog:

   a. Enter the event name or click the **Search** icon to start the Select Contextual Event dialog.

      If you have started the Select Contextual Events dialog, select the event from the tree and click **OK**.

   b. Select the producer or **<Any>** from the **Publisher** dropdown list. A contextual event can have more than one producer.

Selecting **<Any>** will allow the consumer to subscribe to any producer producing the selected event. In the page definition file, the `producer` attribute will be set to the wildcard "*". If your producer is in a dynamic region, you should set this field to **<Any>** so that the subscriber can consume from any producer.

c. Click the **Search** icon next to the **Handler** field.

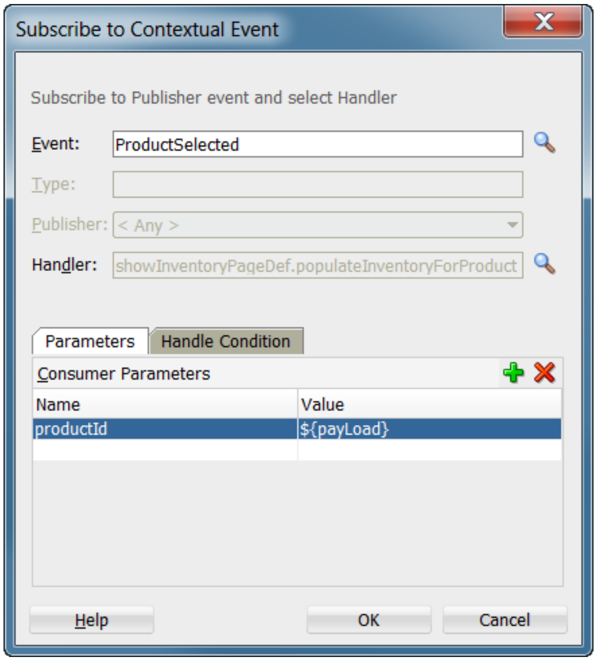
d. In the Select Handler dialog, select the event handler from the tree and click **OK**.

e. If the handler requires parameters, click the **Parameters** tab, click **Add**, and enter name-value pair as parameters.

f. If you want to conditionally handle the event, select the **Handle** tab, and enter an EL Expression that determines the conditions under which the handler will process the event.

g. Click **OK**.

Figure 46-6 Subscribe to a Contextual Event



Subscribe to Contextual Event

Subscribe to Publisher event and select Handler

Event: ProductSelected

Type:

Publisher: < Any >

Handler: showInventoryPageDef.populateInventoryForProduct

Consumer Parameters

Name	Value
productId	${payLoad}

Note:

You can edit the event map by right-clicking the page definition in the Structure window and choosing **Edit Event Map**. You can also edit event attributes in the page definition file or in the Properties window.

What Happens When You Create Contextual Events

When you create an event for the producer, JDeveloper adds an `events` element to the page definition file. Each event name is added as a child. The Event Definition for the

Producer example below shows the `productSelected` event defined in the producer page definition file as a currency change event.

```
<tree IterBinding="ProductVO1Iterator" id="ProductVO1">
      <nodeDefinition DefName="oracle.summit.model.views.ProductVO"
        Name="ProductVO10">
        <AttrNames>
          <Item Value="Id"/>
          <Item Value="Name"/>
          <Item Value="ShortDesc"/>
          <Item Value="LongtextId"/>
          <Item Value="ImageId"/>
          <Item Value="SuggestedWhlslPrice"/>
          <Item Value="WhlslUnits"/>
        </AttrNames>
        <events xmlns="http://xmlns.oracle.com/adfm/contextualEvent">
          <event name="productSelected" eventType="Currency Change Event"
           customPayLoad="#{bindings.Id.inputValue}"/>
          </events>
      </nodeDefinition>
</tree>
```

In the consumer page definition file, a method action binding is created to respond to the event. The Method Action Definition example below shows the method action binding defined in the `showInventoryPageDef` page definition file.

```
<methodAction id="populateInventoryForProduct" RequiresUpdateModel="true"
          Action="invokeMethod"
          MethodName="populateInventoryForProduct"
          IsViewObjectMethod="true"
                  DataControl="BackOfficeAppModuleDataControl"

InstanceName="data.BackOfficeAppModuleDataControl.InventoryVO1">
      <NamedData NDName="productId" NDType="java.lang.String"/>
</methodAction>
```

When the currency change occurs, the event is broadcasted to its consumers. The method action definition is defined in the consuming page definition file.

When you configure an event map, JDeveloper creates an event map entry in the corresponding page definition file. An example below shows the event map on the `showInventoryPageDef` page definition file that maps the `productSelected` event from the `showproductstaskflowdefinition1` region to the `productinventorytaskflowdefinition1` region. It also maps the `populateInventoryForProduct` handler method bindings that is defined in the `showInventoryPageDef` page definition file. The handler processes the payLoad information (containing the `productID`) and passes it to the DVT component so it can query the warehouse inventory for that product.

```
<eventMap xmlns="http://xmlns.oracle.com/adfm/contextualEvent">
    <event name="productSelected">
      <producer region="*">
        <consumer handler="populateInventoryForProduct" region="">
        <parameters>
        <parameter name="productId" value="${payLoad}"/>
        </parameters>
        </consumer>
      </producer>
    </event>
</eventMap>
```

When you set the value for `dispatchMode` to **remote**, JDeveloper adds the attribute `dispatchMode` to the `event` element in the page definition file. `CustomPayLoadType` will be the type of the payload which is being passed, it will be String if `getCustomPayLoad()` returns String. An example below shows the event element with the `dispatchMode` attribute set to **remote** value and the attributes that specify the custom payload to pass to the remote consumer.

```
<methodAction id="eventProducer" RequiresUpdateModel="true"
Action="invokeMethod" MethodName="eventProducer"
       IsViewObjectMethod="false" DataControl="Class1"
       InstanceName="bindings.eventProducer.dataControl.dataProvider"

ReturnName="data.Class1.methodResults.eventProducer_eventProducer_dataControl_dat
aProvider_eventProducer_result">
 <events xmlns="http://xmlns.oracle.com/adfm/contextualEvent">
  <event name="teste" eventType="Action Event" dispatchMode="remote"
customPayloadType="java.lang.String"
        customPayLoad="#{testBean.customPayLoad}"/>
 </events>
</methodAction>
```

How to Control Contextual Events Dispatch

You can control the dispatch of contextual events to child regions at the application level or at the page level.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create contextual events. For more information, see Creating Contextual Events Declaratively.

You may also find it helpful to understand functionality that can be used with contextual events. For more information, see Additional Functionality for Contextual Events.

To disable event dispatch:

1. To disable event dispatch at the application level, set the `dynamicEventSubscriptions` property to `false` in the `adf-config.xml` file, as shown in the following example.

   You can disable the event dispatch to regions that have an event map with producers as wildcards.

   ```
   <?xml version="1.0" encoding="windows-1252" ?>
   <adf-config xmlns="http://xmlns.oracle.com/adf/config"
        xmlns:cef="http://xmlns.oracle.com/adfm/contextualEvent">
        <cef:DynamicRegionEventsConfig dynamicEventSubscriptions="false">
        </cef:DynamicRegionEventsConfig>
   </adf-config>
   ```

2. To disable event dispatch at the individual page level, set the `dynamicEventSubscriptions` property to `false` in the associated page definition file, as shown in the following example.

   Contextual events will not be passed to the page and any of its children.

   ```
   <pageDefinition xmlns="http://xmlns.oracle.com/adfm/uimodel"
                   version="11.1.1.52.8" id="viewBPageDef"
   Package="view.pageDefs"
                   DynamicEventSubscriptions="false">
   ```

What Happens at Runtime: Contextual Events

If both the event producer and the consumer are defined in the same page definition file, then after the corresponding page is invoked and the binding container is created, the event is raised when:

- The corresponding method or action binding is executed
- A value binding is set successfully
- A range binding currency is set successfully

For a method binding, the result of the method execution forms the payload of the event, and the event is queued. In the Invoke Application phase of the JSF lifecycle, all the queued events will be dispatched. The event dispatcher associated with the binding container checks the event map (also in the binding container, as it is part of the same page definition file) for a consumer interested in that event and delivers the event to the consumer.

When the producer and consumer are in different regions, the event is first dispatched to any consumer in the same container, and then the event propagation is delegated to the parent binding container. This process continues until the parent or the topmost binding container is reached. After the topmost binding container is reached, the event is again dispatched to child-binding containers that have regions with pages that have producer set to wildcard "`*` ".

When you raise a remote event from a remote region to pass a data value by using dataControl, the `dispatchMode` attribute must be set to **remote** value to successfully deliver the data value to the remote region; otherwise, it might result in an exception, `java.io.NotSerializableException`. In this scenario, it is recommended not to set `dispatchMode` attribute to **local** value.

Creating Contextual Events Manually

Contextual events in Oracle ADF has two parts, the publisher event (producer) and the handler event (consumer). In the process of creating contextual events manually, follow a detailed procedure to publish and consume an event.

You create contextual events by first creating the event on the producer. You then determine the consumer of the event, and map the producer to the consumer.

How to Create Contextual Events Manually

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create contextual events. For more information, see Creating Contextual Events Manually.

You may also find it helpful to understand functionality that can be used with contextual events. For more information, see Additional Functionality for Contextual Events.

You will need to complete this task:

> Create a contextual event that has a method binding, action binding, value attribute binding, or list binding on the producer page.

To create a contextual event:

1. In the Applications window, double-click the page definition file that contains the binding for the producer of the event.

   A producer must have an associated binding that will be used to raise the event. For example, if a method or operation will be the producer, the associated action binding or method action binding will contain the event.

2. In the Structure window, right-click the binding for the producer and choose **Insert inside *binding name* > events** or **Insert inside *binding name* > Contextual Events > events**.

3. In the Structure window, right-click the **events** element just created, and choose **Insert inside events > event**.

4. In the Insert event dialog, enter a name for the event in the **name** field, and click **Finish**.

   The event is now created. By default, any return of the associated method or operation will be taken as the payload for the event and stored in the EL-accessible variable `${data.payLoad}`. You now need to map the event to the consumer, and to configure any payload that needs to be passed to the consumer.

5. In the Applications window, double-click the page definition file that contains the binding for the consumer.

   The binding container represented by this page provides access to the events from the current scope, including all contained binding containers (such as task flow regions). If regions or other nested containers need to be aware of the event, the event map should be in the page definition of the page in the consuming region.

6. In the Structure window, right-click the topmost node that represents the page definition, and choose **Edit Event Map**.

   > **Note:**
   >
   > If the producer event comes from a page in an embedded dynamic region, you may not be able to edit the event map using the Event Map Editor. You can manually create the event map by editing the page definition file or use **insert inside** steps, as described in Creating Contextual Events Manually.

7. In the Event Map Editor, click the **Add** icon to add an event entry.

8. In the Add New EventMap Entry dialog, do the following:

   a. Use the **Producer** dropdown menu to choose the producer.

   b. Use the **Event Name** dropdown menu to choose the event.

   c. Use the **Consumer** dropdown menu to choose the consumer. This should be the actual method that will consume the event.

   d. If the consuming method or operation requires parameters, click the **Add** icon.

      In the **Param Name** field, enter the name of the parameter expected by the method. In the **Param Value** field, enter the value. If this is to be the payload from the event, you can access this value using the `${data.payLoad}` expression. If the payload contains many parameters and you don't need them all, use the ellipses button to open the Expression Builder dialog. You can use this dialog to select specific parameters under the **payload** node.

You can also click the **Parameters** ellipses button to launch the selection dialog.

e. Click **OK**.

9. In the Event Map Editor, click **OK**.

Creating Contextual Events Using Managed Beans

Oracle ADF supports publishing an action contextual event from code such as from within a managed bean. You can also dynamically create and handle contextual events using managed beans.

How to Create Contextual Events Using Managed Beans

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create contextual events. For more information, see Creating Contextual Events Using Managed Beans.

You may also find it helpful to understand functionality that can be used with contextual events. For more information, see Additional Functionality for Contextual Events.

To create contextual events using managed beans:

1. Create a method in a managed bean to generate the contextual event.

   As the following example shows managed bean code that perform an action when a button is pressed. The myActionPerformed method is invoked and calls methods to generate the contextual event with "myString" as the payLoad:

```
BindingContainer bc BindingContext.getCurrent().getCurrentBindingsEntry();
JUCtrlActionBinding actionBnd =
  (JUCtrlActionBinding)bc.getControlBinding("eventProducer");
...
((DCBindingContainer)bc).getEventDispatcher().queueEvent(actionBnd.
   getEventProducer(),"myString");
```

2. Bind the producer component to the method in the managed bean.

   As the following example shows code for a producer associated with a command button that invokes an action binding and the consumer is an outputText component that displays a string. They are both on the same page.

```
<af:form id="f1">
     <af:button value="eventProducerButton1" id="cb1"
                action="#{MyBean.myActionPerformed}"/>
     <af:panelLabelAndMessage
label="#{bindings.return.hints.label}"id="plam1">
          <af:outputText value="#{bindings.return.inputValue}" id="ot1"/>
     </af:panelLabelAndMessage>
</af:form>
```

What Happens When You Create Contextual Events Using Managed Beans

The page definition file contains the method action bindings for the producer, the consumer, and the event map, like this:

```
<executables>
    <variableIterator id="variables">
      <variable Type="java.lang.String" Name="eventConsumer_return"
                IsQueriable="false" IsUpdateable="0"
                DefaultValue="${bindings.eventConsumer.result}"/>
    </variableIterator>
</executables>
<bindings>
     <methodAction id="eventProducer"
                  InstanceName="AppModuleDataControl.dataProvider"
                  DataControl="AppModuleDataControl" RequiresUpdateModel="true"
                  Action="invokeMethod" MethodName="eventProducer"
                  IsViewObjectMethod="false"
                  ReturnName="AppModuleDataControl.methodResults.eventProducer_
                       AppModuleDataControl_dataProvider_eventProducer_result">
            <events xmlns="http://xmlns.oracle.com/adfm/contextualEvent">
                  <event name="myEvent"/>
            </events>
     </methodAction>
     <methodAction id="eventConsumer" RequiresUpdateModel="true"
                  Action="invokeMethod" MethodName="eventConsumer"
                  IsViewObjectMethod="false" DataControl="AppModuleDataControl"
                  InstanceName="AppModuleDataControl.dataProvider"
                  ReturnName="AppModuleDataControl.methodResults.eventConsumer_
                        AppModuleDataControl_dataProvider_eventConsumer_result">
               <NamedData NDName="str" NDValue="test" NDType="java.lang.String"/>
    </methodAction>
    <attributeValues IterBinding="variables" id="return">
        <AttrNames>
             <Item Value="eventConsumer_return"/>
        </AttrNames>
    </attributeValues>
</bindings>
<eventMap xmlns="http://xmlns.oracle.com/adfm/contextualEvent">
    <event name="myEvent">
      <producer region="eventProducer">
        <consumer region="" handler="eventConsumer">
          <parameters>
            <parameter name="test" value="${data.payLoad}"/>
          </parameters>
        </consumer>
      </producer>
    </event>
</eventMap>
```

How to Dynamically Create and Handle Contextual Events Using Managed Beans

You can programmatically create contextual event definitions, register the event producer and even trigger the contextual event. This is useful when you build task flows which need to trigger different events based on the context where the event is used (for example, when the client that hosts the task flow may change).

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create contextual events. For more information, see Creating Contextual Events Using Managed Beans.

You may also find it helpful to understand functionality that can be used with contextual events. For more information, see Additional Functionality for Contextual Events.

To create and handle contextual events dynamically using managed beans:

1. Create a method in a managed bean to generate the contextual event.

   The following sample shows managed bean code that exercises the `EventDispatcher` API to perform an action when a button is pressed.

   ```
   DCBindingContainer bc = (DCBindingContainer)
   BindingContext.getCurrent().getCurrentBindingsEntry();
   bc.getEventDispatcher().queueEvent(new CustomEventProducer(),
   someCustomEventPayLoad);
   bc.getEventDispatcher().processContextualEvents();
   ```

   The event producer class `CustomEventProducer` used in this above sample defines the contextual event `JDemoEvent`.

   ```
   import oracle.adf.model.events.EventProducer;

   public class CustomEventProducer implements EventProducer {
       public CustomEventProducer() {
           super();
       }
       public String getId() {
           return "JDemoEvent";
       }

       public ArrayList getEventDefinitionsList() {
           ArrayList eventDefList = new ArrayList();

           EventDefinitionImpl demoEvent = new EventDefinitionImpl();
           demoEvent.setEventName("JDemoEvent");
           eventDefList.add(demoEvent);
           return eventDefList;
       }

   }
   ```

2. Bind the producer component to the method in the managed bean.

   The following sample shows the producer page source associated with a command button that invokes an action listener method `produceDynamicContextualEvent()` on the above managed bean.

   ```
   <af:panelGroupLayout id="pgl1">
           <af:button text="Produce Contextual Event" id="b5"

   actionListener="#{viewScope.EventProducer.produceDynamicContextualEvent}"/>
   </af:panelGroupLayout>
   ```

3. Bind the consumer component to the event handler method for the contextual event. (This is an alternative to relying on data controls to perform the binding.)

The following sample shows the consumer page source as an outputText component that displays a string set by the event handler method `setOutputTextDeptName()`.

```
<af:panelGroupLayout id="pgl1">
    <af:outputText value="Got the following via Contextual Event :"
id="ot1"/>
    <af:outputText value="#{backingBeanScope.EventHandler.deptName}" id="ot2"

binding="#{backingBeanScope.EventHandler.outputTextDeptName}"
clientComponent="true"/>
</af:panelGroupLayout>
```

Note that when you create a managed bean as event handler for contextual events not using data controls, you may want to create the `methodAction` bindings manually. This will simplify your task when you want to refresh a UI component or invoke some managed bean code as part of the contextual event. The following sample shows a `methodAction` binding and an `eventMap` like the ones that you would create in the consuming region's page definition.

```
<pageDefinition xmlns="http://xmlns.oracle.com/adfm/uimodel"
id="consumerPageDef" Package="view.pageDefs">
     ...
     <bindings>
        <methodAction DataControl="AppModuleDataControl"
id="EventHandlerMethod" InstanceName="${backingBeanScope.EventHandler}"
                      MethodName="handleEvent">
            <NamedData NDName="eventPayLoad" NDType="java.lang.Object"/>
        </methodAction>
    </bindings>
    <eventMap xmlns="http://xmlns.oracle.com/adfm/contextualEvent">
        <event name="JDemoEvent">
            <producer region="*">
                <consumer region="" handler="EventHandlerMethod">
                    <parameters>
                        <parameter name="eventPayLoad"
value="#{data.payLoad}"/>
                    </parameters>
                </consumer>
            </producer>
        </event>
    </eventMap>
</pageDefinition>
```

Creating Contextual Events Using Plain Java Objects (POJOs)

Oracle ADF provides bounded task flows that can take any number of parameters and can provide a simple feedback to its caller in the form of an outcome. It also has a return parameter, allowing the task flow to return a Java object.

A bounded task flow embedded in a region may have several views that need to be exposed as possible consumers of an contextual event. Instead of creating a Plain Java Object (POJO), a data control, and an event handler for each view, you can create a single POJO and a single data control to process events for all the views in the task flow.

In this approach, the event receiver method dispatches the contextual event request by using the managed bean reference to invoke the `handleEvent` method and passes the payLoad for processing. The managed bean, defined in backing bean scope, can have JSF component references that allow the bean to refresh the dependent user interface.

The following conditions must be satisfied for this approach to work:

- The event receiver method must pass the payLoad as an argument to the method that handles the message.
- The event receiver method must have a second argument that is used to pass an object reference to a managed bean associated with the subscriber region task flow.
- The managed bean passed as a reference must contain a commonly known public method that accepts the event message as input argument. This method handles the event.

How to Configure a Generic Callback Message Handler for a Task Flow

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create contextual events. For more information, see Creating Contextual Events Using Plain Java Objects (POJOs).

You may also find it useful to understand functionality that can be used with contextual events. For more information, see Additional Functionality for Contextual Events.

To create the generic event receiver and managed bean event handler:

1. Create a POJO data control.

   The POJO data control should appear in the Data Control panel.

   For more information about data controls, see the Creating and Configuring Bean Data Controls section in *Developing Applications with Oracle ADF Data Controls*.
2. Create an event receiver class (exposed in the POJO data control) with an event handler method that defines two arguments.

   The first argument is for the payLoad delivered by the contextual event. The second argument is callback handler reference to a managed bean that is implementing the contextual event handler interface.
3. Create a managed bean with backing bean scope that implements the `ContextualEventHandler` interface.

   Implementing the `ContextualEventHandler` interface includes defining the call back method, for example, `handleEvent()`, that will be called by the generic contextual event receiver to pass control to the managed bean.
4. Create a method binding in the page definition file of the view within the task flow that will receive the event.

   a. In the Applications window, double-click the page definition file that will contain the event map.

   b. In the overview editor, click the **Bindings and Executables** tab and then click the **Add** icon for the **Bindings** section
   c. In the Insert Item dialog, select **methodAction** in the **Generic Bindings** category, and select the POJO data control that exposes the event receiver method.
   d. Use the EL Expression Builder to reference the managed bean that was created for the `ContextualEventHandler` argument. Leave the `payLoad` argument blank as this will be provided by the contextual event mapper.

5. Subscribe to the contextual event by defining it in the same page definition file where the method binding was declared.

   For general instructions on subscribing to contextual events, follow Step 7 to Step 8 in How to Subscribe to and Consume Contextual Events.

This is an implementation for a generic and reusable contextual event handler that invokes a method on a configured managed bean to have the contextual event payload handled in the context of a view to be changed in response of an event.

Creating Contextual Events Using JavaScript

Every action and method binding that is accessible from a managed bean can be invoked from JavaScript. ADF Faces provides an `af:serverListener` operation component that can be used to call a managed bean method from client-side JavaScript.

To invoke the `af:serverListener` operation component using the referenced managed bean method, use the `BindingContext` object to look up the current `BindingContainer` and to access the `OperationBinding` or a `JUEventBinding` binding. The `af:serverListener` component can also be used to send a message `payload` from the browser client to the managed bean method.

Creating the Event Map Manually

In Oracle ADF, events are configured in the page definition file for the page or region that will raise the event (the producer). To associate the producer with the consumer that will perform some operations based on the event, you create an event map also in the page definition file.

Under most circumstances, you can create the event map using the Event Map Editor as described in Creating Contextual Events Manually. However, in situations such as when the producer event is from a page in an embedded dynamic region, the Event Map Editor at design time cannot obtain the necessary information to create an event map.

How to Create the Event Map Manually

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create contextual events. For more information, see Creating the Event Map Manually.

You may also find it useful to understand functionality that can be used with contextual events. For more information, see Additional Functionality for Contextual Events.

To create the event map manually:

1. In the Applications window, double-click the page definition file that contains the binding for the consumer.
2. In the Structure window, right-click the topmost node that represents the page definition, and choose **Insert inside *pagedef name* > eventMap**.
3. In the Structure window, select the **eventMap** node, right-click and choose **Insert inside eventMap > event**.
4. In the Insert Event dialog, enter the name of the event and click **OK**.

   Repeat steps 3 and 4 to add more events.
5. Select the **event** node, right-click and choose **Insert inside event > producer**.
6. In the Insert Producer dialog, enter the name of the binding that is producing this event and click OK.

   You can also enter the name of the region which has the event producer, in which case all the consumers specified under this tag can consume the event. You can also enter "*" to denote that this event is available for all consumers under this tag.
7. Select the **producer** node, right-click, and choose **Insert inside producer > consumer**.
8. In the Insert Consumer dialog, enter the name of the handler that will consume the event and click **OK**.

   Repeat steps 7 and 8 to add more consumers.
9. If there are parameters being passed, add the parameter name and value.
   a. Select the **consumer** node, right-click, and choose **Insert inside consumer > parameters**.
   b. Select the **parameters** node, right-click, and choose **Insert inside parameters > parameter**.
   c. In the Insert Parameter dialog, enter the name of the parameter and the value of the parameter and click **OK**. The value can be an EL expression.

      Repeat this step to add more parameters.

Registering a Custom Dispatcher

By default, the contextual event framework uses `EventDispatcherImpl` to dispatch events that would traverse through the regions. You can create a custom event dispatcher to override the default event dispatcher to provide custom behaviors.

After you have created the custom event dispatcher, you must register it in the `Databindings.cpx` file to override the default dispatcher.

How to Register a Custom Dispatcher

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create contextual events. For more information, see Registering a Custom Dispatcher.

You may also find it useful to understand functionality that can be used with contextual events. For more information, see Additional Functionality for Contextual Events.

To register a custom event dispatcher:

1. Create a custom event dispatcher Java class based on the `EventDispatcher` class.
2. Register the custom event dispatcher in the `Databindings.cpx` file with a fully qualified name using the following format:

   ```
   EventDispatcher="package_name.CustomEventDispatcher_name"
   ```

   As the following example shows the code for a custom event dispatcher called `NewCustomEventDispatcher` created in package `NewPackage`.

   ```
   <Application xmlns="http://xmlns.oracle.com/adfm/application"
                version="11.1.1.51.60" id="DataBindings"
   SeparateXMLFiles="false"
                Package="project3" ClientType="Generic"
                EventDispatcher="NewPackage.NewCustomEventDispatcher">
   ```
3. Create the event in the producer's page definition. For more information, see Creating Contextual Events Declaratively, or Creating Contextual Events Manually.
4. Create the event map in the consumer region if the consumer is in a dynamic region. If the consumer is not in a dynamic region, you can also specify the event map in the parent page which holds both the producer and consumer regions. For more information, see Creating the Event Map Manually.

Part VI
Completing Your Application

This part describes tasks that developers can perform to secure, refine, test, and deploy Fusion web applications.

Part VI contains the following chapters:

- Enabling ADF Security in a Fusion Web Application
- Testing and Debugging ADF Components
- Refactoring a Fusion Web Application
- Reusing Application Components
- Customizing Applications with MDS
- Allowing User Customizations at Runtime
- Using the Active Data Service
- Configuring High Availability for Fusion Web Applications
- Deploying Fusion Web Applications
- Using State Management in a Fusion Web Application
- Tuning Application Module Pools

47

Enabling ADF Security in a Fusion Web Application

This chapter describes how you can enable ADF Security in the Fusion web application to define resource grants for Oracle ADF resources and to restrict the user's ability to view web pages associated those resources.
This chapter includes the following sections:

About ADF Security

Security is an important part of any enterprise application. Security implementation in an ADF application decides who can access the application and what they can do once they are logged in. You can visually enable security in the different layers of your Fusion web application.

The ADF Security framework is the preferred technology to provide authentication and authorization services to the Fusion web application. ADF Security is built on top of the Oracle Platform Security Services (OPSS) architecture, which itself is well-integrated with Oracle WebLogic Server. While other security-aware models exist that can handle user login and resource protection, ADF Security is ideally suited to provide declarative, permission-based protection for **ADF bounded task flows**, for top-level web pages that use **ADF bindings** (pages that are not contained in a bounded task flow), and at the lowest level of granularity, for rows of data defined by ADF **entity objects** and their attributes. In this document, these specific resources that the ADF Security framework protects are known as **ADF security-aware resources**.

You enable ADF Security for Fusion web applications when you run the Configure ADF Security wizard, as described in Enabling ADF Security. The wizard configures ADF Security for the entire Fusion web application, so that any web page associated with an ADF security-aware resource is protected by default. This means that after you enable ADF Security, your application is locked down so that the pages are considered secure by default.

After you enable ADF Security you must grant users access rights so that they may view the web pages of the Fusion web application. Access rights that you grant users are known as a **security policy** that you specify for the page's corresponding ADF security-aware resource. Ultimately, it is the security policy on the ADF resource that controls the user's ability to enter a task flow or view a web page.

Because ADF Security is based on Java Authentication and Authorization Service (JAAS), security policies identify the principal (the user or application role), the ADF resource, and the permission (an operation defined by the resource's ADF permission class). For example, the Summit sample application for ADF task flows secures the web pages contained by the `customer-task-flow` task flow to grant access only to logged-in users (also known as **authenticated users**). At runtime, the ADF Security framework performs authorization checking against the task flow's security policy to determine the user's right to complete the view operation. In this case, the security policy must grant the view permission to the user if they are to complete the checkout process.

To simplify the task of defining security policies for users and ADF resources, ADF Security defines a containment hierarchy that lets you define one security policy for the ADF bounded task flow and its contains web pages. In other words, when you define the security policy at the level of the bounded task flow, you protect the flow's entry point and then all pages within that flow are secured by the policy it defines. Additionally, instead of granting access to individual users, you group users into application roles and grant the view permission to the role.

Specifically, you will define security policies in the Fusion web application for the following ADF security-aware resources to make web pages accessible to users:

- **ADF bounded task flow** protects the entry point to the task flow, which in turn controls the user's access to the pages contained by the flow

  For example, a series of web pages may guide new customers through a registration process and the bounded task flow controls page navigation for the process. For a description of bounded task flows, see About Bounded Task Flows.

  The **ADF unbounded task flow** is not an ADF security-aware component and thus does not participate in authorization checks. When you need to protect the constituent pages of an unbounded task flow, define grants for the page definition files associated with the pages instead.

- ADF **page definition files** associated with web pages not contained by a bounded task flow

  For example, a web page may display a summary of best selling products with data coordinated by the ADF bindings of the page's associated ADF page definition file. For a description of page definitions and ADF bindings, see Working with Page Definition Files.

- ADF **entity objects** and attributes of entity objects that reference rows of data and help define collections for display in the user interface

  For example, a web page may display an **ADF Faces** table component that displays columns that ADF bindings map to the attributes of an entity object as its data source. In the case of entity objects, enabling ADF Security does not automatically secure entity objects rows. The data will remain accessible to users until you define a security policy to explicitly protect the entity object or its attributes. For a description of entity objects, see About Entity Objects.

JDeveloper tools support iterative development of security so you can easily create, test, and edit security policies that you create for ADF resources. You can proceed to create test users in JDeveloper and run the application in Integrated WebLogic Server to simulate how end users will access the secured resources. This chapter describes how to configure the repository of user identities and login credentials known as the **identity store**.

> **Note:**
>
> References to the identity store in this chapter are always in the context of *test* user identities that you create for the purpose of running in Integrated WebLogic Server. Typically, you would not migrate these users to the staging environment when you deploy to Oracle WebLogic Server, as described in Preparing the Secure Application for Deployment.

To avoid a situation where you have enabled ADF Security but have not yet defined security policies to grant access to test users, the Configure ADF Security wizard lets you grant temporary view rights to all existing ADF resources (a view permission grant will be added to the security policy for each ADF resource). This wizard option gives you the choice to disable automatic grants and proceed to define security policies for ADF resources as you create each resource or to enable automatic view grants and gradually replace these grants with security policies that you define. To understand iterative security development choices, see ADF Security Process Overview.

> **Tip:**
>
> Before you enable ADF Security and define security policies for the ADF security-aware resources, you will want to understand the rules that govern ADF authorization checking. Understanding these rules will help you to implement the security you intend. For a discussion of these rules, see ADF Security Use Cases and Examples.

Integration of ADF Security and Java Security

The ADF Security model for securing Fusion web application resources is not based on the URL mapping of a security constraint as exemplified by the Java EE security model. In actual practice, security constraints are not feasible for securing a JavaServer Faces (JSF) web application where page navigation is not supported by specific page URLs. For example, when the user navigates to the next page in a task flow, the URL remains the same throughout the flow. As each new page is displayed, there is no means to trigger a URL-based security constraint.

Instead, ADF Security implements a Java Authentication and Authorization Service (JAAS) security model. The JAAS model is policy-based since JAAS is built on the existing Java security model and integrates with any JAAS implementation, including the Oracle Platform Security Services (OPSS) implementation of the JAAS service. Whereas applications that utilize URL security constraints are security-unaware because they rely on the Java EE container to manage security, Fusion web applications require an explicit call to the ADF Security framework to authorize access to resources based on user-defined policies. Thus, when you enable ADF Security and define access policies for ADF resources, your application is security-aware.

ADF Security simplifies the implementation of a JAAS authorization model. This implementation minimizes the work needed to create a security-aware application by exposing security policies on ADF resources in a declarative fashion and performing authorization checks on these resources at runtime.

The policy store in JDeveloper is file-based and contains a list of entries known as **grants**, which define the security policy for the ADF resource. The grant entry includes all the permissions granted to the user to perform operations on the protected resource, for instance, accessing a web page associated with an ADF bounded task flow. Permissions are granted in the policy store to an application role principal.

ADF Security expands on the JAAS model by allowing you to define grants using the actions specified by the ADF Security framework permission classes. These classes are specific to the ADF resource and map the actions to an operation supported by the resource. The policy store for the Fusion web application therefore contains grants that specify:

- One or more permissions that associate an action defined by the resource's permission class with an instance of the ADF resource in the application (currently, only the `view` action is supported for bounded task flows and page definitions resources)
- The grantee, which is an application role defined by your application that you populate with member users or, optionally, enterprise roles for whom you wish to confer the same access rights

In the case of entity objects, the permission class defines `read`, `delete`, and `update` actions that correspond to the `read`, `removeCurrentRow`, and `update` operations of the entity object.

For a description of the ADF permission classes and supported actions, see ADF Security Permission Grants.

ADF Security Use Cases and Examples

The use of ADF Security enables web applications to easily adjust to real-world business security requirements, because rather than securing paths to application resources, you secure the view operation on ADF resources with JAAS. JAAS-based ADF Security provides:

- Declarative security support for ADF resources, such as the bounded task flow

  Because Java EE security is URL-based or page-based, it is not possible to have a navigation control without custom code. With ADF Security, you can control whether or not the user can enter a task flow. Thus, a single security policy for a task flow can control access to multiple web pages. Additionally, because declarative security lets you secure the ADF resource, not the access path, you can reuse the ADF resource elsewhere in the application and it will remain secured.

- Simplified permission assignment by using application roles that allow for the inheritance of permissions

  While Java EE security roles that are used by Java EE security constraints are flat, JAAS permissions are granted to application roles, which can be nested and may be mapped to enterprise roles that the Oracle WebLogic Server domain defines.

- Utility methods for use in EL expressions to access ADF resources in the security context

You can use the ADF Security EL expression utility methods to determine whether the user is allowed to perform a known operation. For example, you can determine whether the user is allowed to view a particular task flow.

Additionally, JDeveloper enables you to quickly create test users and passwords to test security in Integrated WebLogic Server. When you are ready to deploy to Oracle WebLogic Server, you can migrate the application-specific authorization policies to the server and the administrator can configure the application to use an LDAP user repository.

Table 47-1 summarizes the effect that enabling ADF Security has on the application and the various ADF security-aware resources. For further discussion about how you can work most effectively with ADF Security, see Best Practices for Working with ADF Security.

Table 47-1 Summary of ADF Security-Aware Resources

ADF Resource	How ADF Enforces Security	How to Grant Access
Bounded task flows in all user interface projects	Protected by default. Requires a grant to allow users to enter the bounded task flow.	Define the grant for the task flow. Do *not* define grants for individual page definition files associated with the web pages of the bounded task flow.
Page definition files in all user interface projects	Protected by default. Requires a grant to allow users to view the page associated with the page definition.	If the web page is contained by a bounded task flow, define the grant for the task flow. Define the grant for the page definition *only* when the web page is not contained by a bounded task flow or when the page is contained by an unbounded task flow. Note that the unbounded task flow is not an ADF security-aware component and allows no grants.
Entity objects in the data model project	Not protected by default. Requires a grant to prevent access by users.	Define a grant on the entity object to protect data only if you need to control access at the level of the entire data collection. The data displayed by *all* components in the user interface that reference the protected entity object will be protected. Use entity-level security carefully. Instead, consider defining security at the level of the entity attribute. Note that grants in the data model project are saved as metadata on the entity object itself and do not appear in the ADF policy store.

Table 47-1 (Cont.) Summary of ADF Security-Aware Resources

ADF Resource	How ADF Enforces Security	How to Grant Access
Attributes of entity objects in the data model project	Not protected by default. Requires a grant to prevent access by users.	Define a grant on the entity object attribute to protect data when you need to control access at the level of the columns of the data collection. The data displayed by *all* components in the user interface that reference the protected entity attribute will be protected. Note that grants in the data model project are saved as metadata on the entity object itself and do not appear in the ADF policy store.

Additional Functionality for ADF Security

You may find it helpful to understand other **Oracle ADF** features before you start working with ADF Security. Following are links to other functionality that may be of interest.

- To understand the security features of Oracle Platform Security Services, see Introduction to Oracle Platform Security Services in *Securing Applications with Oracle Platform Security Services*.

ADF Security Process Overview

The Oracle ADF Security framework provides several out-of-the-box features to make it easier for developers to code secure ADF applications. ADF Security settings are stored in the application-wide jazn-data.xml data file.

You work in JDeveloper when you want to secure the ADF resources of your Fusion web application. ADF Security will protect your application's bounded task flows and any web pages contained in an unbounded task flow. You enable this protection by running the Configure ADF Security wizard and later by defining ADF security policies to define user access rights for each resource.

As you create the user interface for your application, you may run the Configure ADF Security wizard at any time. You may choose to:

- Iterate between creating web pages in the UI project and defining security policies on their associated ADF resources
- Complete all of the web pages in the UI project and then define security policies on their associated ADF resources

> **Note:**
>
> Before you proceed to secure the Fusion web application, you should become familiar with the ADF security model, as described in ADF Security Use Cases and Examples.

The iterative design and test process is supported by a variety of design time tools.

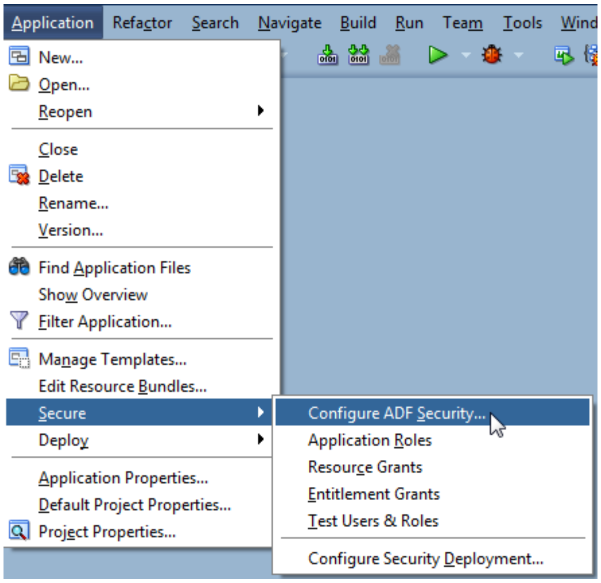
Each time you create a new bounded task flow or ADF page definition file in your user interface projects, the new ADF resource will be visible in the overview editor for the `jazn-data.xml` file. This editor is also called the overview editor for security policies. You use the overview editor to define security policies for ADF resources associated with web pages for the entire application. You can also use the overview editor to sort ADF resources and easily view those that have no security policy yet defined.

You use another editor to provision a few test users in the ADF identity store. The identity store you create in JDeveloper lets you define user credentials (user ID and password). The editor also displays the relationship between users you create and the application roles that you assign them to for the purpose of conferring the access rights defined by ADF security policies.

At design time, JDeveloper saves all policy store and identity store changes in a single file for the entire application. In the development environment, this is the `jazn-data.xml` file. After you configure the `jazn-data.xml` file using the editors, you can run the application in Integrated WebLogic Server and the contents of the policy store will be added to the domain-level store, the `system-jazn-data.xml` file, while the test users will be migrated to the embedded LDAP server that Integrated WebLogic Server uses for its identity store. The domain-level store allows you to test the security implementation by logging on as test users that you have created.

You access all design time tools for security under the main menu, using the **Application > Secure** menu, as shown in Figure 47-1.

Figure 47-1 Accessing the ADF Security Design Time Tools



Design Phase

To enable ADF Security and set up the policy store for the application that you will run in JDeveloper:

1. Enable ADF Security to allow dynamic authentication and enforce authorization by running the Configure ADF Security wizard.

When you run the wizard, if you choose to enable *only* dynamic authentication, skip the remaining design phase steps. The wizard configures files that integrate the security framework with OPSS on Oracle WebLogic Server.

2. Create an ADF security-aware resource, such as a bounded task flow with constituent web pages (or regions) or a top-level web page (or **region**) that is designed using ADF bindings.

   Note: After you run the Configure ADF Security wizard, any web page associated with an ADF security-aware resource will be protected. This means that you must define security policies to make the web pages accessible *before* you can run the application and test security.

3. Associate the ADF security-aware resource with one or more application roles that you create.

   Application roles you create are specific to the application and let you confer the same level of access to a set of users (also known as *member users*). In the test phase you will create some users and add them as members to the application roles you created.

4. Grant view permission to the ADF security-aware resource and each of its associated application roles.

   The grant confers access rights to the application role's member users. Without the grant, the user would not be able to access the ADF security-aware resource. In the test phase, you will create some users and add them to your application roles.

Testing Phase

To provision the identity store and test security using Integrated WebLogic Server:

1. Create some users and, optionally, create their enterprise roles.

   You will log in to the application using the user ID and password you define. An **enterprise role** is a logical role that lets you group users and associate these groups with application roles. The enterprise role is not needed for testing. For more information, see What You May Need to Know About Enterprise Roles and Application Roles.

2. Associate the users you created and, optionally, the enterprise roles, with one or more application roles.

   A member user may belong to more than one application role when you wish to confer the access right granted to multiple application roles.

3. Optionally, replace the default login page with a custom login page.

   The default login page generated by the Configure ADF Security wizard cannot utilize ADF Faces components. It is provided only as a convenience for testing ADF security policies. Your custom login page may be designed with ADF Faces components.

4. Run the application in JDeveloper and access any ADF security-aware resource.

   The first time you attempt to access an ADF security-aware resource, the security framework will prompt you to log in.

5. Log in and check that you are able to access the page and its resources as you intended.

   After you log in, the security framework checks the user's right to access the resource. For example, if you receive an unexpected 401 unauthorized user error, verify that you have created grants as suggested in Best Practices for Working with ADF Security.

Preparation for Staging

To prepare the secure application for deployment to Oracle WebLogic Server in a staging or production environment:

1. Remove any grants to the `test-all` role for all ADF security-aware resources and replace with grants that you define.

   Because ADF resources are secure by default, developers testing the application will be granted view access only after security policies are defined. The Configure ADF Security wizard gives you the option to generate grants to the `test-all` role that will make all ADF resources accessible. To avoid compromising enterprise security, you must eventually replace all temporary grants to the `test-all` role with explicit grants that you define.

2. Remove all user identities that you created.

   JDeveloper must not be used as an identity store provisioning tool, and you must be careful not to deploy the application with user identities that you create for testing purposes. Deploying user identities with the application introduces the risk that malicious users may gain unintended access. Instead, rely on the system administrator to configure user identities through the tools provided by the domain-level identity management system.

3. Confirm that the application roles shown in the policy store are the ones that you want an administrator to eventually map to domain-level groups.

4. Decide whether or not you what to define a security constraint to protect ADF Faces resource files.

   Resource files including images, style sheets, and JavaScript libraries are files that the Fusion web application loads to support the individual pages of the application. These files are not secured by ADF Security, but you can secure their Java EE access paths if you require all users to be authenticated before they can access the application.

5. Migrate the finalized policy store and credentials store to the target server.

   Application policies and credentials can be automatically migrated to the domain policy store when the application is deployed to a server in the Oracle WebLogic environment. Support to automatically migrate these stores is controlled by the target server's configuration. If Oracle Enterprise Manager is used to perform the deployment outside of JDeveloper, then the migration configuration settings can be specified in that tool. For information about migrating the `jazn-data.xml` security policies and the `cwallet.sso` credentials, see the Configuring the OPSS Security Store chapter in *Securing Applications with Oracle Platform Security Services*.

Enabling ADF Security

The Configure ADF Security wizard allows you to enable authentication and authorization separately.

To simplify the configuration process which allows ADF Security to integrate with OPSS, JDeveloper provides the Configure ADF Security wizard. The wizard is the starting point for securing the Fusion web application using ADF Security. The wizard is an application-level tool that, once run, will enable ADF Security for all user interface projects that your application contains.

> **Note:**
>
> Because the Configure ADF Security wizard enables ADF Security for all user interface projects in the application, after you run it, users will be required to have authorization rights to view any web page contained by a bounded task flow and all web pages associated with an ADF page definition. Therefore, after you run the wizard, the application is essentially locked down until you define security policies to grant view rights to the user. For an overview of the process, see ADF Security Process Overview.

How to Enable ADF Security

The Configure ADF Security wizard allows you to choose to enable authentication and authorization separately. You may choose to:

- Enable *only* user authentication.

  Although ADF Security leverages Java EE container-managed security for authentication, enabling only authentication means that you want to use the ADF authentication servlet to support user login and logout, but that you intend to define container-managed security constraints to secure web pages.

- Enable user authentication and also enable authorization.

  Enabling authorization means you intend to control access to the Fusion web application by creating security policies on ADF resources.

The ADF Security framework supports these two choices to give you the option to implement Java EE Security and still be able to support login and logout using the ADF authentication servlet. The benefit of enabling the ADF authentication servlet is that the servlet will automatically prompt the user to log in the first time the application is accessed. The ADF authentication servlet also allows you to redirect the user to a defined start page after successful authentication and does not require passing the target page on the request URL. You will also be able to manage the page redirect when the user logs out of the application. These redirect features provided by ADF Security are not available using only container-managed security.

Note that ADF Security does not perform authentication, but relies on the Java EE container to invoke the configured login mechanism, as described in What Happens at Runtime: How ADF Security Handles Authentication.

> **Best Practice:**
>
> Because Java EE security constraints cannot interact with the task flow to secure the current page of a task flow, container-managed security is not a useful solution when your application is designed with **ADF task flows**. When you use ADF task flows, select the **ADF Authentication and Authorization** option in the Configure ADF Security wizard. This option will allow you to define security policies to protect the task flows of your application.

Because ADF Security delegates authentication to the web container, when you run the Configure ADF Security wizard, the wizard prompts you to configure the authentication method that you want the web container to use. The most commonly used types of authentication are HTTP Basic Authentication and Form-Based Authentication. Basic authentication uses the browser login dialog for the user to enter a user name and password. Note that with basic authentication, the browser caches credentials from the user, thus preventing logout. Basic authentication is useful when you want to test the application without requiring a custom login page. Form authentication allows the application developer to specify a custom login UI. If you choose Form-based authentication, you can also use the wizard to generate a simple login page. The default login page is useful for testing your application with Integrated WebLogic Server.

> **Note:**
>
> Because the generated login page is a simple JSP or HTML file, you will not be able to modify it with ADF Faces components. For information about replacing the default login page with a custom login page that uses ADF Faces components, see Creating a Login Page.

Before you begin:

It may be helpful to have an understanding of the Configure ADF Security wizard. For more information, see Enabling ADF Security.

To enable ADF Security for the application:

1. In the main menu, choose **Application** and then **Secure > Configure ADF Security**.
2. In the ADF Security page, leave the default **ADF Authentication and Authorization** option selected. Click **Next**.

   When you run the wizard with the default option selected, your application will enforce authorization for ADF security-aware resources. Enforcing authorization for ADF resources means that you intend to define security policies for these resources to make the web pages of your application accessible. Until you do so, all pages that rely on the ADF bounded task flows and ADF page definitions will remain protected.

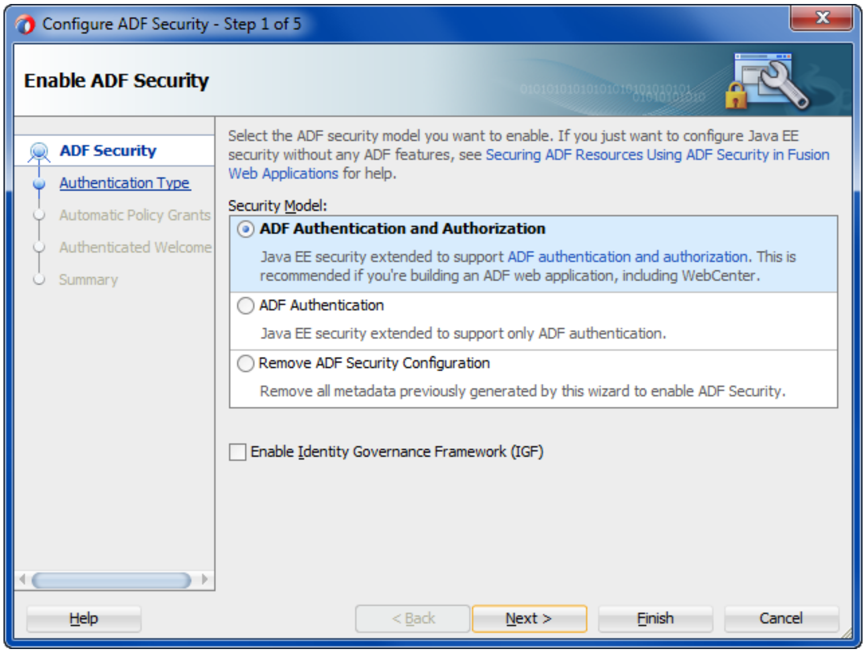
   The other two wizard options to configure ADF Security should not be used when you want to enable ADF Security. Those options allow you to temporarily disable

ADF Security and run your application without security protection, as described in Disabling ADF Security.

Specifically, the first page of the wizard lets you choose among three options, with the default option set to enable **ADF Authentication and Authorization**, as shown in Figure 47-2:

- **ADF Authentication and Authorization** (default) enables the ADF authentication servlet so that you can redirect to a configured web page when the user logs in and logs out. This option also enables ADF authorization to enforce authorization checking against security policies that you define for ADF resources. This option assumes that you will define application roles and assign explicit grants to those roles to manage access to ADF security-aware resources.
- **ADF Authentication** enables the ADF authentication servlet to require the user to log in the first time a page in the application is accessed and supports page redirect by mapping the Java EE application root "/" to the a Java EE security constraint that will trigger user authentication. Since the wizard disables ADF authorization, authorization checking is not performed, whether or not security policies exist for ADF resources. Once the user is logged in, all web pages containing ADF resources will be available to the user.
- **Remove ADF Security Configuration** disables the ADF authentication servlet and prevents ADF Security from checking policy grants without altering the existing policy store. In this case, you may require users to log in, become authenticated, and test access rights against URL security constraints using standard Java EE security. Note that running the wizard with this option disables fine-grained security against ADF resources.

Figure 47-2 Using the Configure ADF Security Wizard to Enable Security-Aware Resources



Configure ADF Security - Step 1 of 5

Enable ADF Security

Select the ADF security model you want to enable. If you just want to configure Java EE security without any ADF features, see Securing ADF Resources Using ADF Security in Fusion Web Applications for help.

Security Model:

● ADF Authentication and Authorization

Java EE security extended to support ADF authentication and authorization. This is recommended if you're building an ADF web application, including WebCenter.

○ ADF Authentication

Java EE security extended to support only ADF authentication.

○ Remove ADF Security Configuration

Remove all metadata previously generated by this wizard to enable ADF Security.

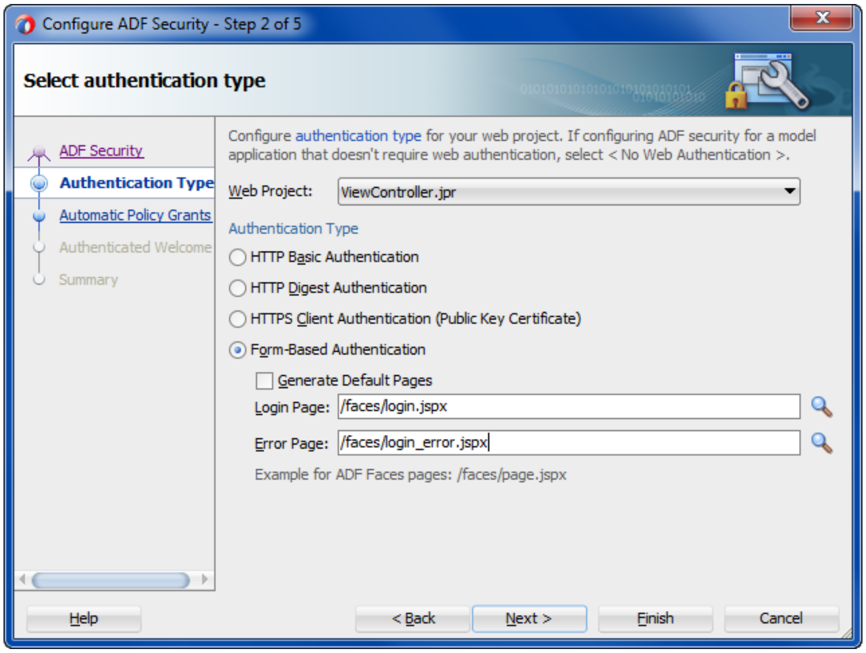
□ Enable Identity Governance Framework (IGF)

3. In the Authentication Type page, select the authentication type that you want your application to use when the user submits their login information. Click **Next**.

A known issue prevents the ADF authentication servlet from working with Basic type authentication and allows a user to access resources after logout. Use form-based authentication instead of basic authentication. For details about this issue, see What You May Need to Know About Fusion Web Application Logout and Browser Caching.

If you select **Form-based Authentication**, you can also select **Generate Default Pages** to allow the wizard to generate a default login and error page. By default the wizard generates the login and error pages at the top level of the user interface project, as shown in Figure 47-3. If you want to change the location, specify the full path relative to the user interface project.

Figure 47-3 Using the Configure ADF Security Wizard to Generate a Simple Login Page



Configure ADF Security - Step 2 of 5

Select authentication type

Configure authentication type for your web project. If configuring ADF security for a model application that doesn't require web authentication, select < No Web Authentication >.

Web Project: ViewController.jpr

Authentication Type

○ HTTP Basic Authentication

○ HTTP Digest Authentication

○ HTTPS Client Authentication (Public Key Certificate)

● Form-Based Authentication

□ Generate Default Pages

Login Page: /faces/login.jspx

Error Page: /faces/login_error.jspx

Example for ADF Faces pages: /faces/page.jspx

4. In the Automatic Policy Grants page, leave the default **No Automatic Grants** option selected. Click **Next**.

   When you select **No Automatic Grants**, you must define explicit grants that are specific to your application. The `test-all` application role provides a convenient way to run and test application resources without the restricted access that ADF authorization enforces. However, it increases the risk that your application may leave some resources unprotected.

   Alternatively, you can use the wizard to grant to the `test-all` application role. When you enable grants to the `test-all` role, you can postpone defining explicit grants to ADF resources until you are ready to refine the access policies of your application. If you decide to enable automatic grants, do not let application development progress too far and the content of the application become well-established before you replace grants to the `test-all` role with the your application's explicit grants. The explicit grant establishes the necessary privilege (for example, `view` on a page) to allow users to access these resources. For more information about the `test-all` role, see How to Use the Built-In test-all Application Role.

5. In the Authenticated Welcome page, select **Redirect Upon Successful Authentication** to direct the user to a specific web page after they log or to define a default destination page after the session is lost due to session timeout. Click **Next**.

   If you leave the **Redirect Upon Successful Authentication** option unselected and your application does not explicitly handle login redirect through the `AuthenticationService` API, then by default the user will be returned to the page from which the login was initiated. However, when the user presses Ctrl-N or Ctrl-T to open a new browser window or tab, they will receive a 403 or 404 error unless a welcome page definition appears in the application's `web.xml` file. You can use this option to specify a welcome page so the definition appears in the application's `web.xml` file.

   Note that if the web page you specify contains ADF Faces components, you must define the page in the context of `/faces/`. For example, the path for `adffaces_welcome.jspx` would appear in the **Welcome Page** field as `/faces/adffaces_welcome.jspx`.

   For details about specifying other redirect options, see How to Redirect a User After Authentication.

6. In the Summary page, review your selections and click **Finish**.

What Happens When You Enable ADF Security

After you run the Configure ADF Security wizard with the default **ADF Authentication and Authorization** option selected in the ADF Security page, you will have:

- Enabled ADF authentication to prompt the user to log in and to allow page redirects
- Enabled ADF authorization checking so that only authorized users will have access to ADF resources

The wizard updates all security-related configuration files and ensures that ADF resources are secure by default. Table 47-2 shows which files the Configure ADF Security wizard updates.

Table 47-2 Files Updated for ADF Authentication and Authorization

File	File Location	Wizard Configuration
web.xml	/public_html/WEB-INF directory relative to the user interface project And, in JDeveloper, in the user interface project under the **Web Content-WEB-INF** node	• Defines the Oracle JpsFilter filter to set up the OPSS policy provider. The filter defines settings that indicate that your servlet has special privileges. It is important that the JpsFilter be the first filter definition in the web.xml file. • Adds the Oracle adfAuthentication servlet definition to require the user to log in the first time ADF resources are accessed. Note that ADF Security does not itself perform authentication, but leverages Java EE container-managed security for this purpose. • When you select the **ADF Authentication and Authorization** option in the wizard, the web.xml maps the adfAuthentication servlet to a security constraint that will trigger user authentication dynamically. • When you select the **ADF Authentication** option in the wizard, the web.xml maps the Java EE application root "/" to the allPages security constraint that will trigger user authentication dynamically. • Sets the authentication method for the Login configuration to handle user login. • When you select the **Redirect Upon Successful Authentication** option in the wizard, the web.xml defines the success_url servlet initialization attribute (init-param element), which whitelists the redirect destination to use upon successful login and provides 1. a single, fixed fallback for the application that does not handle login redirects using the AuthenticationService API or 2. a default destination page after the session is lost due to session timeout. • Defines required security roles, including the role valid-users, which is used to trigger the security constraint that enables dynamic authentication.
adf-config.xml	/.adf/META-INF directory relative to the web application workspace And, in JDeveloper, in the Application Resources panel of the Applications window under the **Descriptors-ADF META-INF** node	• Defines the JAAS security context. • Enables the use of ADF Security security policies for authorization checking (the authorizationEnforce parameter in the <JaasSecurityContext> element is set to true). • Enables triggering a login dialog in Java SE applications, (the authenticationRequire parameter in the <JaasSecurityContext> element is set to true). This parameter is not used in Fusion web applications. Fusion web applications rely on settings in the web.xml file to enable authentication.

Table 47-2 (Cont.) Files Updated for ADF Authentication and Authorization

File	File Location	Wizard Configuration
jps-config.xml	/src/META-INF directory relative to the web application workspace And, in JDeveloper, in the Application Resources panel of the Applications window under the **Descriptors-META-INF** node	• Enables OPSS security services specifically within the JDeveloper design time. When you test data model projects using the Oracle ADF Model Tester or run unit tests for security validation this workspace-specific file must be present. Note that this file is not used in the deployed Fusion web application. For example, when you deploy to Integrated WebLogic Server, OPSS security services are enabled by the DefaultDomain/config/fmwconfig/jps-config.xml file.
weblogic.xml	/public_html/WEB-INF directory relative to the web application workspace And, in JDeveloper, in the user interface project under the **Web Content-WEB-INF** node	• Maps the valid-users security role to an implicit group called users. Oracle WebLogic Server configures all authenticated users to be members of the users group.
jazn-data.xml	./src/META-INF directory relative to the web application workspace And, in JDeveloper, in the Application Resources panel of the Applications window under the **Descriptors-META-INF** node	• Sets the default jazn.com realm name for the XML identity store that you configure for use with Integrated WebLogic Server. You will use this file to store user identities, user groups, and security policies for the ADF Security-enabled application. This file is used during development and enables support for security when running the Oracle ADF Model Tester. However, when you deploy your application, for example, to Integrated WebLogic Server, security policies will be migrated into the configured policy store in the DefaultDomain/config/fmwconfig/system-jazn-data.xml file.

Because authentication is delegated to the web container, the wizard only updates the web.xml file to enable the ADF authentication servlet to trigger authentication dynamically. It defines servlet mapping for the ADF authentication servlet and adds two Java EE security constraints, allPages and adfAuthentication, to the web.xml file, as shown in the following example.

```
<servlet>
   <servlet-name>adfAuthentication</servlet-name>
   <servlet-class>
        oracle.adf.share.security.authentication.AuthenticationServlet
   </servlet-class>
   <init-param>
      <param-name>success_url</param-name>
      <param-value>faces/welcome</param-value>
   </init-param>
   <init-param>
      <param-name>end_url</param-name>
      <param-value>faces/welcome</param-value>
   </init-param>
   <init-param>
```

```
      <param-name>disable_url_param</param-name>
      <param-value>true</param-value>
   </init-param>
   <load-on-startup>1</load-on-startup>
</servlet>
...
<servlet-mapping>
   <servlet-name>adfAuthentication</servlet-name>
   <url-pattern>/adfAuthentication</url-pattern>
</servlet-mapping>
...
<security-constraint>
   <web-resource-collection>
        <web-resource-name>allPages</web-resource-name>
        <url-pattern>/</url-pattern>
   </web-resource-collection>
   <auth-constraint>
        <role-name>valid-users</role-name>
   </auth-constraint>
</security-constraint>
<security-constraint>
   <web-resource-collection>
        <web-resource-name>adfAuthentication</web-resource-name>
        <url-pattern>/adfAuthentication</url-pattern>
   </web-resource-collection>
   <auth-constraint>
        <role-name>valid-users</role-name>
   </auth-constraint>
</security-constraint>
```

The servlet initialization attributes defined by the `init-param` elements shown in the example control redirects for successful login and logout (respectively, `success_url` and `end_url`). These `init-param` elements provide a default or fallback destination when the application does not specify the redirect target during login and logout. In this sense, they define a fixed destination whitelist on the ADF authentication servlet that prevents unvalidated redirects from occurring. The servlet initialization attribute `disable_url_param` may be optionally added to the `web.xml` servlet `init-param` list to block passing `success_url` and `end_url` on the browser URL and provides a level of protection against malicious redirect attacks that might exploit these parameters on the browser URL. This redirect protection works especially well when the application uses the `AuthenticationService` API to support login and logout redirects, which stores the redirect URL parameters as session attributes that may not be changed by the user.

Note that the protection provided by the servlet `init-param` element `disable_url_param` must be opted into and is not enabled by default. Existing applications that rely on the legacy approach of passing redirect parameters on the browser URL Request should not enable the `disable_url_param` attribute. For new applications, it is recommended that you opt in as the example shows (`disable_url_param` set to `true`), to block the usage of `success_url` and `end_url` on the browser URL.

Because the `allPages` constraint maps to the '/' URL, it protects the Java EE application root. This mapping enables the Oracle WebLogic Server web container to trigger user authentication dynamically even before ADF Security is accessed. When the user first accesses the application, it forces the container to challenge the user for the user name and password. Then when the user accesses a page protected by ADF Security, there is no longer a need to authenticate the user and no need to redirect to the ADF authentication servlet.

> **Note:**
>
> You can remove the allPages constraint from the web.xml file if you prefer to provide a login link or button to explicitly trigger login. You could also have a link or button to perform logout. For details about creating a custom component to perform login and logout, see Creating a Login Page. If you keep the constraint to allow dynamic authentication, because it covers everything under the Java EE application root, your login page may not display supporting resources at runtime, as described in How to Ensure That the Custom Login Page's Resources Are Accessible for Explicit Authentication.

Because every user of the application is required to be able to log in, the security constraint defined against the adfAuthentication resource allows all users to access this web resource. As such, the security role associated with the constraint must encompass all users. To simplify this task, the Java EE valid-users role is defined. The weblogic.xml file maps this role to an implicit users group defined by Oracle WebLogic Server. This mapping ensures that every user will have this role because Oracle WebLogic Server configures all properly authenticated users as members of the users group, as described in What You May Need to Know About the valid-users Role.

> **Note:**
>
> The adfAuthentication resource constraint provides the definition of a single standard URL pattern against the ADF authentication servlet. Your web pages can provide an explicit login or logout link that references the ADF authentication servlet URL pattern. This explicit login scenario is an alternative to generating a simple login form in the Configure ADF Security wizard and relying on ADF authentication to prompt the user to log in. For details about handling the explicit login scenario, see Creating a Login Page.

To enable authorization, the wizard updates the adf-config.xml file and sets the authorizationEnforce parameter in the <JaasSecurityContext> element to true, as shown in the following example.

```
<JaasSecurityContext

initialContextFactoryClass="oracle.adf.share.security.JAASInitialContextFactory"
  jaasProviderClass="oracle.adf.share.security.providers.jps.JpsSecurityContext"
                             authorizationEnforce="true"
                             authenticationRequire="true"/>
```

When authorization is enabled, the ADF Security Context gets the user principal from the HttpServletRequest once the user is authenticated by the container. The user submits a user name and password and that data is compared against the data in the identity store where user information is stored. If a match is found, the originator of the request (the user) is authenticated. The user principal is then stored in the ADF Security Context, where it can be accessed to obtain other security-related information (such as the group the user belongs to) in order to determine authorization rights. For details about accessing the ADF Security Context, see Getting Information from the ADF Security Context.

What Happens When You Generate a Default Form-Based Login Page

The wizard-generated login and error pages are simple HTML pages that are added to the top-level folder of your user interface project. The generated login page defines an HTML form that will submit the user's login request with the standard `j_security_check` action. This action together with form-based authentication, which is the default option for container authentication provided by the wizard, allows the web container to authenticate users from many different web application resources.

The wizard updates the `web.xml` file to specify form-based authentication and identify the location of the pages, as shown in the following example.

```
<login-config>
   <auth-method>FORM</auth-method>
   <form-login-config>
       <form-login-page>/login.html</form-login-page>
       <form-error-page>/error.html</form-error-page>
   </form-login-config>
</login-config>
```

Your application will display the wizard-generated login page from a server-side redirect in response to the unauthenticated user attempting to access a protected resource. This is known as implicit authentication because the redirect to the login page only occurs when user navigates to a page that contains an ADF security-aware resource.

Note that web applications also have a notion of public pages and allow for explicit, as well as implicit authentication. This means that users should be able to log in to the application by clicking a login link before they navigate to secured content. For information about creating and using a login link, see Creating a Login Page.

What You May Need to Know About the Configure ADF Security Wizard

The first time you run the Configure ADF Security wizard and enable authentication and authorization, you secure ADF resources at the level of the application. Additionally, you select specific project-level settings for the user interface project, including the authentication type and the authentication welcome. The wizard adds these web application settings to the `web.xml` file in the project you select. When your application contains multiple user interface projects and `web.xml` files, you can return to the wizard and configure these settings in the `web.xml` file for another user interface project that you select.

What You May Need to Know About ADF Authentication

Fusion web applications that use Java EE container-managed authentication for login and ADF authentication for logout integrate with Oracle Single Sign-On (Oracle SSO) with no special requirements. The ADF authentication servlet handles the details of logout and invalidates the user session. However, when the application uses login and logout methods provided by Servlet 3.0, Oracle SSO is not supported. Additionally, Servlet 3.0 login and logout is only compatible with application servers that fully support Java EE 6 or later.

On the first access to a page that relies on an ADF security-aware resource, when the user is not yet logged in, the application server creates a session and the `JpsFilter`, configured in the `web.xml` when you run the Configure ADF Security wizard, creates a subject containing the `anonymous` user principal and the `anonymous-role` role principal. With this role principal, the anonymous user session allows the unauthenticated user to access public web pages that are not associated with any ADF security-aware resources (including ADF bounded task flows or page definitions). Upon login, WebLogic Server does not invalidate the existing anonymous user session, and for security reasons, only creates a new session ID for the authenticated user session.

The authenticated user session behavior enforced by WebLogic Server preserves the user's ability to view public pages that had previously been visited by cloning the anonymous user session data to the authenticated user session with the new ID. While this behavior differs from the way HTTP sessions are created on WebLogic Server, there is no security risk involved in carrying anonymous user session data into the authentication user session.

> **Note:**
>
> In the Fusion web application, the session bean is initialized only when the session is established. If the session is created as `anonymous`, the session bean will not get re-initialized when the user logs in even though WebLogic Server creates a new session. This is the case when going from a public page to a secured page. However, when the user enters the application on a secured page, the session bean does get initialized upon user login.
> For this reason, Fusion web applications should not rely on session-scoped managed beans to get the current user data from the authenticated session. The correct way for the application to obtain the current user is from the ADF Security Context. For details about the API to work with the ADF Security Context, see Getting Information from the ADF Security Context.

In the case of pages associated with ADF security-aware resources, you must explicitly grant `view` permission to `anonymous-role` to make the page accessible to the anonymous user. For details about granting privileges to the anonymous user, see How to Make an ADF Resource Public.

What You May Need to Know About the Built-In test-all Role

The Configure ADF Security wizard lets you enable automatic grants to the built-in `test-all` application role for the purpose of granting view permission to all ADF security-aware resources in your application. Without a permission grant, either an automatic view grant or an explicit grant that you define, ADF Security authorization checking enforcement would prevent you from being able to run the application and access its resources. You can run the wizard with the `test-all` application role feature enabled and then gradually replace automatic view grants with explicit grants. Be aware that you must not deploy the application with grants to the `test-all` application role in place, since this feature makes all ADF resources public. If you choose to enable the built-in `test-all` application role in the wizard, see How to Remove the test-all Role from the Application Policy Store, before deploying your application.

What You May Need to Know About the valid-users Role

The valid-users role is a Java EE security role defined by ADF Security to ensure that all users will access the adfAuthentication servlet web resource defined in the web.xml file. The Configure ADF Security wizard updates the weblogic.xml file to map this ADF Security role to the users principal, as shown in the following example. This mapping ensures that every user will have this role, because Oracle WebLogic Server configures all properly authenticated users as members of the users group.

```
<security-role-assignment>
   <role-name>valid-users</role-name>
   <principal-name>users</principal-name>
</security-role-assignment>
```

At runtime, the users principal is added automatically to a successfully authenticated subject by OPSS. From a security perspective, the valid-users role supports ADF authentication only in the case where you need to control access to web resources using security constraints alone. The end result of this mapping relies entirely on Java EE security and does not involve JAAS Permissions.

Creating Application Roles

To grant permissions to ADF objects, you need to create application roles that are mapped to enterprise groups. When you define roles, you have two options. You can define them either as application roles or as enterprise roles. Application roles are local to an application and it can contain only users and roles defined in the application, whereas enterprise roles are available to all applications deployed in the domain.

You create application roles to represent the policy requirements of the application and to define groups of users with the same view permission rights. The application roles that you create in the application policy store are specific to your application. For example, in the context of the work flow, there may be application roles such as Application Customer Role and Application Employee Role, defined in the SummitADF_TaskFlows workspace of the Summit ADF sample applications.

At runtime, the access rights are conferred on the user through the application role for which the user is defined as a member. Thus, before you can define security policies, the policy store must contain the application roles that you intend to issue grants to. This can be an application role that you define (such as Application Customer Role) or it can be one of the two built-in application roles defined by OPSS: authenticated-role or anonymous-role. JDeveloper provides the built-in application roles to let you make ADF resources public, as described in How to Make an ADF Resource Public.

After you create the application role, you will:

- Grant permissions to the application roles, as described in Defining ADF Security Policies.
- Associate test users with each application role, as described in Creating Test Users.

Best Practice:

The ADF Security framework enforces a role-based access control mechanism with permissions granted either to application roles or to individual users. Although you may only need to test security and therefore might not need to create groups of users, you should still create application roles (with at least one user member). Later when you define security polices on the ADF resources, the overview editor for the application policy store will allow you to select an application role for the grant.

How to Create Application Roles

JDeveloper lets you add application roles to the policy store of the `jazn-data.xml` file, which appears in the **Descriptors/META-INF** node of the Application Resources panel.

Note:

When you create application roles, be sure to add the new application roles to the policy store, not the identity store. Roles that you add to the identity store define enterprise security roles and provide a way to conveniently group users in the identity store. For more details about enterprise roles, see What You May Need to Know About Enterprise Roles and Application Roles.

To create application roles in the policy store of the `jazn-data.xml` file, you use the Application Roles page of the overview editor for the `jazn-data.xml` file. This editor lets you view the relationship between identity store members and the application roles you create.

Before you begin:

It may be helpful to have an understanding of application roles. For more information, see Creating Application Roles.

To create application roles:

1. In the main menu, choose **Application** and then **Secure > Application Roles**.
2. In the Application Roles page of the `jazn-data.xml` overview editor, select the policy store for your application from the **Security Policy** dropdown list.

   The policy store that JDeveloper creates in the `jazn-data.xml` file is automatically based on the name of your application.
3. In the **Roles** list, click the **New** icon.
4. In the **Name** field, enter the name of the role and click any other field to add the application role to the policy store.
5. If you have already set up test users in the identity store, you can map users and roles, as described in How to Associate Test Users with Application Roles.

What Happens When You Create Application Roles

When you add an application role to the policy store, JDeveloper updates the `jazn-data.xml` file located in the `src/META-INF` directory relative to the application workspace. Application roles are defined in `<app-role>` elements under `<policy-store>`, as shown in the following example. Because the policy store `<application>` element names the application, at runtime all application roles that you create will be visible to your application only. Other web applications may define a policy store with their own set of application roles.

```
<policy-store>
     <applications>
         <application>
            <name>SummitADFTaskFlows</name>
            <app-roles>
               <app-role>
                  <name>Application Employee Role</name>
                  <display-name>Application Employee Role</display-name>
                  <class>oracle.security.jps.service.policystore.
                                                      ApplicationRole</class>
               </app-role>
               ...
            </app-roles>
            <jazn-policy>
              ...
            </jazn-policy>
         </application>
    </applictions>
</policy-store>
```

What You May Need to Know About Enterprise Roles and Application Roles

An **enterprise role** is a role that is maintained in the domain identity store (as opposed to an application identity store). Enterprise roles are available to every application deployed in the domain and are therefore also called external roles.

An **application role** is a role used by a Fusion web application. It is specific to the application, defined by the application policy, and not necessarily known to the Java EE container. Application roles are scoped in the sense that they can contain only users and roles defined in the application. Application roles must be mapped to enterprise roles.

You use the overview editor for the `jazn-data.xml` file to create enterprise roles to group users that you add to the identity store. You can use this mechanism to assign entire groups of users to application roles that you have defined for the purpose of conferring access rights defined by ADF security policies, as described in How to Associate Test Users with Application Roles.

However, Integrated WebLogic Server does not require you to create enterprise roles to run the application within JDeveloper. For the purpose of testing the application, it may be sufficient to create a few test users and assign them directly to application roles. When you run the application in JDeveloper, the users and any enterprise roles you defined will be created in the default security provider (which is embedded LDAP for Integrated WebLogic Server).

Typically, when you deploy the application for staging, you will migrate only the policy store to the target server. You can configure JDeveloper deployment options so that the identity store, including test users and enterprise roles, is not migrated, as described in How to Configure, Deploy, and Run a Secure Application in JDeveloper.

After you deploy the secure application, Oracle Fusion Middleware will merge your application's policy store with the policies of the domain-level policy store. To complete this task, the administrator for the Oracle WebLogic Server will eventually map the application roles of your policy store to the existing domain-level enterprise roles. This application role mapping at the domain level allows enterprise users to access application resources according to the ADF security policies you have defined. The domain-level application role mapping by the administrator also allows you to develop the ADF security policies of your application without requiring any knowledge of the identity store in the production environment.

Defining ADF Security Policies

The ADF security implementation consists of two main concepts; authentication and authorization. Authentication ensures the identity of the user based on the identity store and authorization ensures that the user only has access to resources they have been granted access to in the policy store.

Authorization relies on a policy store that is accessed at runtime and that contains permissions that grant privileges to execute predefined actions, like `view`, on a specified object. Initially, after you run the Configure ADF Security wizard, the policy store defines no grants. And, because the default wizard option **ADF Authentication and Authorization** enables authorization checking, the web pages of your application that rely on the ADF security-aware resources will be inaccessible to users. You must use JDeveloper to define explicit grants for the resources that you want to permit users to access.

> **Best Practice:**
>
> When you run the Configure ADF Security wizard with the default option **ADF Authentication and Authorization** selected, you will lock down the web pages of your application. This affords the most protection to the Fusion web application possible since you will define explicit grants to allow users to access only the pages you intend. For a discussion of this guideline and others, see Best Practices for Working with ADF Security.

Before you can define security policies, the policy store for your application must contain the application roles that you intend to issue grants to. This can be an application role that you define (such as `Application Customer Role`) or it can be one of the two built-in application roles defined by OPSS: `authenticated-role` or `anonymous-role`. You use application roles to classify users, so that each member of the same role possesses the same access rights. As such, the security policy names the application role as the principal of the grant, rather than specific users. For details about defining application roles, see Creating Application Roles.

For the user interface project, you use the overview editor for security policies to secure ADF resources, including ADF task flows and ADF page definitions. You open the editor on the `jazn-data.xml` file by double-clicking the `jazn-data.xml` file (located in the Application Resources panel) or by choosing **Secure > Resource Grants** from the **Application** menu in the main menu.

Note that when you open the `jazn-data.xml` file, the overview editor provides additional editor pages that you use to create test users, enterprise roles, and application roles.

For the data model project, you do not secure entity objects or their attributes using the overview editor for security policies. Instead, you set metadata directly on these objects to manage whether or not the databound UI component displays the data. For details about granting permissions for row-level security, see How to Define Policies for Data.

How to Make an ADF Resource Public

It is a common requirement that some web pages be available to all users, regardless of their specific access privileges. For example, the home page should be seen by all visitors to the site, while a corporate site should be available only to those who have identified themselves through authentication.

In both cases, the page may be considered public, because the ability to view the page is not defined by the users' specific permissions. Rather, the difference is whether the user is anonymous or a known identity.

In the ADF security model, you differentiate between the absence of security and public access to content by granting access privileges to the `anonymous-role` principal. The anonymous role encompasses both known and anonymous users, thus permission granted to `anonymous-role` allows access to a resource by unauthenticated users, for example, guest users. To provide access to authenticated users only, the policy must be defined for the `authenticated-role` principal.

> **Note:**
>
> For details about creating a public home page which contains links to other pages in the application, see How to Create a Public Welcome Page.

Before you begin:

It may be helpful to have an understanding of ADF security policies. For more information, see Defining ADF Security Policies.

You will need to complete these tasks:

1. Create bounded task flows, as described in Creating a Task Flow.
2. Create web pages with an ADF page definition file, as described in Working with Page Definition Files.
3. Run the Configure ADF Security wizard, as described in Enabling ADF Security.
4. Create application roles, as described in Creating Application Roles.

To grant public access to ADF security-aware resources:

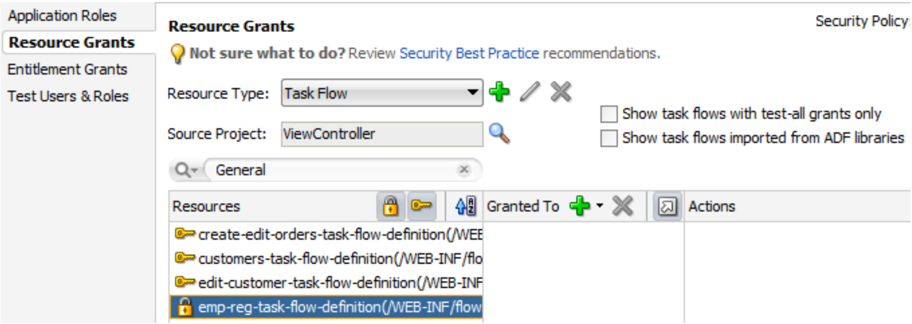
1. In the main menu, choose **Application** and then **Secure > Resource Grants**.

2. In the Resource Grants page of the overview editor for security policies, select one of the following resources from the **Resource Type** dropdown list:
   - **Task Flow** when you want to make a bounded task flow public. The application displays the web pages under the permission you define for the task flow itself. Thus, all constituent web pages of the bounded task flow will become public.
   - **Web Page** when you want to make individual web pages public. Typically, these pages are defined by an unbounded task flow and are top-level pages in the application, such as a home page.
3. In the **Resources** column, select the ADF resource for which you want to grant access rights.

   The resource you select should display the lock icon in the first column next to the resource name. The lock icon indicates that the resource has no security policy defined and therefore is "locked"—which means it remains inaccessible to users until you define a grant. For example, in Figure 47-4, the ADF resource `customer-registration-task-flow` (a bounded task flow) shows the lock icon since no grant has been made.

   > **Tip:**
   >
   > Click the key toggle icon in the header for the overview editor's first column to hide or show resources that already have grants and display only the resources without grants. The key icon indicates that the resource has a grant that will make the resource accessible to users with sufficient access rights.

Figure 47-4 Selecting an ADF Security-Aware Resource in the Overview Editor



Resource Type: Task Flow

Source Project: ViewController

□ Show task flows with test-all grants only

□ Show task flows imported from ADF libraries

General

create-edit-orders-task-flow-definition(/WEE
customers-task-flow-definition(/WEB-INF/flo
edit-customer-task-flow-definition(/WEB-INF
emp-reg-task-flow-definition(/WEB-INF/flow

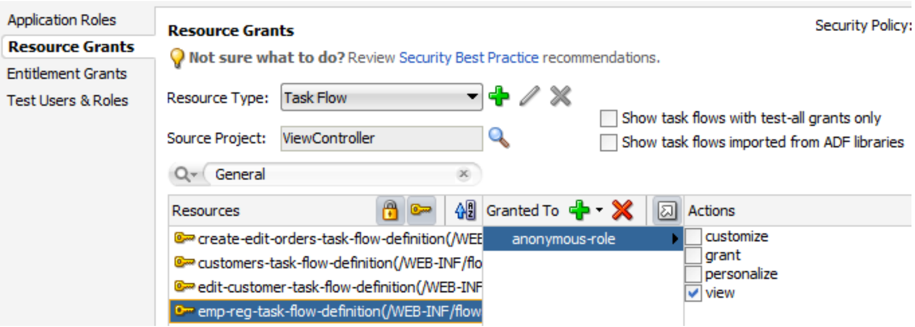
4. In the **Granted to** column, click the **Add Grantee** icon and choose **Add Application Role**.
5. In the Select Application Roles dialog, select one of these built-in application roles:
   - **anonymous-role** means the resource will be accessible to anyone who visits the site. A grant to this role is necessary if you want to make a web page associated with an ADF security-aware resource accessible before a user logs in. For example, you would grant to `anonymous-role` for a task flow that manages customer registration.

- **authenticated-role** means the resource will be accessible only to authenticated users (ones who visit the site and log in). For example, you would grant to `authenticated-role` for an employee registration task flow.

6. In the Select Application Roles dialog, click **OK**.
7. In the Resource Grants page of the overview editor, in the **Actions** column, leave the **view** action selected.

   By default, the overview editor shows **view** selected, as shown in Figure 47-5. The view action is the only action currently supported for Fusion web applications.

Figure 47-5 Granting to anonymous-role in the Overview Editor



Application Roles
Resource Grants
Entitlement Grants
Test Users & Roles

Resource Grants
Security Policy:
Not sure what to do? Review Security Best Practice recommendations.

Resource Type: Task Flow
Source Project: ViewController
□ Show task flows with test-all grants only
□ Show task flows imported from ADF libraries
General

Resources
create-edit-orders-task-flow-definition(/WEE
customers-task-flow-definition(/WEB-INF/flo
edit-customer-task-flow-definition(/WEB-INF
emp-reg-task-flow-definition(/WEB-INF/flow

Granted To
anonymous-role

Actions
□ customize
□ grant
□ personalize
☑ view

What Happens When You Make an ADF Resource Public

When you define a security policy, the overview editor for security policies updates the `jazn-data.xml` file located in the `/src/META-INF` node relative to the web application workspace.

The overview editor writes the policy information to the `<policy-store>` section of the file. The security policy, or grant, contains both a grantee and one or more permissions. The grantee is the application role that the policy is being defined for—in this case, the anonymous role. Each permission defines the resource being secured and the action that can be performed against that resource.

The following example shows a security policy in the `jazn-data.xml` file that makes a customer registration task flow public. The grant to `anonymous-role` contains a single view permission for a bounded task flow, `customer-registration-task-flow`. With this grant, all users will be able to enter the customer registration task flow and complete the customer registration process. Additional grants to the anonymous role may be made and will appear in the `<permissions>` section of the anonymous role grant.

```
<policy-store>
   ...
   <jazn-policy>
      <grant>
         <grantee>
            <principals>
               <principal>
                  <class>oracle.security.jps.internal.core.
                                      principals.JpsAnonymousRoleImpl</class>
                  <name>anonymous-role</name>
               </principal>
            </principals>
         </grantee>
         <permissions>
            <permission>
                  <class>oracle.adf.controller.security.TaskFlowPermission</
class>
                  <name>/WEB-INF/customer-registration-task-flow.xml#
                                         customer-registration-task-flow</name>
                  <actions>view</actions>
            </permission>
            ...
         </permissions>
      ...
      </grant>
      ...
   </jazn-policy>
</policy-store>
```

What Happens at Runtime: How the Built-in Roles Are Used

The `anonymous-role` and `authenticated-role` names are special roles defined by Oracle Platform Security Services (OPSS).

When you run the Configure ADF Security wizard, the wizard configures the `JpsFilter` definition in the `web.xml` file to enable support for the anonymous role. The enabled anonymous role allows ADF Security to support browsing of the site by anonymous users—those users who have not yet logged in. In contrast, the authenticated role is not declared and is always recognized by default. ADF Security supports both of these roles.

When an end user first accesses an ADF security-aware resource, the system creates a subject and populates it with the anonymous role principal. As long as the ADF security-aware resource being accessed has the view grant to anonymous role, the user is permitted access. If the anonymous role is not a grantee of the ADF resource, the user is prompted to log in. After logging in, the authenticated role is added to the subject. The wizard also adds the `JpsFilter` definition to the `web.xml` file, where `remove.anonymous.role` set to `false` ensures that the anonymous role principal is available even after the user logs in. With the authenticated role principal, the user may access resources that have an explicit grant to the authenticated role.

How to Define Policies for ADF Bounded Task Flows

You define the access policy for an ADF bounded task flow by creating permission grants in the Resource Grants page of the overview editor for security policies. The grants you create will appear as metadata in the policy store section of the `jazn-data.xml` file. This metadata defines a permission target (in this case, the bounded task flow definition name) for which you have issued grants to authorize the members of a specific application role.

Best Practice:

Do not create permission grants for the individual web pages of a bounded task flow. When the user accesses the bounded task flow, security for all pages will be managed by the permissions you grant to the task flow. And, because the contained web pages (with associated page definitions) will be inaccessible by default, ADF Security prevents users from directly accessing the pages of the task flow. This supports a well-defined security model for task flows that enforces a single entry point for all users. For further information about implementing security policies, see Best Practices for Working with ADF Security.

You can sort the task flows in the overview editor by clicking the toggle buttons in the **Task Flow** header, as described in Table 47-3.

Table 47-3 Resource Grant Toggle Buttons for Bounded Task Flows

Button	Toggle Action	Description
🔒	Shows/hides bounded task flows with no grants	Represents a bounded task flow with no permission grants defined. The web pages that the task flow calls will not be accessible to any user.
🔑	Shows/hides bounded task flows with grants	Represents a bounded task flow with one or more permission grants defined. The web pages that the task flow calls will be accessible to users who are members of the application role that received the grant.

The list of available actions displayed by the overview editor is defined by the task flow permission class (`oracle.adf.controller.security.TaskFlowPermission`). The permission class maps these actions to the operations supported by the task flow. Table 47-4 shows the actions displayed by JDeveloper for ADF bounded task flows.

Note that the `view` action is the only action currently supported for Fusion web applications. Do not select `customize`, `grant`, or `personalize` actions—they are reserved for future use in task flow security.

Table 47-4 Secured Actions of ADF Bounded Task Flows

Grantable Action	Effect on the User Interface
`view`	Controls who can read and execute a bounded task flow in a Fusion web application. This is the only operation that the task flow supports.
`customize`	Reserved for future use. This action is not checked at runtime.
`grant`	Reserved for future use. This action is not checked at runtime.
`personalize`	Reserved for future use. This action is not checked at runtime.

To define a grant for the task flow security policy, use the Resource Grants page of the overview editor for the `jazn-data.xml` file.

Before you begin:

It may be helpful to have an understanding of ADF security policies. See Defining ADF Security Policies.

You will need to complete these tasks:

1. Create bounded task flows, as described in Creating a Task Flow.

> **Best Practice:**
>
> If you are creating bounded task flows in separate UI projects of the same application, you will want to assign unique task flow definition names. This is necessary because a grant's task flow definition name is scoped in the `jazn-data.xml` policy store by path (for example, `/WEB-INF/`*`mytaskflow`*`-definition.xml#`*`mytaskflow`*`-definition`). Therefore creating bound task flows with unique definition names is the only way to impose project-level scoping of the grants.

2. Run the Configure ADF Security wizard, as described in Enabling ADF Security.
3. Create application roles, as described in Creating Application Roles.

To define a permission grant on an ADF bounded task flow:

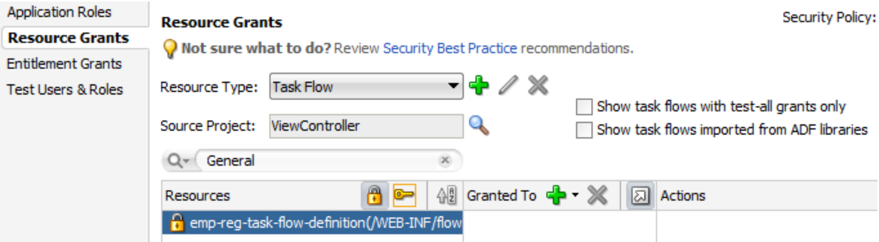
1. In the main menu, choose **Application** and then **Secure > Resource Grants**.
2. In the Resource Grants page of the overview editor for security policies, select **Task Flow** from the **Resource Type** dropdown list.

   The overview editor displays all the task flows that your application defines. Task flows are defined by task flow definition files (`.xml`) that appear in the Web Content/Page Flows node of the user interface project.
3. In the **Resources** column, select the task flow for which you want to grant access rights.

   The first time you make a grant to a bounded task flow, the first column should display the **Resources without any grants** icon (represented by a "lock") next to the task flow name. The overview editor displays the lock icon to indicate that a resource has no security policy defined and therefore is "locked"—which means it remains inaccessible to users until you define a grant.

   > **Tip:**
   >
   > Click the **Resources with grants** icon (represented by a "key") in the header for the **Resources** column to hide all task flows that already have grants. This will display only task flows without grants, as shown in Figure 47-6. Additionally, you can type a partial task flow name in the search field to display only the task flows with character-matching names.

Figure 47-6 Hiding Task Flows with Grants in the Overview Editor



Resource Type: Task Flow

Source Project: ViewController

□ Show task flows with test-all grants only

□ Show task flows imported from ADF libraries

4. In the **Granted to** column, click the **Add Grantee** icon and select **Add Application Role**.

5. In the Select Application Roles dialog, select the application role that you want to make a grantee of the permission.

   The Select Application Roles dialog displays application roles from the `jazn-data.xml` file. It also displays the built-in OPSS application roles, `anonymous-role` and `authenticated-role`, as described in What Happens at Runtime: How the Built-in Roles Are Used.

   If you do not see application roles that are specific to your application, create the role, as described in Creating Application Roles.

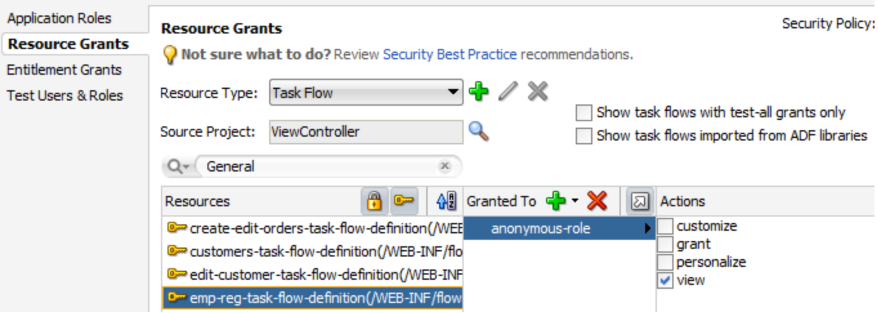
6. In the Select Application Roles dialog, click **OK**.

7. In the Resource Grants page of the overview editor, in the **Actions** column, leave the **view** action selected.

   By default, the overview editor shows the **view** action selected, as shown in Figure 47-7. The view action is the only action currently supported for Fusion web applications. Do not select **customize**, **grant**, or **personalize** actions—they are reserved for future use and will not be checked by ADF Security at runtime.

   The `TaskFlowPermission` class defines task flow—specific actions that it maps to the task flow's operations, as described in Table 47-4.

Figure 47-7 Granting to an Application Role for a Bounded Task Flow Definition in the Overview Editor



Application Roles
Resource Grants
Entitlement Grants
Test Users & Roles
Resource Grants
Security Policy:
Not sure what to do? Review Security Best Practice recommendations.

Resource Type: Task Flow

Source Project: ViewController

□ Show task flows with test-all grants only

□ Show task flows imported from ADF libraries

General
Resources
Granted To
Actions
create-edit-orders-task-flow-definition(/WEE
customers-task-flow-definition(/WEB-INF/flo
edit-customer-task-flow-definition(/WEB-INF
emp-reg-task-flow-definition(/WEB-INF/flow
anonymous-role

□ customize
□ grant
□ personalize
☑ view

8. You can repeat these steps to make additional grants as desired.

   The same task flow definition can have multiple grants made for different application roles. The grants appear in the policy store definition of the `jazn-data.xml` file, as described in What Happens When You Define the Security Policy.

How to Define Policies for Web Pages That Reference a Page Definition

You define the access policy for an ADF page definition by creating permission grants in the Resource Grants page of the overview editor for security policies. The grants you create will appear as metadata in the policy store section of the `jazn-data.xml` file. This metadata defines a permission target (in this case, the page definition name) for which you have issued grants to authorize the members of a specific application role.

> **Best Practice:**
>
> Create permission grants for the individual web page only when the page is not a constituent of a bounded task flow. Page-level security is checked for pages that have an associated page definition binding file *only* if the page is directly accessed or if it is accessed in an *unbounded* task flow. For further information about implementing security policies, see Best Practices for Working with ADF Security.

You can sort the web page definition resources in the overview editor by clicking the toggle buttons in the **Resources** header, as described in Table 47-5.

Table 47-5 Resource Grant Toggle Buttons for Web Page Definitions

Button	Toggle Action	Description
🔒	Shows/hides top-level pages with no grants	Represents a page definition with no permission grants defined for a web page that is contained in an unbounded task flow. The web page will not be accessible to any user.
🔑	Shows/hides top-level pages with grants	Represents a page definition with one or more permission grants defined for a web page that is contained in an unbounded task flow. The web page will be accessible to users who are members of the application role that received the grant.
📄	Shows/hides pages included in a bounded task flow	Represents a page definition associated with a web page that also is contained in a bounded task flow. Do not grant to these web page definitions. Instead, define a security policy for the bounded task flow.

Table 47-5 (Cont.) Resource Grant Toggle Buttons for Web Page Definitions

Button	Toggle Action	Description
🔑	Shows/hides unsecurable pages (with no page definition)	Represents a web page with no page definition defined that is contained in an unbounded task flow. (Pages like this that are contained by a bounded task flow are secured by the bounded task flow's permission.) The web page will be accessible to all users since it is not secured by an associated ADF security-aware resource. Optionally, you can secure the page by adding an empty page definition file, as described in What You May Need to Know About Defining Policies for Pages with No ADF Bindings.

The list of available actions displayed by the overview editor is defined by the region permission class (`oracle.adf.share.security.authorization.RegionPermission`). The permission class maps these actions to the operations supported by the ADF page definition for the web page. Table 47-6 shows the actions displayed by JDeveloper for ADF page definitions.

Note that the `view` action is the only action currently supported for Fusion web applications.

Do not select `customize`, `grant`, or `personalize` actions—they are implemented for page definition security only in WebCenter Portal: Framework applications.

Table 47-6 Securable Actions of ADF Page Definitions

Grantable Action	Effect on the User Interface
`view`	Controls who can view the page. This is the only operation that the page definition supports. All other operations support Oracle WebCenter Portal: Framework.
`customize`	Controls who can make implicit changes (such as minimize/restore, delete, or move) to a WebCenter Portal customizable component (in a Panel Customizable or Show Detail Frame) contained in a page of a custom application (one enabled to use Oracle WebCenter Portal's Composer) or a WebCenter Portal application. For details, see Introduction to WebCenter Portal Assets in *Developing for Oracle WebCenter Portal*.
`grant`	Confers the rights specified by all WebCenter Portal-specific actions combined; it is equivalent to granting all other actions. It also controls who can make grants to other users and who can change security settings on the page using Oracle WebCenter Portal's Composer. For details, see Introduction to WebCenter Portal Assets in *Developing for Oracle WebCenter Portal*.
`personalize`	Controls who can make implicit changes (such as minimize/restore, delete, or move) to a WebCenter Portal customizable component (in a Panel Customizable or Show Detail Frame) contained in a page of a custom application (one enabled to use Oracle WebCenter Portal's Composer) or a WebCenter Portal application. For details, see Introduction to WebCenter Portal Assets in *Developing for Oracle WebCenter Portal*.

To define a grant for the page definition security policy, use the Resource Grants page of the overview editor for security policies.

Before you begin:

It may be helpful to have an understanding of ADF security policies. See Defining ADF Security Policies.

You will need to complete these tasks:

1. Create the top-level web pages with an ADF page definition file, as described in Working with Page Definition Files.

> **Best Practice:**
>
> If you are creating top-level web pages in separate UI projects of the same application, you will want to assign unique page file names. This is necessary because a grant's page definition name is scoped in the `jazn-data.xml` policy store by package (for example, `view.pageDefs.`*`mytoppage`*`PageDef`). Therefore creating top-level pages with unique file names is the only way to impose project-level scoping of the grants.

2. Run the Configure ADF Security wizard, as described in Enabling ADF Security.
3. Create application roles, as described in Creating Application Roles.

To define a permission grant on an ADF page definition:

1. In the main menu, choose **Application** and then **Secure > Resource Grants**.
2. In the Resource Grants page of the overview editor for security policies, select **Web Page** from the **Resource Type** dropdown list.

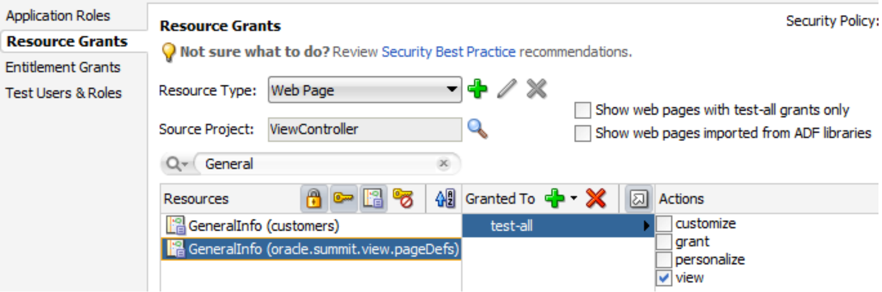
   The Resource Grants page of the overview editor displays all web pages, including those that have an associated ADF page definition. This includes any web page that uses ADF bindings or any web page for which you have created an empty page definition. Page definitions are defined by `PageDef.xml` files that appear in the Application Sources node of the user interface project.
3. In the **Resources** column, select the page definition for which you want to grant access rights.

   The first time you make a grant to a page definition, the first column should display the **Resource without any grants** icon (represented by the "lock" icon) next to the page definition name. The editor displays the lock icon to indicate that a resource has no security policy defined and therefore is "locked"—which means it remains inaccessible to users until you define a grant. For example, the page definition `account_updateUserInfo` shown in Figure 47-8 displays the lock icon since no grant has been made. Other page definitions in Figure 47-8 show the **Page included in bounded task flow** icon because they are not top-level pages and thus are securable by the containing bounded task flow.

   Do not create grants for individual web page definitions that display the **Page included in bounded task flow** icon. Security policies for the associated web pages are secured by their bounded task flow. For example, in Figure 47-8, the page definition associated with the `account_addressDetails.jsff` region will be secured by the containing bounded task flow.

> **Tip:**
>
> You can type a partial page definition name in the search field to display only the page definitions with character-matching names. For example, a search on the word `general` would display only the page definitions that begin with the word `general`, as shown in Figure 47-8.

Figure 47-8 Matching Page Definitions by Name in the Overview Editor


Application Roles
Resource Grants
Entitlement Grants
Test Users & Roles
Resource Grants
Security Policy:
Not sure what to do? Review Security Best Practice recommendations.
Show web pages with test-all grants only
Show web pages imported from ADF libraries
Resources
Granted To
Actions

Resource Type: Web Page

Source Project: ViewController

Search: General

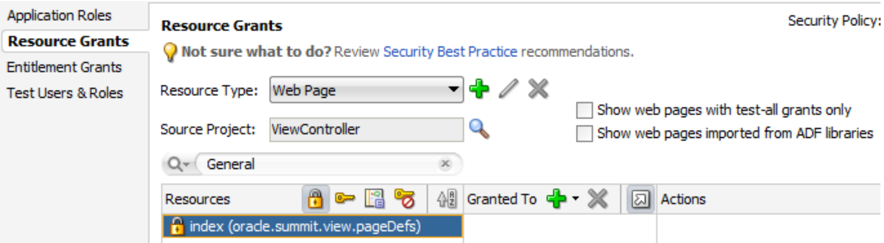
☐ Show web pages with test-all grants only

☐ Show web pages imported from ADF libraries

Resources	Granted To	Actions
GeneralInfo (customers)	test-all	☐ customize ☐ grant ☐ personalize ☑ view
GeneralInfo (oracle.summit.view.pageDefs)		

> **Tip:**
>
> You can click the **Resources with grants** icon (represented by the "key" icon) to hide all page definitions that already have grants. Confirm that the **Show pages included in a bounded task flow** toggle button in the header for the **Resources** column is toggled off to hide all page definitions that are included in a bounded task flow (by default, it is set to hide these pages). This will display top-level pages that have no grants (and unsecurable pages, if any), as shown in Figure 47-9.

Figure 47-9 Hiding Web Pages with Grants in the Overview Editor


Application Roles
Resource Grants
Entitlement Grants
Test Users & Roles
Resource Grants
Security Policy:
Not sure what to do? Review Security Best Practice recommendations.
Show web pages with test-all grants only
Show web pages imported from ADF libraries
Resources
Granted To
Actions

Resource Type: Web Page

Source Project: ViewController

Search: General

☐ Show web pages with test-all grants only

☐ Show web pages imported from ADF libraries

Resources	Granted To	Actions
index (oracle.summit.view.pageDefs)		

4. In the **Granted to** column, click the **Add Grantee** icon and select **Add Application Role**.
5. In the Select Application Roles dialog, select the application role that you want to make a grantee of the permission.

   The Select Application Roles dialog displays application roles from the `jazn-data.xml` file. It also displays the built-in OPSS application roles, **anonymous-role** and **authenticated-role**, as described in What Happens at Runtime: How the Built-in Roles Are Used.

If you do not see application roles that are specific to your application, create the role, as described in Creating Application Roles.

6. In the Select Application Roles dialog, click **OK**.

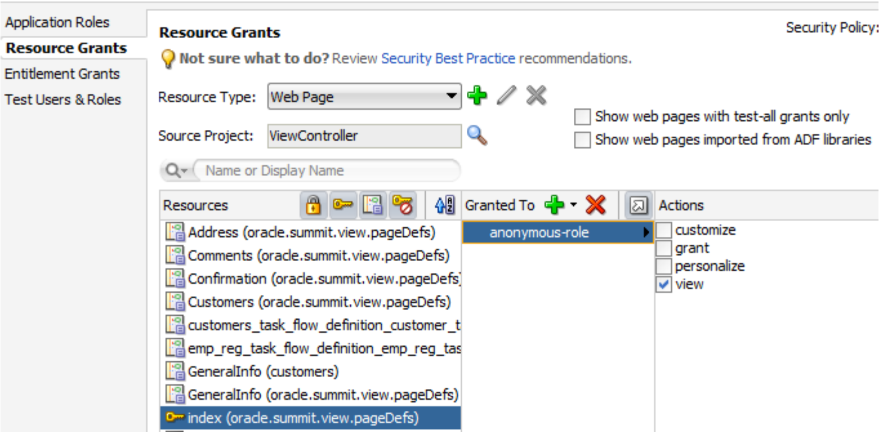
7. In the Resource Grants page of the overview editor, in the **Actions** column, leave the **view** action selected.

   By default, the overview editor shows **view** selected, as shown in Figure 47-10. The view action is the only action currently supported for Fusion web applications.

   The actions **customize**, **grant**, or **personalize** are implemented for use in WebCenter Portal: Framework applications or custom applications that are enabled to use Oracle WebCenter Portal's Composer.

   The `RegionPermission` class defines page definition—specific actions that it maps to the page's operations, as described in Table 47-6.

Figure 47-10 Granting to an Application Role for an ADF Page Definition in the Overview Editor



Application Roles
Resource Grants
Entitlement Grants
Test Users & Roles

Resource Grants
Security Policy:
Not sure what to do? Review Security Best Practice recommendations.

Resource Type: Web Page
Source Project: ViewController
Name or Display Name
□ Show web pages with test-all grants only
□ Show web pages imported from ADF libraries

Resources
Address (oracle.summit.view.pageDefs)
Comments (oracle.summit.view.pageDefs)
Confirmation (oracle.summit.view.pageDefs
Customers (oracle.summit.view.pageDefs)
customers_task_flow_definition_customer_t
emp_reg_task_flow_definition_emp_reg_tas
GeneralInfo (customers)
GeneralInfo (oracle.summit.view.pageDefs)
index (oracle.summit.view.pageDefs)

Granted To
anonymous-role

Actions
□ customize
□ grant
□ personalize
☑ view

8. You can repeat these steps to make additional grants as desired.

   The same page definition can have multiple grants made for different application roles. The grants appear in the policy store definition of the `jazn-data.xml` file, as described in What Happens When You Define the Security Policy.

How to Define Policies to Control User Access to ADF Methods

ADF methods that your application defines may be dropped into the user interface as command components. By default, users who have access to the page that displays the command component for the method will also have rights to execute the method. When you want to create additional security to restrict access to the method operation, you must create a resource grant and test the permission at the level of the user interface. ADF Security does not perform authorization checking for ADF methods; you must enable authorization checking in your application. Based on a resource permission you have granted to the user for the ADF method, the user interface will either enable or disable the command component.

To control user access to a method that your page displays as a command component, you complete these steps:

1. Create a resource grant for a custom resource type and resource.

2. Enforce the permission grant in the user interface.

Creating a Resource Grant to Control Access to ADF Methods

The resource type that you create will be set to the matcher class `oracle.security.jps.ResourcePermission`, which will allow you to grant permission to parts of the application that are not protected by ADF Security. For example, your page may display a button that lets users cancel a product shipment to customers. However, only members of a specific application role may be allowed to cancel a shipment. To enforce this rule, you can create a resource permission that may be checked declaratively in the user interface at runtime to enable or disable the button.

To create a resource permission for user interface components that you want to protect, use the overview editor for security policies.

Before you begin:

It may be helpful to have an understanding of how ADF Security handles ADF methods. For more information, see How to Define Policies to Control User Access to ADF Methods.

You will need to complete these tasks:

1. Create the command component that executes the method that you want secure, as described in Creating Command Components to Execute Methods.
2. Run the Configure ADF Security wizard, as described in Enabling ADF Security.
3. Create application roles, as described in Creating Application Roles.

To grant a resource permission on a custom resource type:

1. In the main menu, choose **Application** and then **Secure > Resource Grants**.
2. In the Resource Grants page of the overview editor for security policies, next to the **Resource Type** dropdown list, click the **New Resource Type** button.
3. In the Create Resource Type dialog, enter the name of the resource type, a display name to display when granting resource permissions in JDeveloper, and a description.

   Enter a resource type and display name that is appropriate for the user interface resource that needs to be protected. For example, if you want to protect a method button that cancels a product shipment, you might enter the resource type `CancelShipment` and display name `Cancel Shipment`.
4. Next to the **Actions** list, click the **Add Action** button and enter the name of the action that you want to protect. Click **OK**.

   The action name can be any name that you want to associate with the custom resource type. For example, if you want to protect a button that invokes a method, you might enter the action name `invoke`. You can add multiple actions to the list when you need to support granting instances of the resource permission to separate application roles.
5. In the Resource Grants page of the overview editor for security policies, in the **Resources** column, click the **Add Resource** icon.

   The first time you make a grant to a custom resource type, no resource will be defined. You must create a custom resource for the policy that will be used to check user permissions in the user interface at runtime.

6. In the Create Resource dialog, enter the name of the resource and a display name to display when granting resource permissions in JDeveloper, and then click **OK**.

   Enter a resource name and a display name that describes the user interface resource that you want to protect. For example, if you want to protect a method button that cancels a product shipment, you might enter the resource name `CancelShipmentButton` and display name `Cancel Shipment Button`.

7. In the **Granted to Roles** column, click the **Add Grantee** icon and select **Add Application Role**.

8. In the Select Application Roles dialog, select the application role that you want to make a grantee of the permission.

   The Select Application Roles dialog displays application roles from the `jazn-data.xml` file. It also displays the built-in OPSS application roles, **anonymous-role** and **authenticated-role**, as described in What Happens at Runtime: How the Built-in Roles Are Used.

   If you do not see application roles that are specific to your application, create the role, as described in Creating Application Roles.

9. In the Select Application Roles dialog, click **OK**.

10. In the Resource Grants page of the overview editor, in the **Actions** column, select the desired action.

    By default, the overview editor shows all actions of the custom resource type unselected. The available actions are defined by the resource type.

11. You can repeat these steps to make additional grants as desired.

    The same ADF method resource can have multiple grants made for different application roles. The grants appear in the policy store definition of the `jazn-data.xml` file, as described in What Happens When You Define the Security Policy.

Enforcing the Resource Grant in the User Interface

You use the Expression Builder dialog that you display for the UI component display property to define an EL expression that checks the user's access rights to a previously defined custom resource type. When you run the application, the component will appear either enabled or disabled based on the outcome of the EL expression resource permission evaluation.

For example, you can define the `userGrantedPermission` expression on the `disabled` attribute of the `af:button#cb1` button, as shown in the following example. In this case, the expression tests whether the user has permission and then either enables the button or, when the user does not have permission, disables the button. Because the expression is not defined on the button's `rendered` attribute, the page always displays the button.

```
<af:button actionListener="#{bindings.myMethodName.execute}"
     text="myMethodName"
     disabled="#{!securityContext.userGrantedPermission
       ['resourceName=CancelShipmentButton,resourceType=CancelShipment,
             action=invoke']}
     id="cb1"/>
```

Before you begin:

It may be helpful to have an understanding of the limitations of ADF method authorization checking. For more information, see How to Define Policies to Control User Access to ADF Methods.

You will need to complete this task:

> Create the resource grant for a custom resource type, as described in Creating a Resource Grant to Control Access to ADF Methods.

To check the resource permission using an expression:

1. In the Applications window, double-click the page that contains the command component bound to the ADF method.
2. In the visual editor for the page, select the command component that is used to execute the ADF method.
3. In the Property window, click the **Property Menu** dropdown menu next to the **Disabled** field and choose **Expression Builder**.
4. In the Expression Builder, expand the **ADF Bindings - securityContext** node and select **userGrantedPermission**, and then, in the **Expression** field, enter a concatenated string that defines the permission.

   Enter the permission string as a semicolon-separated concatenation of `resourceName=`*`aResourceName`*`;resourceType=`*`aResourceType`*`;action=`*`actionName`*. For example, to enable or disable a command button used to invoked the method in a page, you would enter an expression similar to the one shown in the above example.

   In the example, the expression determines whether a resource policy grants invoke privileges for the resource type named `CancelShipment` to the user's defined application role. The resource policy must exist in the application policy store to test the expression at runtime.
5. Click **OK**.

What Happens When You Define the Security Policy

When you define a security policy, the overview editor for security policies updates the `jazn-data.xml` file located in the `/src/META-INF` node relative to the web application workspace.

The overview editor writes the policy information to the `<policy-store>` section of the file. The security policy, or grant, contains both a grantee and one or more permissions. The grantee is the application role that the policy is being defined for. Each permission defines the resource being secured and the action that can be performed against that resource.

The following example shows a security policy in the `jazn-data.xml` file that grants unauthenticated users access to an employee registration task flow and a top-level web page used to access the task flow. The grant to the `anonymous-role` application role contains a view permission for a bounded task flow, `emp-reg-task-flow`, and a view permission on the web page with the `indexPageDef` page definition. With this grant, unauthenticated users will be able to enter the employee registration task flow and view the welcome page.

For the web page, notice that permission has been defined on the `indexPageDef` page definition created for the welcome page (`index.jsf`). Also, note that this is a top-level web page that is not already secured by a bounded task flow.

```
<policy-store>
    ...
    <jazn-policy>
        <grant>
            <grantee>
                <principals>
                    <principal>
                         <name>anonymous-role</name>
                         <class>oracle.security.jps.internal.core.principals.
                                       JpsAnonymousRoleImpl</class>
                     </principal>
                 </principals>
             </grantee>
             <permissions>
                  <permission>
                      <class>oracle.adf.controller.security.TaskFlowPermission</
class>
                      <name>/WEB-INF/flows/emp-reg-task-flow-definition.xml#
                                                  #emp-reg-task-flow-definition</
name>
                      <actions>view</actions>
                  </permission>
                   <permission>

<class>oracle.adf.share.security.authorization.RegionPermission</class>
                        <name>oracle.summit.view.pageDefs.indexPageDef</name>
                        <actions>view</actions>
                  </permission>
                  ...
             </permissions>
        </grant>
        ...
    </jazn-policy>
</policy-store>
```

What Happens at Runtime: How ADF Security Policies Are Enforced

Grants that you make for ADF resources are standard JAAS Permissions. When you enable ADF Security in your application, Oracle Platform Security Service (OPSS) running in Oracle WebLogic Server will utilize the grants to allow authorization. In authorization mode, ADF Security uses fine-grained authorization, implemented with JAAS Permissions to perform security checks for access rights to pages. The ADF Security enforcement logic checks to see whether the user, represented by the JAAS subject, has the right permissions to access the resource.

The subject contains the user's principals, which include a user principal that contains their name (could be anonymous, before logging on, or some user name after logging on), and their list of role principals, which would include `authenticated-role` and some number of other roles that are obtained from the policy and identity stores. The principal is created to represent all of the user's memberships in application roles defined in the policy store. In turn, each application role may have multiple Permissions associated with them. These are the ADF security policies that are created through the overview editor for the `jazn-data.xml` file.

> **Note:**
>
> ADF security policies are scoped by application. This scoping allows two applications to refer to the same permission target, without producing unintentional results. You are not required to name application resources to impose application scoping of the policy store information.

Before you run the application using Integrated WebLogic Server, you will need to provision the identity store with test users and add these users to the application roles that you want to configure. The application roles can define members that are specific users or groups of users (also known as **enterprise roles**), as described in Creating Test Users.

Then at runtime, whether the current user has view permission on the page they are trying to access will be determined by the context of the page:

- If the page is an activity of a bounded task flow, the task flow controller determines the permission.
- If the page is a top-level page with an associated page definition file, the **ADF Model** layer determines the permission.

Oracle Platform Security Services then checks to see whether the subject contains the roles that have the corresponding permissions needed to access the page. If the user is authorized, then the task flow is entered.

In the case of a bounded task flow and top-level pages (defined by an unbounded task flow), if the user is not authorized, **ADF Controller** throws an exception and passes control to an exception handler that the task flow configuration specifies. For details about specifying an error page, see How to Redirect a User After Authentication.

What You May Need to Know About Defining Policies for Pages with No ADF Bindings

The default Configure ADF Security wizard option **ADF Authentication and Authorization** enables authorization checking and secures a web page whenever the page is associated with an ADF security-aware resource. Therefore, after you run the wizard, a web page will not be secured if both of these conditions exist:

- The page does not display databound ADF Faces components and therefore no ADF page definition exists for the page.
- The page is not a constituent page of a bounded ADF task flow. (Any page that the user accesses as a process of a bounded task flow is checked under the permission of the task flow.)

JDeveloper will generate an ADF page definition file for you whenever you design a web page using the **Data Controls panel** to create databound ADF Faces components. However, if your web page does not use ADF bindings, you can still create an *empty* page definition file by right-clicking the web page in the user interface project and choosing **Go to Page Definition**. The page definition file can remain empty because the page does not need to work with ADF bindings to support databound ADF Faces components.

Once you associate a web page with an ADF page definition file, empty or not, authorization checking will be enforced when the user accesses the associated web page. You can define security policies for the page as you would any other ADF page definition. For details about making grants to an empty ADF page definition, see How to Define Policies for Web Pages That Reference a Page Definition.

Before you begin:

It may be helpful to have an understanding of security for individual page definition files. For more information, see How to Define Policies for Web Pages That Reference a Page Definition.

You may also find it helpful to understand how JDeveloper normally generates page definition files. For more information, see Working with Page Definition Files.

To create an empty page definition that you can define security policies for:

1. In the Applications window, locate the web page you want to secure, right-click the node and choose **Go to Page Definition**.
2. In the confirmation dialog, click **Yes** to create a new page definition for the page.

   The page definition will be added to the `pageDefs` package.

How to Use Regular Expressions to Define Policies on Groups of Resources

When you want to define a grant that applies to multiple resources at once, you can create patterns as defined by the `java.util.regex.Pattern` class to form a regular expression that gets evaluated at runtime. For example, to match a grant to a set of resources, you can enter the expression `.*` (specifies any character zero or more times) on the name of the permission. ADF Security does not support the use of regular expressions on other security objects, such as the principal name.

You might use this feature to group bounded task flows that would have the same permissions into their own subfolders of `WEB-INF` and define the grant for the entire folder, as shown in the following example. In this case, the expression uses the dot character (defined as, any character) followed by the asterisk quantifier (defined as, zero or more times).

```
...
<grant>
   <grantee>
      <principals>
         <principal>
            <class>oracle.security.jps.service.policystore.ApplicationRole</
class>
            <name>anonymous-role</name>
         </principal>
      </principals>
   </grantee>
   <permissions>
       <permission>
          <class>oracle.adf.controller.security.TaskFlowPermission</class>
          <name>/WEB-INF/.*</name>
          <actions>view</actions>
       </permission>
   </permissions>
</grant>
```

As the overview editor for the jazn-data.xml file does not support the use of regular expressions in the user interface, you must edit the file directly. Do not edit the policy store of the system-jazn-data.xml file directly. Instead, add grants using regular expressions to the jazn-data.xml file. These grants will then be merged to the policy store when you run or deploy the application.

The use of more complex regular expressions enables you to define business rules in the policy, thus creating a very targeted set of permissions. For example, you can grant the view permission on all page definitions and deny specific page definitions at the same time by defining an exclusion set in your regular expression. The following example shows how the view permission is granted to anonymous-role for all pages except those for which the page definition name starts with custom.

```
<grant>
   <grantee>
      <principals>
         <principal>
            <class>oracle.security.jps.service.policystore.ApplicationRole</
class>
            <name>anonymous-role</name>
         </principal>
      </principals>
   </grantee>
   <permissions>
      <permission>
         <class>oracle.adf.share.security.authorization.RegionPermission</class>
         <name>[^(custom)].*</name>
         <actions>view</actions>
      </permission>
   </permissions>
</grant>
```

Table 47-7 shows some of the basic regular expression metacharacters that you can use in your policy definitions.

Table 47-7 Description of Metacharacters

Metacharacter	Description
[abc]	a, b, or c (included in list)
[^abc]	Any character except a, b, or c (negation)
[a-zA-Z]	a to z or A to Z, inclusive (range)
[a-d[m-p]]	a to d, or m to p ~= [a-dm-p](union)
[a-z&&[def]]	d, e, or f (intersection)
[a-z&&[^bc]]	a through z, without b and c: [ad-z] (subtraction)
[a-z&&[^m-p]]	a through z, and not m through p
.*	Any number of arbitrary characters (note this expression uses a dot and an asterisk together)

How to Define Policies for Data

ADF entity objects in the data model project are security-aware, meaning that predefined resource-specific permissions exist that a developer can grant. Additionally, you can secure just the individual attributes of entity objects.

Entity objects that you secure restrict users from updating data displayed by any web page that renders a UI component bound by an ADF binding to the data accessed by the secured entity object. Additionally, when you secure an entity object, you effectively secure any **view object** in the data model project that relies on that entity object. As such, entity objects that you secure define an even broader access policy that applies to all UI components bound to this set of view objects.

To secure row data using ADF entity objects:

1. Define a permission map for the specific actions of the entity object or the attributes of the entity object that you want to secure.
2. Grant the permission to an application role that you have added to the policy store.

Defining Permission Maps on ADF Entity Objects

In the data model project, you use the overview editor for the entity object to define a permission map for the specific actions allowed by the entity object. The metadata consists of a permission class, a permission name, and a set of actions mapped to binding operations.

The list of available operations displayed by the overview editor is defined by the entity object permission class (`oracle.adf.share.security.authorization.EntityPermission`). The permission class maps the operations supported by the entity object to actions. Table 47-8 shows the securable operations of the entity object.

Table 47-8 Securable Operations of ADF Entity Objects

Securable Operation	Expected Mapped Action	Corresponding Implementation
`read`	`read`	View the rows of a result set that has been restricted by a `WHERE` clause fragment.
`update`	`update`	Update any attribute of the bound collection.
`removeCurrentRow`	`delete`	Delete a row from the bound collection.

> **Note:**
>
> Unlike `read` and `update` operations, the securable operation that maps to the `delete` action is defined with the unique identifier, `removeCurrentRow`. Because authorization checks in Fusion web applications rely on the operation name and not the action name, you must test the `removeCurrentRow` operation name rather than the `delete` privilege name. For details about the entity row API and EL expressions that allow you to test entity permissions, see How to Perform Authorization Checks for Entity Object Operations.

To secure all row-level data that the entity object accesses, use the overview editor for the entity object.

Before you begin:

It may be helpful to have an understanding of ADF security for entity objects. For more information, see How to Define Policies for Data.

You will need to complete this task:

> Create entity objects in the data model project, as described in Creating a Business Domain Layer Using Entity Objects.

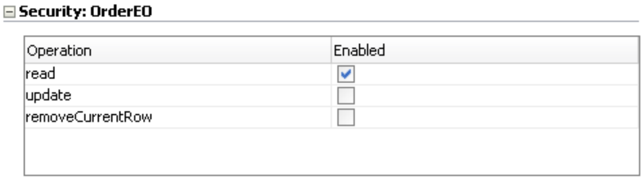
To secure an operation on an entity object:

1. In the Applications window, double-click the entity object that you want to secure.
2. In the overview editor, click the **General** navigation tab.
3. In the General page, expand the **Security** section and select the operations you want to secure for the entity object.

   The **Security** section displays the securable operations that the `EntityPermission` class defines. The class maps the entity object—specific actions to the entity object's operations, as described in Table 47-8.

   For example, to enable **read** permission, select it as shown in Figure 47-11. The permissions appear in the XML definition of the entity object.

Figure 47-11   Permission Enabled on read Operation for an ADF Entity Object



⊟ Security: OrderEO

Operation	Enabled
read	☑
update	□
removeCurrentRow	□

Defining Permission Maps on ADF Entity Object Attributes

In the data model project, you use the overview editor for the entity object to define a permission map for the specific actions allowed by the entity object attribute. The metadata consists of a permission class, a permission name, and a set of actions mapped to binding operations.

The list of available operations displayed by the overview editor is defined by the entity object permission class (`oracle.adf.share.security.authorization.EntityPermission`). The permission class maps the operations supported by entity object attributes to actions. Table 47-9 shows the securable operations of entity object attributes.

Table 47-9   Securable Operations of ADF Entity Object Attributes

Securable Operation	Expected Mapped Action	Corresponding Implementation
`update`	`update`	Update a specific attribute of the bound collection.

To secure individual columns of data that the entity object accesses, use the Attributes page of the overview editor for the entity object.

Before you begin:

It may be helpful to have an understanding of ADF security for entity objects. For more information, see How to Define Policies for Data.

You will need to complete this task:

> Create entity objects in the data model project, as described in Creating a Business Domain Layer Using Entity Objects.

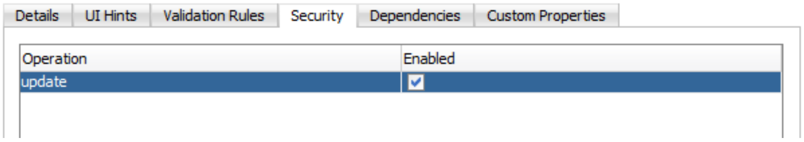
To secure an operation on an entity object attribute:

1. In the Applications window, double-click the entity object that defines the attribute you want to secure.
2. In the overview editor, click the **Attributes** navigation tab.
3. In the Attributes page, select the attribute to secure, and then click the **Security** tab and select the **update** operation.

   The **Security** tab displays the securable operations that the `EntityAttributePermission` class defines. The class maps the entity object-specific actions to the entity object's operations, as described in Table 47-9.

   For example, to enable **update** permission, select it as shown in Figure 47-12. The permission map appears in the XML definition of the entity object.

Figure 47-12 Permission Enabled on update Operation for an ADF Entity Object Attribute



Operation	Enabled
update	☑

Granting Permissions on ADF Entity Objects and Entity Attributes

Once a permission target is configured, any data that derives from entity objects or their attributes remains unsecured until you explicitly define policy grants for the entity object's permission target.

To define the access policy for an existing entity object or entity attribute permission target, use the overview editor for security policies.

Before you begin:

It may be helpful to have an understanding of ADF security for entity objects. For more information, see How to Define Policies for Data.

You will need to complete these tasks:

1. Run the Configure ADF Security wizard, as described in Enabling ADF Security.
2. Create application roles, as described in Creating Application Roles.
3. Define the permission target for the entity object or the attributes of the entity object, as described in Defining Permission Maps on ADF Entity Objects.

To define the access policy for an entity object or entity attribute:

1. In the main menu, choose **Application** and then **Secure > Resource Grants**.

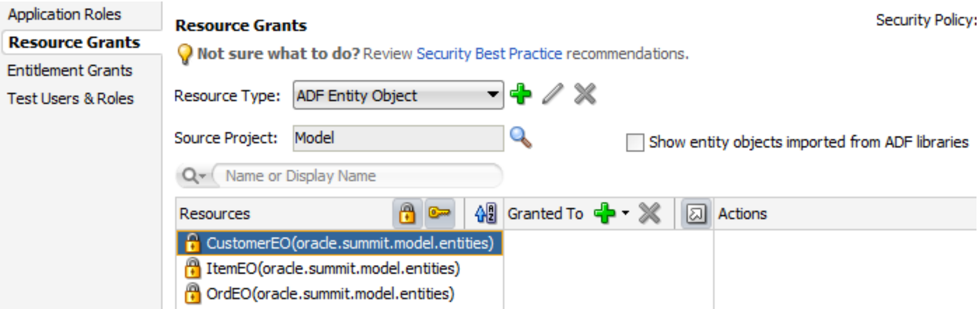
2. In the Resource Grants page of the overview editor for security policies, select **ADF Entity Object** or select **ADF Entity Object Attribute** from the **Resource Type** dropdown list.

   The Resource Grants page of the overview editor displays all entity objects (or entity attributes) for which you previously defined permission targets.

3. In the **Resources** column, select the ADF entity object or entity attribute for which you want to grant access rights.

   The first time you make a grant to an ADF entity object or entity attribute, the first column displays the **Resource without any grants** icon (represented by the "lock" icon) next to the method name, as shown in Figure 47-13. The editor displays the lock icons to indicate that a resource has no security policy defined and therefore is "locked"—which means it remains inaccessible to users until you define a grant.

Figure 47-13 Entity Objects Without Grants in the Overview Editor



Resource Grants

Not sure what to do? Review Security Best Practice recommendations.

Resource Type: ADF Entity Object

Source Project: Model

□ Show entity objects imported from ADF libraries

CustomerEO(oracle.summit.model.entities)
ItemEO(oracle.summit.model.entities)
OrdEO(oracle.summit.model.entities)

4. In the **Granted to Roles** column, click the **Add Grantee** icon and select **Add Application Role**.

5. In the Select Application Roles dialog, select the application role that you want to make a grantee of the permission.

   The Select Application Roles dialog displays application roles from the `jazn-data.xml` file. It also displays the built-in OPSS application roles, **anonymous-role** and **authenticated-role**, as described in What Happens at Runtime: How the Built-in Roles Are Used,

   If you do not see application roles that are specific to your application, create the role, as described in Creating Application Roles.

6. In the Select Application Roles dialog, click **OK**.

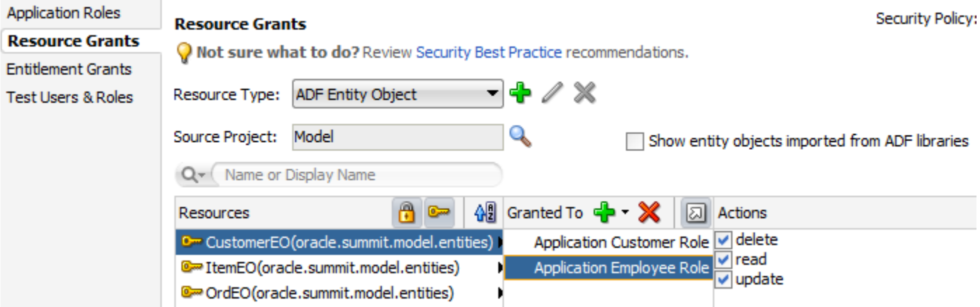
7. In the Resource Grants page of the overview editor, in the **Actions** column, select the action that you want to grant to a specific application role.

   For entity objects, the overview editor shows **read** action selected by default. You may select the **update** action and **delete** action if you have configured their corresponding permission targets, as described in Table 47-8.

   For entity object attributes, the overview editor shows the **update** action selected by default, corresponding to the permission target described in Table 47-9. No other actions are supported for entity object attributes.

   Figure 47-14 shows the **delete**, **read**, and **update** privileges granted to the **Application Employee Role** application role for an entity object.

Figure 47-14 Granting to an Application Role for an ADF Entity Object in the Overview Editor



Resource Grants

Security Policy:

Not sure what to do? Review Security Best Practice recommendations.

Resource Type: ADF Entity Object

Source Project: Model

□ Show entity objects imported from ADF libraries

Name or Display Name

Resources	Granted To	Actions
CustomerEO(oracle.summit.model.entities)	Application Customer Role	☑ delete
ItemEO(oracle.summit.model.entities)	Application Employee Role	☑ read
OrdEO(oracle.summit.model.entities)		☑ update

8. You can repeat these steps to make additional grants as desired.

   The same ADF entity object or entity attribute can have multiple grants made for different application roles. The grants appear in the policy store definition of the `jazn-data.xml` file, as described in What Happens When You Define the Security Policy.

How to Aggregate Resource Grants as Entitlement Grants

You define end user access for individual securable Oracle ADF artifacts by creating resource grants. However, for ease of administration and maintenance, resource grants can also be defined as entitlement grants, in which case multiple securable application artifacts are aggregated into a named security group that can be granted to application roles using a single statement. For example, when you want to authorize access to the data exposed in the application by an ADF Business Components **entity object**, multiple entity objects may require the same security policy. Instead of individually granting access privileges to each entity object, you can group the privileges as a set in an entitlement and grant them all at once.

You create entitlement grants in the Entitlements Grants page of the overview editor for security policies. The grants you create will appear as metadata in the policy store section of the `jazn-data.xml` file. This metadata defines an entitlement (identified in the XML definition as `<permission-set>`) comprised of resource instance /action pairs that you select. This entitlement is a grantable entity that you then grant to an application role.

The list of resource types appears in the overview editor for security policies. The resource type you select filters the resource instances defined within the projects of your application's workspace. The resource type selection also determines the list of available actions displayed by the overview editor. For example, when you select the **Task Flow Permission** resource type, the overview editor will display all of the task flows in the user interface projects that you select and also displays the **view** action that you can associate with the available ADF bounded task flow resources.

Table 47-10 lists the resource types displayed in JDeveloper and identifies the associated resource and supported actions for each type.

Table 47-10 Resource Types of Securable Oracle ADF Artifacts

Resource Type	Supports These Resources and Actions
`Task Flow`	Defines view actions on ADF bounded task flows in a user interface project that you select.
`Web Page`	Defines view actions on regions and web pages backed by ADF page definition files in a user interface project that you select.
`ADF Entity Object`	Defines read, update, and delete actions on entity objects in a data model project that you select.
`ADF Entity Object Attribute`	Defines update actions on entity object attributes of a specific entity object in a data model project that you select.

To define an entitlement grant for a securable Oracle ADF artifact, use the Entitlement Grants page of the overview editor for security policies.

Before you begin:

It may be helpful to have an understanding of ADF security for entity objects. For more information, see How to Aggregate Resource Grants as Entitlement Grants.

You will need to complete this task:

- Create application roles, as described in Creating Application Roles.

To define an entitlement grant for an Oracle ADF artifact:

1. In the main menu, choose **Application** and then **Secure > Entitlement Grants**.
2. In the Entitlement Grants page of overview editor for security policies, click the **Add Entitlements** icon in the **Entitlements** section.

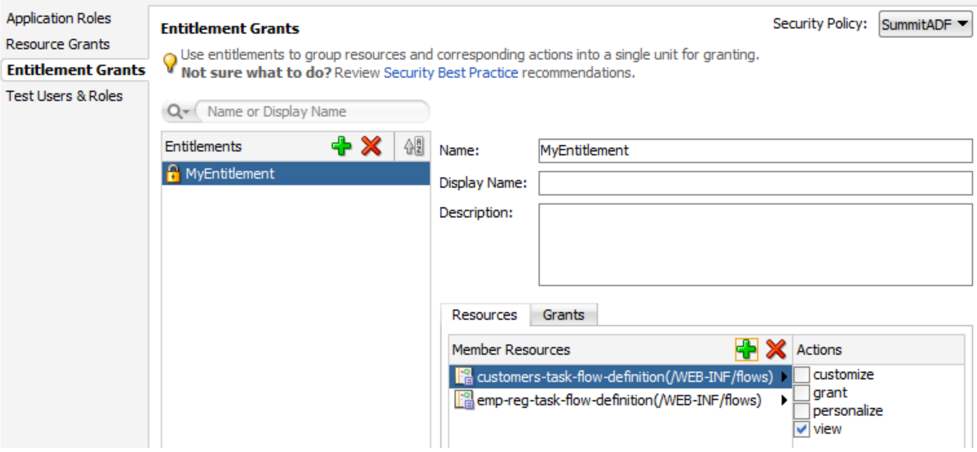
   The overview editor displays all the resources that your application defines.
3. In the Entitlement Grants page, click the **Resources** tab and then click the **Add Member Resource** icon to add a member resource to the entitlement.
4. In the Select Resources dialog, select the resource from the **Resource Type** dropdown and then select the desired project in the **Source Projects** section.

   The dialog displays all the projects in your application workspace.
5. In the **Available Resources** section, select the resource and click the **Add** icon.

   The dialog displays all the resources define by your selected project.
6. In the **Actions** lists, select the desired action for the selected resource.

   Figure 47-15 shows the overview editor with the **View** action selected for the task flow and added to **MyEntitlement**.

Figure 47-15 Adding a Bounded Task Flow as a Resource in an Entitlement Grant



Security Policy: SummitADF

Use entitlements to group resources and corresponding actions into a single unit for granting.
Not sure what to do? Review Security Best Practice recommendations.

MyEntitlement

Name: MyEntitlement
Display Name:
Description:

customers-task-flow-definition(/WEB-INF/flows)
emp-reg-task-flow-definition(/WEB-INF/flows)

□ customize
□ grant
□ personalize
☑ view

7. Add other desired resources to the list.
8. In the Entitlement Grants page, click the **Grants** tab and then click the **Add Role Grants** icon to grant the entitlement to an application role.
9. In the Select Application Roles dialog, select one or more custom application roles.

   The dialog displays all the application roles from the `jazn-data.xml` file. You must not add a grant to a predefined application role (also called **duty roles** in the terminology of Oracle Fusion Applications). Only select custom application roles that either you created in JDeveloper or that were created by an IT security manager for this purpose.
10. Click **OK**.
11. You can repeat these steps to add other resources and make grants on those resources to the same entitlement for the same custom application role.

What Happens After You Create an Entitlement Grant

When you use the security policy editor in JDeveloper to create an entitlement grant, JDeveloper modifies the source for the application policy store in the `jazn-data.xml` file. The policy store section of the file contains a `<resource-type>` definition (that identifies the actions supported for resources of the selected type), a `<resource>` definition (to identify the resource instance that you selected from your application and mapped to a resource type), a `<permission-set>` definition (to define the resources and actions to be granted as an entitlement), and a `<grant>` definition with one or more entitlements (defined in the XML as a permission set) granted to the desired application roles (the grantee).

As the following example illustrates, entitlement-based security policies in the Oracle Fusion application are defined in the `<jazn-policies>` element and consist of one or more entitlements granted to a single application role.

```
<?xml version="1.0" ?>
<jazn-data>
  <policy-store>
    <applications>
      <application>
  <name>MyApp</name>

<app-roles>
  <app-role>
    <name>AppRole</name>
    <display-name>AppRole display name</display-name>
    <description>AppRole description</description>
    <guid>F5494E409CFB11DEBFEBC11296284F58</guid>
    <class>oracle.security.jps.service.policystore.ApplicationRole</class>
  </app-role>
</app-roles>

<role-categories>
  <role-category>
    <name>MyAppRoleCategory</name>
    <display-name>MyAppRoleCategory display name</display-name>
    <description>MyAppRoleCategory description</description>
  </role-category>
</role-categories>

<!-- resource-specific OPSS permission class definition -->
<resource-types>
  <resource-type>
    <name>APredefinedResourceType</name>
    <display-name>APredefinedResourceType display name</display-name>
    <description>APredefinedResourceType description</description>
    <provider-name>APredefinedResourceType provider</provider-name>
    <matcher-class>oracle.security.jps.ResourcePermission</matcher-class>
    <actions-delimiter>,</actions-delimiter>
    <actions>write,read</actions>
  </resource-type>
</resource-types>

<resources>
  <resource>
    <name>MyResource</name>
    <display-name>MyResource display name</display-name>
    <description>MyResource description</description>
    <type-name-ref>APredefinedResourceType</type-name-ref>
  </resource>
</resources>

<!-- entitlement definition -->
<permission-sets>
  <permission-set>
    <name>MyEntitlement</name>
    <display-name>MyEntitlement display name</display-name>
    <description>MyEntitlement description</description>
    <member-resources>
      <member-resource>
        <type-name-ref>APredefinedResourceType</type-name-ref>
        <resource-name>MyResource</resource-name>
        <actions>write</actions>
      </member-resource>
    </member-resources>
  </permission-set>
</permission-sets>

<!-- Oracle function security policies -->
<jazn-policy>
  <!-- function security policy is a grantee and permission set -->
```

```
        <grant>
          <!-- application role is the recipient of the privileges -->
          <grantee>
            <principals>
              <principal>
                <class>
                   oracle.security.jps.service.policystore.ApplicationRole
                </class>
                <name>AppRole</name>
                <guid>F5494E409CFB11DEBFEBC11296284F58</guid>
              </principal>
            </principals>
          </grantee>

          <!-- entitlement granted to an application role -->
          <permission-set-refs>
            <permission-set-ref>
              <name>MyEntitlement</name>
            </permission-set-ref>
          </permission-set-refs>
        </grant>
      </jazn-policy>
    </application>
  </applications>
 </policy-store>
</jazn-data>
```

Creating Test Users

For testing purpose, ADF Security allows developers to create users in the jazn-data.xml file.

JDeveloper provides editors to help you create both the identity and the policy stores. You create both repositories in an application-specific `jazn-data.xml` file. The editor for the identity store section of the file lets you enter the list of valid user IDs and their assigned passwords. The same editor lets you create application roles and assign the test users or enterprise roles as members of the application roles. Once defined, this information appears in the policy store section of the `jazn-data.xml` file.

How to Create Test Users in JDeveloper

You seed the identity store of your application with a temporary set of users to simulate the actual users' experience in your production environment. When you run the application in Integrated WebLogic Server, you can log in as any test user and be conferred access rights to view the secure ADF resources of your application.

You can use the identity store to organize users into enterprise roles. Because you typically will configure JDeveloper's deployment options to prevent migrating the identity store to a staging environment, enterprise roles that you create in the `jazn-data.xml` file are for convenience only. For more details about the use of enterprise roles, see What You May Need to Know About Enterprise Roles and Application Roles.

> **Caution:**
>
> If you choose to deploy the identity store to your standalone server, you must not create users and enterprise roles in your local identity store that are already configured for Oracle WebLogic Server. For example, if you were to deploy the identity store with the user `weblogic` and enterprise role `Administrators`, you would overwrite the default administration configuration on the target server. For a complete list of global roles that Oracle WebLogic Server installs by default, see the Users, Groups, and Security Roles chapter in *Securing Resources Using Roles and Policies for Oracle WebLogic Server*.

To enable the user to view resources, you make grants against application roles rather than against the users who are the members of those roles. Therefore, after you seed the identity store with test users, you must associate each user or enterprise role group with an application role. This association confers the access rights defined by ADF security policies to users. For details about conferring access rights to users, see How to Associate Test Users with Application Roles.

You should avoid choosing a user name already configured for Oracle WebLogic Server (for example, do not enter `webcenter`). For the list of user names installed by Oracle WebLogic Server, see the Users, Groups, and Security Roles chapter in *Securing Resources Using Roles and Policies for Oracle WebLogic Server*.

Before you begin:

It may be helpful to have an understanding of the identity store. See Creating Test Users.

To create test users and groups:

1. In the main menu, choose **Application** and then **Secure > Test Users & Roles**.
2. In the Test Users & Roles page of the overview editor for security policies, select the realm for your application from the **Realm** dropdown list and perform the following steps.

   JDeveloper uses the realm `jazn.com` by default.

   a. In the **Users** list, click the **New User** icon.
   b. In the **Name** field, enter the user name.
   c. In the **Password** field, enter the password for the user and click any other field to add the password to the identity store.

      The password must contain at least eight characters and at least one of the characters must be a special character (such as !, %, ^, &, $ and so on).
3. Optionally, in the overview editor for security policies, click the **Enterprise Roles** navigation tab, select the realm for your application from the **Realm** dropdown list, and perform the following steps.

   You create enterprise roles only when you want to organize users into groups that you will add to an application role. For the purpose of creating test users to run the application using Integrated WebLogic Server, you do not need to create enterprise role groups.

   a. In the **Enterprise Roles** list, click the **New Role** icon.

b. In the **Name** field, enter the name of the enterprise role and click any other field to add the role to the identity store.

   If you create enterprise role groups, you should avoid choosing a role name that is already configured for Oracle WebLogic Server (for example, do not enter `Administrators`). For a complete list of the default group names installed by Oracle WebLogic Server, see Users, Groups, and Security Roles in *Securing Resources Using Roles and Policies for Oracle WebLogic Server*.

What Happens When You Create Test Users

When you provision the identity store with user identities and enterprise role groups, JDeveloper updates the `jazn-data.xml` file located in the `/src/META-INF` node relative to the web application workspace.

The dialog writes the user information to the `<jazn-realm>` section of the file corresponding to the identity store. Each user identity has a user name and a user login password. Each enterprise role contains one or more member users.

The following example shows the identity store in the `jazn-data.xml` file with two users and two enterprise roles. The user `cmagee` is a member of the `Enterprise Employee Group` enterprise role, while user `214` is a member of the `Enterprise Customer Group` enterprise role.

```
<jazn-data>
   <jazn-realm default="jazn.com">
      <realm>
         <name>jazn.com</name>
         <users>
            <user>
               <name>cmagee</name>
               <display-name>Colin Magee</display-name>
               <description>Sales Rep</description>
               <credentials>{903}5AUHlE+qvDxxAhxIMoXJ/WLhMF0ynue7</credentials>
            </user>
            <user>
               <name>214</name>
               <display-name>Ojibway Retail</display-name>
               <description>Customer</description>
               <credentials>{903}rijN2KQCBSeBQ/JNZlv8GwwUiGWk5IQa</credentials>
            </user>
            ...
         </users>
         <roles>
            <role>
               <name>Enterprise Employee Group</name>
               <members>
                  <member>
                     <type>user</type>
                     <name>cmagee</name>
                  </member>
               </members>
            </role>
            <role>
               <name>Enterprise Customer Group</name>
               <members>
                  <member>
                     <type>user</type>
                     <name>214</name>
```

```
                        </member>
                        ...
                    </members>
                </role>
                ...
            </roles>
        </realm>
        ...
    </jazn-realm>
</jazn-data>
```

How to Associate Test Users with Application Roles

Because the ADF Security framework enforces a role-based access control mechanism with permissions granted to application roles, you define a set of roles in the policy store that are specific to your application. For example, in the context of the work flow, there may be roles such as customer, product specialist, supervisor, and administrator.

After you create an application role, you can proceed to associate users that you created in the identity store with one or more roles. At runtime, users who are members of an application role will be conferred the access rights of their application roles. You can assign a user to more than one application role when you want to confer the right of multiple resource grants to a particular user.

For example, one authenticated user might belong to the supervisor role and an employee role, while another user might belong only to the employee role. The security policy for a bounded task flow that permits customer records to be browsed and edited may confer view permission to the supervisor role and limit view permission to the browse page for the employee role. Thus, grants to application roles support multiple levels of access. If the authenticated user is not a member of an application role with a view permission grant for the target ADF resource, the security framework will return an unauthorized user message.

Before you begin:

It may be helpful to have an understanding of the identity store. For more information, see Creating Test Users.

You will need to complete these tasks:

1. Run the Configure ADF Security wizard, as described in Enabling ADF Security.
2. Create application roles, as described in Creating Application Roles.
3. Define security policies for ADF security-aware resources, as described in Defining ADF Security Policies.
4. Create test users, and, optionally, create enterprise role groups, as described in How to Create Test Users in JDeveloper.

To associate users with application roles:

1. In the main menu, choose **Application** and then **Secure > Application Roles**.
2. In the Application Roles page of the overview editor for security policies, select the policy store for your application from the **Security Policy** dropdown list.

   The policy store that JDeveloper creates in the `jazn-data.xml` file are automatically based on the name of your application.

3. In the **Roles** list, select an existing application role and complete these tasks as appropriate:
   a. In the **Mappings** section, click the **Add User or Role** icon dropdown menu and choose **Add User**, then in the Select Users dialog select the previously created user from the list and click **OK**.
   b. Optionally, if you have defined enterprise roles in the identity store, in the **Mappings** section, click the **Add User or Role** icon dropdown menu and choose **Add Enterprise Role**, then in the Select Enterprise Roles dialog select the previously created enterprise role from the list and click **OK**.

What Happens When You Configure Application Roles

When you associate users with application roles, JDeveloper updates the `jazn-data.xml` file located in the `/src/META-INF` node relative to the web application workspace.

The dialog writes the user information to the `<policy-store>` section of the file. Each application role contains one or more member users or enterprise roles.

The following example shows the policy store in the `jazn-data.xml` file with the `Application Employee Role` application role, which contains two members, `Enterprise Employee Group` and `Enterprise Manager Group`.

```
<policy-store>
     <applications>
         <application>
            <name>SummitADF</name>
            <app-roles>
               <app-role>
                  <name>Application Employee Role</name>

<class>oracle.security.jps.service.policystore.ApplicationRole</class>
                  <display-name>Application Employee Role</display-name>
                  <members>
                     <member>
                        <class>oracle.security.jps.internal.core.principals.

JpsXmlEnterpriseRoleImpl</class>
                        <name>Enterprise Employee Group</name>
                     </member>
                     <member>
                        <class>oracle.security.jps.internal.core.principals.

JpsXmlEnterpriseRoleImpl</class>
                        <name>Enterprise Manager Group</name>
                     </member>
                     ...
                  </members>
               </app-role>
               ...
            </app-roles>
            <jazn-policy>
              ...
            </jazn-policy>
         </application>
    </applictions>
</policy-store>
```

Creating a Login Page

Both implicit and explicit authentication are supported by ADF Security. You can create a login link, a login page, a welcome page and, perform various other actions.

ADF Security allows for implicit and explicit authentication:

- In an **implicit authentication** scenario, if a user who is not yet authenticated tries to access a web page associated with ADF security-aware resources that are not granted to `anonymous-role`, then authentication is triggered dynamically. After the user successfully logs in, another check will be done to verify whether the authenticated user has view access granted on the requested page's ADF security-aware resource.
- In an **explicit authentication** scenario, your application has a public page that displays a login link, which, when clicked, triggers an authentication challenge to log in the user. The login link may optionally specify some other target page that should be displayed (assuming the authenticated user has access) after the successful authentication.

The implicit and explicit authentication scenarios are handled for you by default when you run the Configure ADF Security wizard, as described in What You May Need to Know About ADF Authentication. However, when you customize the default, generated login page (or supply your own page) to use ADF Faces components, you will need to configure the container-managed deployment descriptor (`web.xml` file).

To explicitly handle user authentication:

1. Create a login link component and add it to the public home web page for your application.
2. Create a managed bean using API specific to Servlet 3.0 to handle the login attempts by the user.
3. Create a JSF login page using ADF Faces components.
4. Ensure that the login page's resources are accessible.

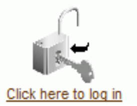
For more information about implicit and explicit authentication, see What Happens at Runtime: How ADF Security Handles Authentication.

How to Create a Login Link Component and Add it to a Public Web Page for Explicit Authentication

You can create a standard login link component that can be added to any page in your application to enable users to authenticate or subsequently log off. This component keeps track of the authenticated state of the user and returns the appropriate login or logout URLs and icons. The login link component will redirect users back to a specific page once they are authenticated. Hence, using this login link component provides you with a single, consistent object.

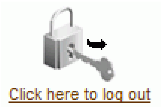
To the unauthenticated user, the login link component will look similar to Figure 47-16.

Figure 47-16 Component Before User Logs In



Then when the user clicks the login link and logs in as a user with the appropriate credentials, the component will look similar to Figure 47-17.

Figure 47-17 Component After User Logs In



Before you begin:

It may be helpful to have an understanding of ADF authentication. For more information, see Creating a Login Page.

You will need to complete this task:

> Copy your login and logout image files (GIF, JPG, or PNG files) to the `public_html` directory of your project.

> **Note:**
>
> The images used should reference the appropriate skin image if your application uses skins. For more information about skins, see the Customizing the Appearance Using Styles and Skins chapter in *Developing Web User Interfaces with Oracle ADF Faces*.

To create the login link component and add it to a page:

1. In the Applications window, double-click the web page that will display the component.
2. In the ADF Faces page of the Components window, from the Common Components panel, drag a **Link** and drop it on the page.
3. In the Structure window, right-click **af:link** and choose **Go to Properties**.
4. In the Property window, to specify the label for the Link component, enter an Expression Language (EL) expression in the **Text** field.

   For example, you can enter an EL expression similar to this to conditionally render the link text:

   ```
   #{securityContext.authenticated ? &quot;Click to log out&quot; :
                                     &quot;Click to log in&quot;}
   ```

The `authenticated` property of the `securityContext` bean will evaluate to true and render the log out option if the current user is authenticated. Otherwise, the link is rendered with the login option.

5. In the Property window, to specify the URL for the Link component, enter an EL expression in the **Destination** field.

   For example, this EL expression conditionally renders a URL to forward the user depending on whether or not they are authenticated:

```
#{securityContext.authenticated ? \"Logout\" : \"Login\"}" id="gl2"
destination="#{securityContext.authenticated ?
                                              \"/adfAuthentication?
logout=true\" : \"/adfAuthentication?login=true\"}
```

   When the user clicks the link, the destination will be determined by the `success_url` and `end_url` settings on the ADF authentication servlet `init-param`. Note that these settings should specify the view activity name instead of the web page name to ensure control flow rules are enforced for page navigation after login. The `authenticated` property of the `securityContext` bean will evaluate to true and forward to the page defined by `success_url` if the current user is authenticated. When the user is not authenticated, there is no need to forward to the login page because the ADF authentication servlet triggers log in, which is handled by the container-managed security configuration. Note that log out is handled by the ADF authentication servlet which invalidates the session.

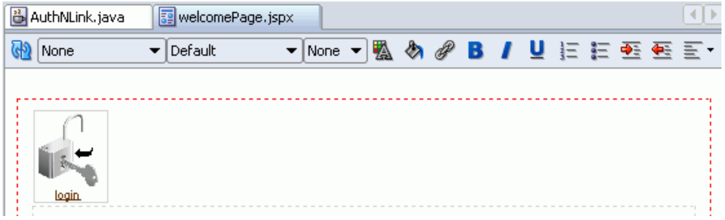
6. In the Property window, to specify the link component image for the Link component, enter an EL expression in the **Icon** field.

   For example, this EL expression conditionally renders the link component image as the lock GIF if the user is not authenticated; otherwise, renders the image with the key GIF:

```
#{securityContext.authenticated ? '/images/lock.gif' : '/images/key.gif'}
```

   Figure 47-18 shows how the login link component appears when added to the global menu facet of the page.

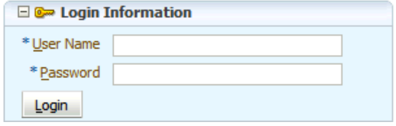
Figure 47-18 Login Link Component on the Page



How to Create a Login Page Specifically for Explicit Authentication

The default login form that is generated for you when you run the Configure ADF Security wizard is provided as a convenience for testing your application within JDeveloper. The default form does not allow you to customize the page using ADF Faces components to match the user interface of the application. You can replace the default form with an ADF Faces-based login page that enables you to include customizable components, as shown in Figure 47-19.

Figure 47-19 Login Page



However, if designing a login page with ADF Faces components is not a requirement, then a simple JSP or HTML login page can be also used. For details about generating a simple login page when running the Configure ADF Security wizard, see How to Enable ADF Security.

Creating Login Code for the Backing Bean

Before you create the login page as an ADF Faces page, you need to create a managed bean to handle login attempts. In this login bean sample, authentication is handled programmatically using Servlet 3.0-specific API. The login bean handles the login action initiated from a login link with the `doLogin()` method. Upon successful login, this method invokes the servlet API to redirect to a landing page. You will add this bean to the `adfc-config.xml` file and register it with `request` scope.

> **Note:**
>
> This backing bean example described in this section does not authenticate with Oracle Single Sign-On (Oracle SSO). The backing bean handles authentication using login and logout methods provided by Servlet 3.0, which provides generic support for programmatic login on application servers that support Java EE 6 or later.

Before you begin:

It may be helpful to have an understanding of the login page. For more information, see How to Create a Login Page Specifically for Explicit Authentication.

You will need to complete this task:

Run the Configure ADF Security wizard to enable ADF Authentication and Authorization.

To create and register a backing bean for login:

1. In the Applications window, right-click the project in which you want to create the backing bean and choose **New** and then **From Gallery**.
2. In the New Gallery, expand **General**, select **Java** and then **Class**, and click **OK**.
3. In the Create Java Class dialog, enter the name for the login page backing bean class file and disable the default options **Constructors from Superclass** and **Implement Abstract Methods**, and click **OK**.

   For convenience, you might name the backing bean based on the name of your login page, for example, ***LoginPageName*.java**.
4. In the Applications window, expand **Application Sources** and double-click the new ***LoginPageName*.java** backing bean.

5. In the source editor, create two private fields by adding the following in the declaration section of the *LoginPageName*`.java` file:

```
private String _username;
private String _password;
```

6. Generate or create public accessors for both fields.

   You can right-click in the source editor and choose **Generate Accessors** to add the following public accessors to the file:

```
public void setUsername(String _username) {
    this._username = _username;
}

public String getUsername() {
    return _username;
}

public void setPassword(String _password) {
    this._password = _password;
}

public String getPassword() {
    return _password;
}
```

7. Import the following classes:

```
javax.faces.application.FacesMessage
javax.faces.context.ExternalContext
javax.faces.context.FacesContext

javax.servlet.ServletException
javax.servlet.http.HttpServletRequest
javax.servlet.http.HttpSession
```

8. Add a `doLogin()` method to this Java class to handle user attempts to log in:

```
public String doLogin() {
  FacesContext ctx = FacesContext.getCurrentInstance();

  if (_username == null || _password == null) {
     showError("Invalid credentials",
                 "An incorrect username or password was specified.", null);
  } else {
      ExternalContext ectx = ctx.getExternalContext();
      HttpServletRequest request = (HttpServletRequest)
                                   ctx.getExternalContext().getRequest();
      try {
          request.login(_username, _password); // Servlet 3.0 login
          _username = null;
          _password = null;
          HttpSession session = request.getSession();
          session.setAttribute("success_url", "/faces" +
ctx.getViewRoot().getViewId());
          redirect(ectx.getRequestContextPath() + "/adfAuthentication");
      } catch (ServletException fle) {
          showError("ServletException", "Login failed.
             Please verify the username and password and try again.",
null);
      }
22   }
23   return null;
24 }
```

The `doLogin()` method performs the following tasks:

Line 2 and **Lines 6-9** get an object encapsulating the HTTP request from the `ExternalContext`.

Lines 3-5 handle an error when credentials have not been provided. The `showError()` method is a method which you will implement to deal with miscellaneous problems with the login process.

Lines 11-13 attempt to log in the user issuing the request using the Servlet 3.0 API `login()` method. In this programmatic login example, which relies on the Servlet API, Oracle Single Sign-On (Oracle SSO) is not supported.

Lines 14-16 create an HTTP session and set the ADF authentication servlet parameters `success_url` as an attribute of the session. Setting `success_url` on the session ensures the destination page URL is not passed as a request parameter that would be subject to malicious redirects. In this example, the destination page is the current page obtained from the Faces context.

Note: If you run the ADF Security wizard and use the wizard to create a login form, upon successful login, you would want the session to redirect to the target protected page instead of returning to the login page. In this case, `session.setAttribute` in lines 15 and 16 should take `success_url` and the full path of the destination page as attribute, for example: `/faces/protectedPage`.

Line 17 calls a method, `redirect()` which you will implement later in this section to forward the user to the URL that the ADF authentication servlet gets from the HTTP session, as set in Line 15.

Lines 18-21 handle a `ServletException`, which can be thrown by many different problems with a login. For example, exceptions can result from incorrect credentials or attempts to log into a locked account or uses of an expired password. The `showError()` method is a method which you will implement to deal with miscellaneous problems with the login process.

Line 23 returns null so that ADF Controller will not attempt to follow a control flow case.

9. Create stubs for the methods `redirect()` and `showError()`.

10. Add a `redirect()` method with its actions:

```
1  private void redirect(String forwardUrl) {
2    FacesContext ctx = FacesContext.getCurrentInstance();
3    ExternalContext ectx = ctx.getExternalContext();
4    try {
5      ectx.redirect(forwardUrl);
6    } catch (IOException ie) {
7        showError("IOException", "An error occurred during redirecting.
8                        Please consult logs for more information.", ie);
9    }
10 }
```

The `redirect()` method performs the following tasks:

Line 2 gets the ADF Faces `ctx`, which forwards a response to a particular URI.

Lines 3-5 use the `ctx` to redirect the current HTTP response to the URL.

Lines 6-8 handle an `IOException`, which is thrown when the request cannot be read or the response cannot be written to.

11. Implement a `showError()` method:

```
private void showError(String errType, String message, Exception e){
  FacesMessage msg =
        new FacesMessage(FacesMessage.SEVERITY_ERROR, errType, message);
  FacesContext.getCurrentInstance().addMessage("d2:it35", msg);
  if (e != null) {
      e.printStackTrace();
  }
}
```

    This `showError()` method adds a summary error message to the `FacesContext`, and then prints the full stack trace of the exception to the console.

12. Optionally, when you want to trigger logout programmatically from a task flow method call activity, create a stub for a `logoff()` method and implement it as:

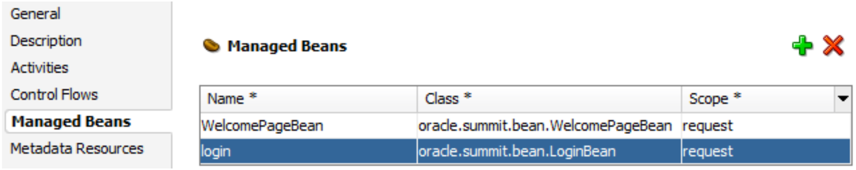
```
public String logoff() {
  FacesContext ctx = FacesContext.getCurrentInstance();
  ExternalContext ectx = ctx.getExternalContext();
  HttpServletRequest httpRequest = (HttpServletRequest) ectx.getRequest();
  try {
      httpRequest.logout(); // Servlet 3.0 logout
      HttpSession session = httpRequest.getSession(false);
      if (session != null) {
          session.invalidate();
      }
      String logoutUrl = ectx.getRequestContextPath() + "/faces" +
                            ctx.getViewRoot().getViewId();
      redirect(logoutUrl);
  } catch (ServletException e) {
      showError("ServletException", "An error occurred during logout.
                  Please consult logs for more information.", e);
  }
  return null;
}
```

    This `logoff()` method uses the `logout()` method of Servlet 3.0 API to process a logoff action.

13. Save the Java file.
14. In the Applications window, expand **WEB-INF** and double-click **adfc-config.xml**.
15. In the editor window, click the **Overview** tab.
16. In the overview editor, click the **Managed Beans** navigation tab.
17. In the Managed Beans page, in the **Managed Beans** section, click the **Add** icon and enter a name for the bean, enter the fully qualified class name, and select scope **request**.

    For example, the class name might look like `oracle.summit.bean.LoginBean`, as shown in Figure 47-20.

Figure 47-20 Login Bean Registered in adfc-config.xml File



Name *	Class *	Scope *
WelcomePageBean	oracle.summit.bean.WelcomePageBean	request
login	oracle.summit.bean.LoginBean	request

18. Save all.

Creating an ADF Faces-Based Login Page Specifically for Explicit Authentication

A simple login page that utilizes ADF Faces layout components and ADF Faces user interface components includes two input fields and a button. You must bind the properties of these UI components to the login handler methods that you defined in the managed bean for the login page.

Note that the page that you can create with this procedure does not support implicit authentication. When you choose to implement implicit authentication, you can customize the default, wizard-generated login form which supports container-managed authentication. The Java EE container expects a form that relies on the `j_security_check` mechanism to handle user-submitted `j_username`, and `j_password` input. In the explicit authentication scenario documented here, Java EE container-managed authentication is not used.

Before you begin:

It may be helpful to have an understanding of explicit authentication. For more information, see How to Create a Login Page Specifically for Explicit Authentication.

You will need to complete this task:

Create the managed bean to handle the user's login attempts, as described in Creating Login Code for the Backing Bean.

To create the ADF Faces-based login page for explicit authentication:

1. In the Applications window, right-click the project in which you want to create the login page and choose **New** and then **New Gallery**.
2. In the New Gallery, expand **Web Tier**, select **JSF** and then **Page**, and click **OK**.
3. In the Create JSF Page dialog, select **JSP XML**.
4. In the **File Name** field, specify a name for your login page. For example, enter `LoginPage.jspx`.

   Select no other options and do not select the option to expose UI components in a managed bean. You will manually bind the components to the managed bean you created for the login page.
5. Click **OK**.
6. Save the page.
7. In the ADF Faces page of the Components window, from the Layout panel, drag a **Panel Box** and drop it on the Structure window below the **af:form** node.
8. In the Property window, enter values for the **Text**, **Horizontal**, **Width**, and **Height** fields.

For example, to create a basic login page, you can enter:

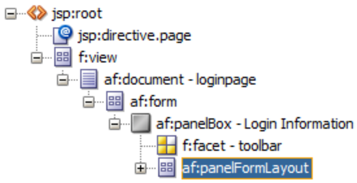
Text set to `Login Information`

Icon set to `/images/key_ena.png`

Width/**Height** set to `300`/`200` pixels

9. From the Components window, drag a **Panel Form Layout** and drop it below the **af:panelBox** node in the Structure window, as shown in Figure 47-21.

Figure 47-21 Login Page Structure with Panel Form Layout



```
jsp:root
    jsp:directive.page
    f:view
        af:document - loginpage
            af:form
                af:panelBox - Login Information
                    f:facet - toolbar
                    af:panelFormLayout
```

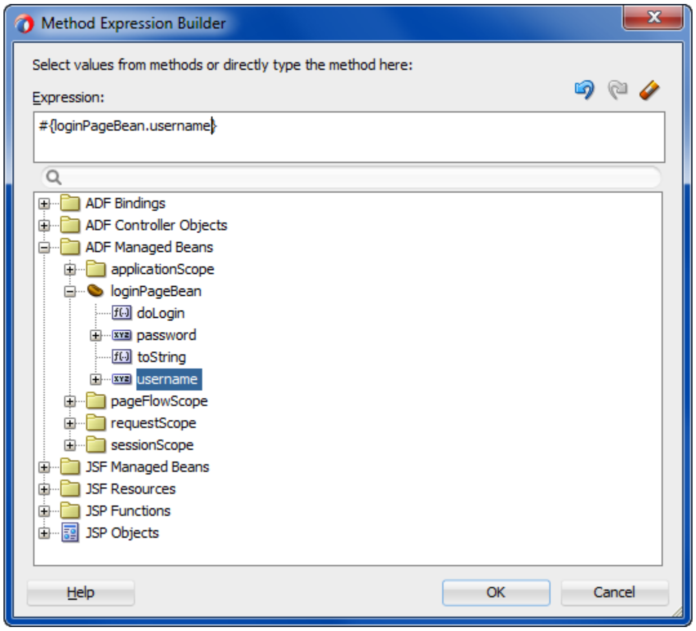
10. From the Common Components panel, drag an **Input Text** and drop it on the **Panel Form Layout** node for the **username** field and another **Input Text** for the password field.
11. In the Property window for the input fields, enter `Username` and `Password` in the **Label** fields and expand the **Behavior** section and select **true** from both fields' **Required** dropdown list.
12. In the Property window, for the password field, expand the **Appearance** section select **true** from the **Secret** dropdown list.
13. To handle processing of the values for the two input fields, perform these steps for each field:

    a. In the Structure window, select one of the input fields (for example, select the **af:inputText - Username** node), and then in the Property window click the **Property Menu** dropdown menu next to the **Value** field and choose **Expression Builder**.

    b. In the Expression Builder, expand **ADF Managed Beans** and expand your login bean, and then select the expression value corresponding to your bean's handler method.

       For example, if you selected the **username** input field in the Structure window, then you would select **username** from the Expression Builder dialog, as shown in Figure 47-22.

Figure 47-22 username Selection in Expression Builder Dialog



Method Expression Builder

Select values from methods or directly type the method here:

Expression:

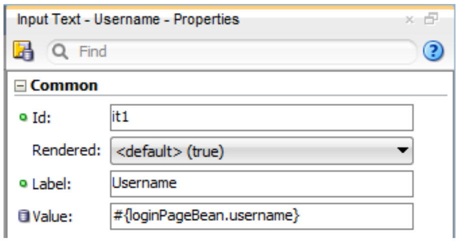
#{loginPageBean.username}

- ADF Bindings
- ADF Controller Objects
- ADF Managed Beans
  - applicationScope
  - loginPageBean
    - doLogin
    - password
    - toString
    - username
  - pageFlowScope
  - requestScope
  - sessionScope
- JSF Managed Beans
- JSF Resources
- JSP Functions
- JSP Objects

c. Click **OK**.

The expression shown in the Property window binds the field to the managed bean you created for the login page. For example, for the username input field, the expression is similar to `#{loginPageBean.username}` as shown in Figure 47-23.

Figure 47-23 username Value in Property Window



Input Text - Username - Properties

Common

Id: it1

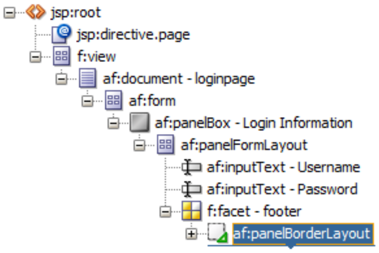
Rendered: <default> (true)

Label: Username

Value: #{loginPageBean.username}

14. From the Components window, from the Layout panel, drag a **Panel Border Layout** and drop it inside the **footer** node of the **af:panelBox** node, as shown in Figure 47-24.

Figure 47-24 Login Page Structure with Panel Border Layout



- jsp:root
  - jsp:directive.page
  - f:view
    - af:document - loginpage
      - af:form
        - af:panelBox - Login Information
          - af:panelFormLayout
            - af:inputText - Username
            - af:inputText - Password
            - f:facet - footer
              - af:panelBorderLayout

15. In the JSP/HTML visual editor, delete the panel border layout facets labeled **End**, **Top**, and **Bottom**. Leave only the **Start** facet.

16. In the Structure window, expand **Panel Border Layout facets** and select the **start** node, and then from the Components window drag and drop a **Button**.

17. In the Property window, enter `Login` in the button component's **Text** field.

18. To handle processing of the login button action, perform these steps.

    a. In the Structure window, select **af:button** under the **start** node, and then in the Property window, click the **Property Menu** dropdown menu next to the **Action** field and choose **Expression Builder**.

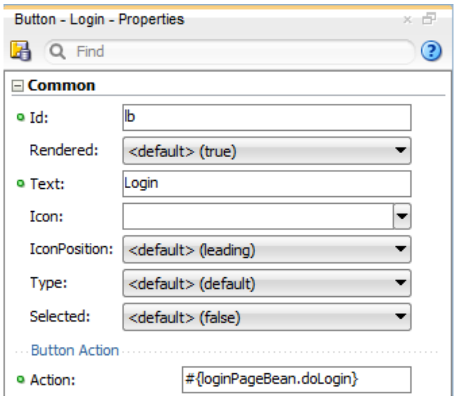
    b. In the Expression Builder, expand **ADF Managed Beans** and expand your login bean, and then select the expression value corresponding to your login method.

       For example, if you created a method in the login bean named `doLogin()`, then you would select **doLogin** in the Expression Builder dialog.

    c. Click **OK**.

       The expression shown in the Property window binds the button to the managed bean you created for the login page. For example, for the login button, the expression is similar to `#{loginPageBean.doLogin}` as shown in Figure 47-25.

Figure 47-25 login Action in Property Window



Button - Login - Properties

Common

Id: lb

Rendered: <default> (true)

Text: Login

Icon:

IconPosition: <default> (leading)

Type: <default> (default)

Selected: <default> (false)

Button Action

Action: #{loginPageBean.doLogin}

19. Save the page.

Ensuring That the Login Page Is Public

Because the application is secured by ADF Security, all web pages defined within bounded task flows and any web page defined by an ADF page definition will be inaccessible by default. Since all users must be allowed to log on, the login page should remain publicly accessible, and thus you should add no databound components to the page. As long as the login page uses no databound components, then it will be accessible by default.

No further steps are required to ensure that the container will always redirect to the defined authentication point before allowing access to the page (which in this case is the authentication page).

How to Ensure That the Custom Login Page's Resources Are Accessible for Explicit Authentication

When you run the ADF Security wizard and choose the **ADF Authentication** option (because you do not want to enable ADF authorization) and your application uses a custom login page or a custom error page, you may need to edit the default Java EE security constraint added to the `web.xml` file by ADF Security. As shown in the following example, the default URL pattern (`/*`) defined in the `allPages` security constraint covers everything under the Java EE application root, meaning that the resource files (such as images, style sheets, or JavaScript library) used by the login page are also included.

```
<security-constraint>
   <web-resource-collection>
      <web-resource-name>allPages</web-resource-name>
      <url-pattern>/*</url-pattern>
   </web-resource-collection>
   <auth-constraint>
      <role-name>valid-users</role-name>
   </auth-constraint>
</security-constraint>
```

If the pages you create use resources, such as images, CSS files, or JavaScript libraries, the default `allPages` security constraint in the `web.xml` file will prevent those resources from loading at runtime. To allow your application to display those resources, you should save the resources in a folder of their own and then edit the `allPages` security constraint so that the resources folder is not contained in the URL pattern.

Note that this resource issue does not apply when you run the ADF Security wizard and choose the **ADF Authentication and Authorization** option (the default). Specifically, in that case, the default generated constraint is on the ADF Authentication servlet and the constraint (`/adfAuthentication`) excludes any resource files.

How to Create a Public Welcome Page

Because web applications are generally secured, there is always a need for a starting point or home page for unauthenticated users. To create this public welcome page, you create an ADF Faces page to act as the entry point for the application, which contains links to other pages within the application. However, only links to public pages should be rendered to unauthenticated users and, conversely, links to secured pages should be rendered only after the user has logged in and has the appropriate privileges to view the target page.

Best Practice:

When the user presses Ctrl-N or Ctrl-T to open a new browser window or tab and no welcome page is defined in the application's `web.xml` file, the browser will display a 403 or 404 error. To prevent this error, you must specify a welcome page definition in the application's `web.xml` file. You can create this definition when you run the Configure ADF Security wizard. For details about running the wizard, see How to Enable ADF Security.

Ensuring That the Welcome Page Is Public

After you have created a regular ADF Faces page, the page will, by default, be public and accessible by unauthenticated users. If, however, you have associated the welcome page with an ADF resource, for example, by dropping databound ADF Faces components into the welcome page using the Data Controls panel, then ADF Security will secure the page by default. You can make any ADF resource publicly accessible using the overview editor for security policies to grant a view privilege on the resource to the provided `anonymous-role`. For details about the `anonymous-role` see, What Happens When You Make an ADF Resource Public.

Adding Login and Logout Links

You can add login and logout links to your public welcome page so that users can explicitly log in and out while they are in the application. While Java EE container-managed security supports the concept of authentication when accessing a secured resource, there is no standard way to log out and stay within a secured application. However, it is a common practice in web applications to allow the user to stay on the same page if that page is public or to return the user to the welcome page if that page is secured. While adding the login and logout links to each page would let the user end their login session anywhere within the application (and return to the welcome page), having these links on the welcome page enables users to explicitly authenticate on entering the application.

For example, you can create an ADF Faces panel group with three components, including an output text area, an image, and login/logout links. To render the appropriate login or logout link, you can use an EL expression that evaluates the user’s authentication status. Specifically, you can use `securityContext.authenticated` to access the ADF security context, as shown in the following example. The expression evaluates to `true` or `false` and, in this example, the result determines which login/logout image and link to display. As the following example shows, the link triggers the `logon` and `logoff` methods of the security bean.

```
<af:panelGroupLayout inlineStyle="width:100%; height:15px;" id="ptpgl3">
     <af:spacer width="7" height="10" id="pts2"/>
     <af:outputText value="Welcome #{securityContext.userName}!"
                           inlineStyle="font-weight:bold; width:100px" id="ptot2"
                           rendered="#{securityContext.authenticated}"/>
     <af:image source="#{securityContext.authenticated ? '/images/lock.gif' : '/
images/key.gif'}"
               id="pti2" inlineStyle="width:16px; height:16px;"
shortDesc="switchable icon"/>
     <af:link text="Logon" id="l1" action="#{securityBean.logon}"
                rendered="#{!securityContext.authenticated}"/>
     <af:link text="Logoff" id="l2" action="#{securityBean.logoff}"
```

```
                rendered="#{securityContext.authenticated}"/>
     <f:facet name="separator">
         <af:spacer width="5" height="10" id="pts1"/>
     </f:facet>
</af:panelGroupLayout>
```

The security bean that implements the logon and logoff methods may use the ADF AuthenticationService API to protect the application from malicious redirects. This achieves the same level of protection as invoking the ADF authentication servlet with null success_url and end_url parameters that the web.xml file whitelists, but is considered a best practice alternative to handling navigation with task flows. For more details about redirects, see How to Redirect a User After Authentication.

As an alternative to rendering the link directly within a page, you can create a login link component with the login and logout links that you can add to a page template, as described in How to Create a Login Link Component and Add it to a Public Web Page for Explicit Authentication.

Hiding Links to Secured Pages

Since an anonymous user should not have access to any secured pages, any navigation component on the welcome page that points to a secured page should be hidden from view based on the following two criteria:

- Is the user authenticated with a known user identity?
- Does the specified user identity have permission to view the target?

If either of these criteria has not been met, the rendered attribute of any navigation component on a public page that points to a secured resource must have its rendered attribute set to false, thus hiding it from the anonymous user. To enforce these rules within your welcome page, see Using Expression Language (EL) with ADF Security.

How to Redirect a User After Authentication

When you have chosen to implement implicit authentication, after the user accesses a secured web page and logs in, the ADF authentication servlet will redirect back to the original page that initiated the login request. With ADF Security authentication enabled, the ADF automatically passes on the session map the original page as the ADF authentication success_url parameter. Typically, this is the desired behavior.

Additionally, when you have chosen to implement implicit authentication, you can protect the Fusion web application from malicious redirects by specifying the success_url parameter as an init-param element within the web.xml file. However, because the ADF authentication servlet automatically passes the original page as the success_url parameter, which supersedes any web.xml setting, in practice the only scenario in which an init-param setting in web.xml takes effect is when the user explicitly types the adfAuthentication URL into the browser.

However, when you choose to implement explicit authentication and display an explicit login link in your page, the ADF Security framework is not able to determine the originating page. In this case, your login link ensures the ADF Authentication servlet redirects to the desired page after authentication by passing the view activity name of the web page as the servlet success_url parameter, as shown in the following example.

```
<af:link text="Login" destination="/adfAuthentication?success_url=/faces/
viewactivityname"/>
```

> **Best Practice:**
>
> For explicit authentication, to ensure that the Fusion web application handles the ADF authentication redirect as an ADF Controller navigation event, specify the redirect target by its view activity name. Specifically, without specifying a view activity as the redirect target, command components, such as the `af:button` component, will always return to the original page after login. Passing the view activity name as the redirect target of the ADF authentication servlet `success_url` and `end_url` parameters ensures the application handles navigation using control flow rules in all cases.

In the case of explicit authentication, following the best practice of defining the `success_url` with a task flow view activity name protects the Fusion web application from malicious redirects by ensuring the ADF authentication servlet ignores any `adfAuthentication` URL that a user may attempt to type into the browser. Alternatively, when a task flow activity is not convenient, your application can use the `AuthenticationService` API to implement login and logout methods that handle the redirect in a secure way. For more information about using the API in a security bean, see What You May Need to Know About Redirecting to a Different Host Server.

In cases where the user is authenticated but not authorized to view a web page, you can redirect the ADF authentication servlet to an error page in your application. Error handling in Fusion web applications is under the control of the ADF Controller exception handler unless you have created an application that does not use a task flow in its design. For example, in an unbounded task flow, where you have defined an unbounded task flow with a top-level welcome page and a browse page (secured through its ADF page definition), you would see an error page from the application, named `authorizationErrorPage.jspx`, specified in the `adfc-config.xml` file, as shown in the following example.

```
<adfc-config xmlns="http://xmlns.oracle.com/adf/controller" version="1.2">
  <exception-handler>authorizationErrorPage</exception-handler>
  <view id="welcomePage">
    <page>/welcomePage.jspx</page>
  </view>
  <view id="browse">
    <page>/browse.jspx</page>
  </view>
  <view id="authorizationErrorPage">
    <page>/authorizationErrorPage.jspx</page>
  </view>
  <control-flow-rule>
    <from-activity-id>welcomePage</from-activity-id>
    <control-flow-case>
      <from-outcome>goToSecuredPage</from-outcome>
      <to-activity-id>browse</to-activity-id>
    </control-flow-case>
  </control-flow-rule>
</adfc-config>
```

For details about how to specify an error page as a view activity for the ADF Controller exception handler, see Handling Exceptions in Task Flows.

In cases where the user is not authenticated and an authorization failure occurs, the framework redirects to the ADF authentication servlet, which in turn triggers a Java EE constraint that prompts for login. In this case, container-managed security relies on the login page and error page that you specify in the `<login-config>` element of the `web.xml` file.

If you create a Fusion web application without utilizing task flows, then you can specify an `<init-param>` setting in `web.xml` for the **ADF binding filter**, as shown in the following example. In this case, when no task flow is present in the application, page authorization checking is handled by the ADF binding filter, and the `unauthorizedErrorPage` parameter will be passed to the ADF binding request handler.

> **Note:**
>
> The `unauthorizedErrorPage` parameter feature is provided for compatibility with previous releases where ADF Controller was not available. In Fusion web applications, when you need to redirect users to an error page, you use the task flow exception handler to specify the error page, as shown in the previous example.

```
<filter>
    <filter-name>adfBindings</filter-name>
    <filter-class>oracle.adf.model.servlet.ADFBindingFilter</filter-class>
    <init-param>
        <param-name>unauthorizedErrorPage</param-name>
        <param-value>faces/authorizationErrorPage.jspx</param-value>
    </init-param>
</filter>
```

How to Trigger a Custom Login Page Specifically for Implicit Authentication

When you have chosen to implement implicit authentication to allow users to access protected resources (which may be by direct URL access or by navigating to an ADF Security protected resource), the ADF Security authentication servlet initiates the Java EE container to display the login form configured in the `web.xml` file. For implicit authentication, when you have customized the default, wizard-generated login form to use ADF Faces components, you must modify the URL pattern for the servlet mapping to reference the ADF Faces servlet.

You can accomplish this in the Authentication Type page of the Configure ADF Security wizard when you configure ADF Security, or in the `web.xml` file directly. If you have already run the Configure ADF Security wizard, you can use the following procedure to confirm that the `web.xml` file has been updated as described.

When you have chosen to implement explicit authentication and have created a public page to allow users to login to access protected resources, the login form you create will not be triggered by ADF Security. In the explicit authentication scenario, you do need to configure the `web.xml` file.

> **Note:**
>
> Configuring the `web.xml` file with the login page application path when your application implements explicit authentication programmatically, as described in Creating an ADF Faces-Based Login Page Specifically for Explicit Authentication, may produce unpredictable results and login may fail. Only modify the `web.xml` file when you are implementing implicit authentication and you have created a form that relies on the `j_security_check` action, as does the default, wizard-generated login form.

Before you begin:

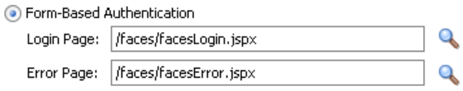
It may be helpful to have an understanding of the login page. For more information, see How to Create a Login Page Specifically for Explicit Authentication.

To reference an ADF Faces login page for implicit authentication:

1. In the Applications window, expand **WEB-INF** and double-click **web.xml**.
2. In the overview editor for the `web.xml` file, click the **Security** navigation tab.
3. In the Security page, expand the **Login Authentication** section, and set the login page to include a reference to the ADF Faces servlet such that the login page can be part of the ADF Faces lifecycle `/faces/ADFlogin.jspx` page.

   When you add a page using the file browser, the path entered in the `web.xml` file will not specify `/faces`. Modify the entry so that the path references the servlet mapping path for the ADF Faces servlet. For example, if the URL pattern specified by the mapping is `/faces/*`, then your path should look like `/faces/`*`yourpage`*`.jspx`, as shown in Figure 47-26.

   Figure 47-26 Adding a Reference to the Faces Servlet in the Login Configuration

   ● Form-Based Authentication
   Login Page: /faces/facesLogin.jspx
   Error Page: /faces/facesError.jspx

How to Ensure That the Redirect Destination Page is Available

When the ADF authentication servlet directs the user to login, it puts the redirect destination defined by the `success_url` on the Session map. However, in the case where the session times out before the user has authenticated, the Session map looses the value of `success_url` and the application looses the ability to navigate to the destination page.

To ensure the login redirect destination is always available to the application, you can define the `success_url` parameter as a task flow navigation outcome rule or you can define it statically when you run the Configure ADF Security wizard as a single whitelisted page that will appear in the `web.xml` file.

Because the `web.xml` file must be defined prior to deployment, the Fusion web application supports an alternative, dynamic approach to configuring the redirect URL `success_url` parameter that involves defining connection reference values. These values may also be configured in `connections.xml` file directly.

To use the `connections.xml` file as a persistence store for the ADF authentication servlet redirect destination:

1. In the `web.xml` file, add the `init-param` named `oracle.adf.share.security.authentication.connections` to the ADF authentication servlet definition and specify the `param_value` as the connection reference name that appears in the `connections.xml` file, located in the `adf/META-INF` folder, as the following example shows.

```
<servlet>
  <servlet-name>adfAuthentication</servlet-name>
    <servlet-class>
      oracle.adf.share.security.authentication.AuthenticationServlet
    </servlet-class>
    <init-param>
      <param-name>
        oracle.adf.share.security.authentication.connections
      </param-name>
      <param-value>my_connection_name</param-value>
    </init-param>
   ...
</servlet>
```

2. In the `connections.xml` file, define the `success_url` and, optionally, `end_url` in the named connection reference, as the following example shows.

```
<References xmlns="http://xmlns.oracle.com/adf/jndi">
  <Reference name="my_connection_name"
             className=

"oracle.adf.share.security.authentication.AuthenticationConnection"
             xmlns="">
    <Factory className=

"oracle.adf.share.security.authentication.AuthenticationConnection"/>
    <RefAddresses>
      <StringRefAddr addrType="success_url">
        <Contents>
           http://localhost:port/context_root/faces/welcomePage</Contents>
      </StringRefAddr>
      <!-- end_url for logout -->
      <StringRefAddr addrType="end_url">
        <Contents>
           http://localhost:port/context_root/faces/publicPage</Contents>
      </StringRefAddr>
    </RefAddresses>
  </Reference>
</References>
```

Note that the destination URL may omit the host name, port number, and context root, when redirecting the application to the current request host, port, and context root. In this case, the URL is specified as `/faces/pagename`, where `/faces` loads the page in the context of ADF Faces and supports pages designed with ADF Faces components.

At runtime, if the redirect URL is not on Session map, the ADF authentication servlet gets the connection name from `init-param` named `oracle.adf.share.security.authentication.connections` to lookup the connection from `connections.xml` file. If no connection is named or if `success_url` (login) or end_url (logout) are not in connections.xml, then the ADF authentication servlet will check if success_url or end_url is defined in the web.xml.

What You May Need to Know About Redirecting to a Different Host Server

During logout it may be necessary to redirect the user to an arbitrary web page or host server that differs from the originating server of the Fusion web application. In this scenario, the application should be protected from a malicious redirect by ensuring the destination page URL is not placed by the application on the Request and will instead use the Session map. To accomplish this, a best practice is to block passing ADF redirect parameters on the browser URL by adding the servlet init-param element disable_url_param set to true to the ADF authentication servlet definition in the web.xml file as this example shows.

```
<servlet>
   <servlet-name>adfAuthentication</servlet-name>
   <servlet-class>
        oracle.adf.share.security.authentication.AuthenticationServlet
   </servlet-class>
   <init-param>
      <param-name>success_url</param-name>
      <param-value>faces/welcome</param-value>
   </init-param>
   <init-param>
      <param-name>end_url</param-name>
      <param-value>faces/welcome</param-value>
   </init-param>
   <init-param>
      <param-name>disable_url_param</param-name>
      <param-value>true</param-value>
   </init-param>
   <load-on-startup>1</load-on-startup>
</servlet>
...
```

Legacy applications that were built before ADF enforced passing the redirect parameters on the Session map that would otherwise expose the parameters on the browser URL Request should not set the disable_url_param to true and should instead consider implementing a security bean using the ADF interface oracle.adf.share.security.AuthenticationService to handles ADF-specific login and logout methods by placing the target destination on the Session map.

The following example illustrates how to use the logout() method of the AuthenticationService interface. This method will internally put the passed in URL on the Session map and will ignore the end_url parameter that might explicitly be set on the URL by a user. Similarly, the login() method places the login URL destination on the Session map. When you need logout to redirect to a web page on a different server, your method implementation may pass the URL of the desired web page and still be protected from a malicious redirect.

```
import javax.faces.context.ExternalContext;
import javax.faces.context.FacesContext;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import oracle.adf.share.security.AuthenticationService;
import oracle.adf.share.security.authentication.AuthenticationServiceUtil;

public class SecurityBean {
    public SecurityBean() {
        super();
    }

    public boolean isAuthenticated() {
        ExternalContext ectx =
                          FacesContext.getCurrentInstance().getExternalContext();
        return ectx.getUserPrincipal() != null;
    }

    public void logon() {
        if (!isAuthenticated())
        {
            FacesContext ctx = FacesContext.getCurrentInstance();
            AuthenticationServiceUtil.getAuthenticationService().login("/faces" +
                                        ctx.getViewRoot().getViewId(), null,
null);
        }
    }

    public void logoff() {
        if (isAuthenticated())
        {
            AuthenticationServiceUtil.getAuthenticationService().
                                                  logout("/faces/index", null);
        }
    }
}
```

An alternative to the programmatic approach handles placing the destination URL on the Session map by invoking the ADF authentication servlet in an explicit link with null value success_url and end_url parameters. In this case, the ADF authentication servlet will try to obtain the redirect values from the web.xml file (or the connections.xml file) and will ignore any values that a user might attempt to enter on the URL. This approach requires configuring the ADF authentication servlet definition in the web.xml by specifying an init-param element for the success_url and end_url parameters. In effect, this is a declarative approach that results in a whitelist with a single destination for login and logout redirects.

What You May Need to Know About Fusion Web Application Logout and Browser Caching

When basic type authentication is in effect as specified in the Fusion web application's web.xml file, the browser caches authentication credentials. This is a known issue with basic authentication that re-authenticates users after the logout redirect is performed by the ADF authentication servlet. In this scenario, in order to complete the logout session and prevent users from accessing resources, it is necessary to close the browser and restart a new browser session.

To ensure the ADF application completes logout and prevents a user from being able to access resources after logout, use form-based authentication instead of basic authentication. You can select form-based authentication when you run the Configure ADF Security wizard, as described in How to Enable ADF Security.

What You May Need to Know About Displaying Error Pages in Internet Explorer

When you enable ADF Security to display a custom error page, any HTTP errors that the application generates in response to a request to view a page, should result in the error page being displayed. However, in certain versions of Internet Explorer, the browser will display either the custom error page or an error message built into Internet Explorer. If Internet Explorer displays only an error message you can disable the option to display error message numbers in the Advanced settings of the Internet Options dialog. In Internet Explorer 10, this option is enabled by default. You should disable **Show friendly HTTP error messages** when you want to display a custom error page that you enabled with the Configure ADF Security wizard.

To ensure custom error pages display in Internet Explorer 10:

1. In Internet Explorer toolbar, click the **Tools** button.
2. In the Internet Options dialog, click the **Advanced** tab.
3. In the Advanced tab, scroll and locate **Show friendly HTTP error messages**.
4. Click the option to disable it and click **OK**.

Testing Security in JDeveloper

For testing ADF Security, JDeveloper lets you run the application using Integrated WebLogic Server and decides whether to migrate security objects defined in your application.

Integrated WebLogic Server enables you to run the application directly within JDeveloper and determine whether or not to migrate security objects, including the application policies, users, and credentials that your application defines. By default, all security objects are migrated to Integrated WebLogic Server each time you run the application.

How to Configure, Deploy, and Run a Secure Application in JDeveloper

JDeveloper is configured by default to deploy the security objects from your application repositories to Integrated WebLogic Server each time you run the application. You can change this behavior by selecting security deployment options in the Application Properties dialog to:

- Decide whether to overwrite the domain-level policies with those from the application `jazn-data.xml` file
- Decide whether to overwrite the system credentials from the application's `cwallet.sso` file

  The `cwallet.sso` file (located in JDeveloper in the Application Resources panel of the Applications window under the **Descriptors-META-INF** node) stores credentials as securely kept objects that are presented to the authentication provider to be matched against identities. The file is encrypted and cannot be browsed or edited within JDeveloper. At design-time, different components make use of `cwallet.sso` file and are responsible for creating the necessary credentials in it.

- Decide whether to migrate the identity store portion of the `jazn-data.xml` file to the domain-level identity store

If you make no changes to the deployment settings, each time you run the application, JDeveloper will overwrite the domain-level security policies and system credentials. Additionally, JDeveloper will migrate new user identities you create for test purposes and update existing user passwords in the embedded LDAP server that Integrated WebLogic Server uses for its identity store. However, if you prefer to run the application without updating the existing security objects in Integrated WebLogic Server, you have this option.

Before you begin:

It may be helpful to have an understanding of using Integrated WebLogic Server. For more information, see Testing Security in JDeveloper.

To configure security deployment and run the application in JDeveloper:

1. In the main menu, choose **Application** and then **Secure > Configure Security Deployment**.
2. In the Application Properties dialog, in the Deployment page, in the **Security Deployment Options** section, select the security objects that you want to deploy to Integrated WebLogic Server.

   By default, each time you run the application, JDeveloper will overwrite the application policies and system credentials at the domain level with those from the application. If you prefer not to overwrite either of these repositories, deselect **Application Policies** or **Credentials**. When deselected, JDeveloper will merge only new polices or credentials into the domain-level stores.

   By default, each time you run the application, JDeveloper will migrate new user identities you create for test purposes and update existing user passwords in the embedded LDAP server that Integrated WebLogic Server uses for its identity store. You can disable migration of the application identity store by deselecting **Users and Groups**.
3. Click **OK**.
4. In the Applications window, right-click the user interface project that contains the secured web pages and choose **Run**.

   When you choose **Run** on the user interface project, JDeveloper will run the application using the default run target you configured for the project. For example, you can configure a task flow activity as the run target to start your application. To configure the default run target, see Testing Task Flows.

   The Create Default Domain dialog appears the first time you run the application and start a new domain in Integrated WebLogic Server. Use the dialog to define an administrator password for the new domain. Passwords you enter can be eight characters or more and must have a numeric character.

What Happens When You Configure Security Deployment Options

When you run the application using Integrated WebLogic Server, JDeveloper migrates the security policies and credentials to the domain level based on security deployment configuration settings specified in the Application Properties dialog. During the deployment process, JDeveloper updates the weblogic-application.xml file that it adds to the deployment archive file with the Application Properties settings, as shown in the following example. Note that these settings are not added to the weblogic-application.xml file in the application source directory and thus are not visible.

```
<application-param>
    <param-name>jps.credstore.migration</param-name>
    <param-value>OVERWRITE</param-value>
</application-param>
<application-param>
    <param-name>jps.policystore.migration</param-name>
    <param-value>OVERWRITE</param-value>
</application-param>
```

The OVERWRITE value allows you to modify the security policies and credentials in your application and redeploy either to Oracle WebLogic Server running in development mode or to Integrated WebLogic Server (set up to run in development mode by default).

> **Note:**
>
> When you eventually deploy to a production environment, the migration settings in the weblogic-application.xml file are ignored; it would be considered a security vulnerability to allow existing policies and credentials to be overwritten. For information about deploying to a production environment, see Preparing the Secure Application for Deployment.

JDeveloper also updates the weblogic-application.xml file with OPSS lifecycle listeners, as shown in the following example. To initiate the migration process before the application runs, the lifecycle listeners observe the migration settings for policies and credentials and overwrite the security objects at the domain level.

```
<listener>
    <listener-class>
        oracle.security.jps.wls.listeners.JpsApplicationLifecycleListener
    </listener-class>
</listener>
<listener>
    <listener-class>
        oracle.security.jps.wls.listeners.JpsAppVersionLifecycleListener
    </listener-class>
</listener>
```

During the migration process, JDeveloper maps the Oracle Platform Security Services (OPSS) application role member classes to the Integrated WebLogic Server member classes and migrates the users to WebLogic Server identity store users and migrates the roles to WebLogic Server identity store groups. In Oracle WebLogic Server, users is an implicit group equivalent to OPSS authenticated-role.

Identity store migration is not controlled by the application lifecycle listener settings in the weblogic-application.xml file. Instead, an Oracle WebLogic Mbean handles migrating the identities when running in Integrated WebLogic Server or when deploying from JDeveloper. If the user already exists, the Mbean will not migrate the entire user definition. Only the user password will be updated.

How to Use the Built-In test-all Application Role

When you run the Configure ADF Security wizard, you can enable the option to add the `test-all` application role to the policy store in the `jazn-data.xml` file. When you enable this option, you also specify the scope of grants to the application role for your application:

- Select **Grant to Existing Objects Only** when you want JDeveloper to grant view rights to the `test-all` application role and you want this policy to apply to all the ADF task flows and web pages that appear in your user interface project at the time you run the wizard.
- Select **Grant to All Objects** when you want JDeveloper to grant view rights to the `test-all` application role and you want this policy to apply to all existing and future ADF task flows and web pages that developers will create in the user interface project. Note that the wizard displays the option **Grant to New Objects** after you run the wizard the first time with the **Grant to All Objects** option selected.

After you run the wizard, the `test-all` role appears in the `jazn-data.xml` file and is visible in the overview editor for security policies. You will not need to populate the `test-all` role with test users since the wizard assigns the built-in application role `anonymous-role` to the `test-all` role. In this case, all users will automatically have the `anonymous-role` principal and will be permitted to access the application.

> **Note:**
>
> Before you deploy the application, you must remove all occurrences of the `test-all` role from the policy store, as described in How to Remove the test-all Role from the Application Policy Store. This will prevent unauthorized users from accessing the web pages of your application.

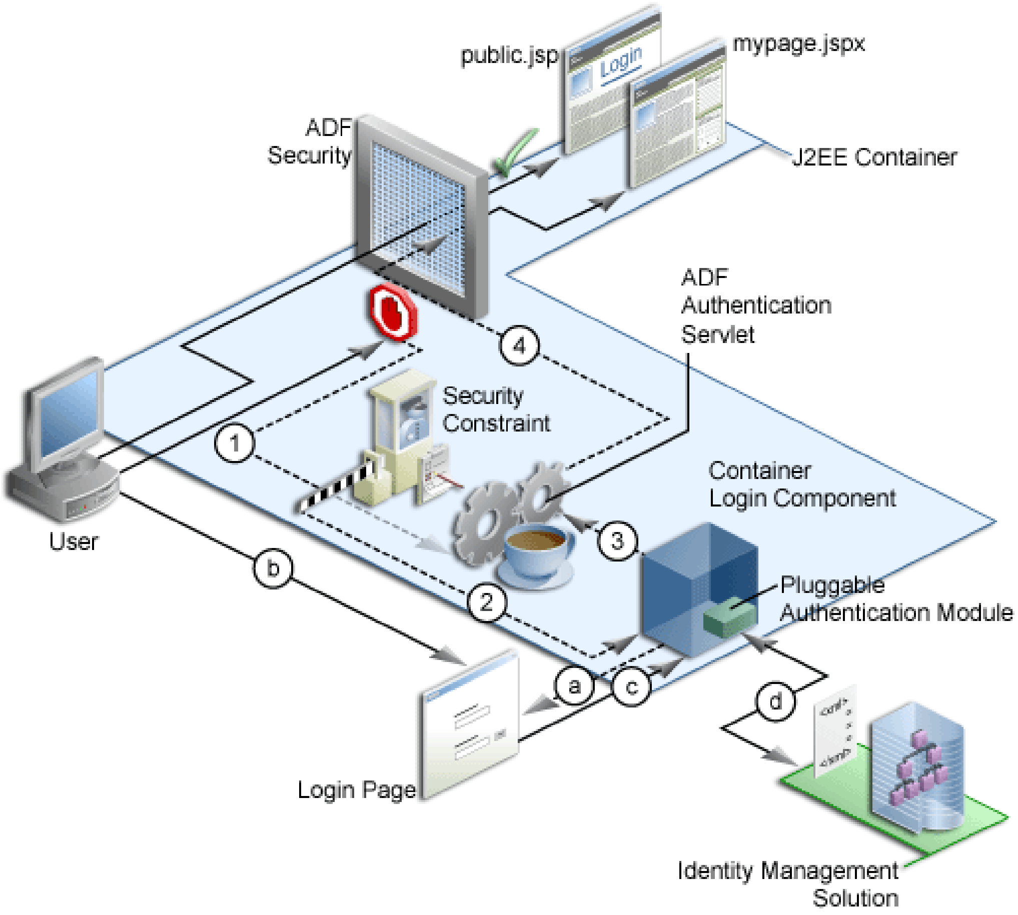
You can rerun the wizard and disable automatic grants at any time. Once disabled, new ADF task flows and web pages that you create will not utilize the `test-all` role and will therefore require that you define explicit grants, as described in Defining ADF Security Policies.

What Happens at Runtime: How ADF Security Handles Authentication

When you test the application in JDeveloper using Integrated WebLogic Server, the identity store is migrated to the embedded LDAP server, with information stored in Oracle Internet Directory.

Figure 47-27 illustrates the authentication process when users attempt to access an ADF bounded task flow or any web page containing ADF bindings (such as `mypage.jspx`) without first logging in. Authentication is initiated implicitly because the user does not begin login by clicking a login link on a public page. In the case of the secured page, no grants have been made to the anonymous user.

Figure 47-27 ADF Security Implicit Authentication



In Figure 47-27, the implicit authentication process assumes that the resource does not have a grant to `anonymous-role`, that the user is not already authenticated, and that the authentication method is Form-based authentication. In this case, the process is as follows:

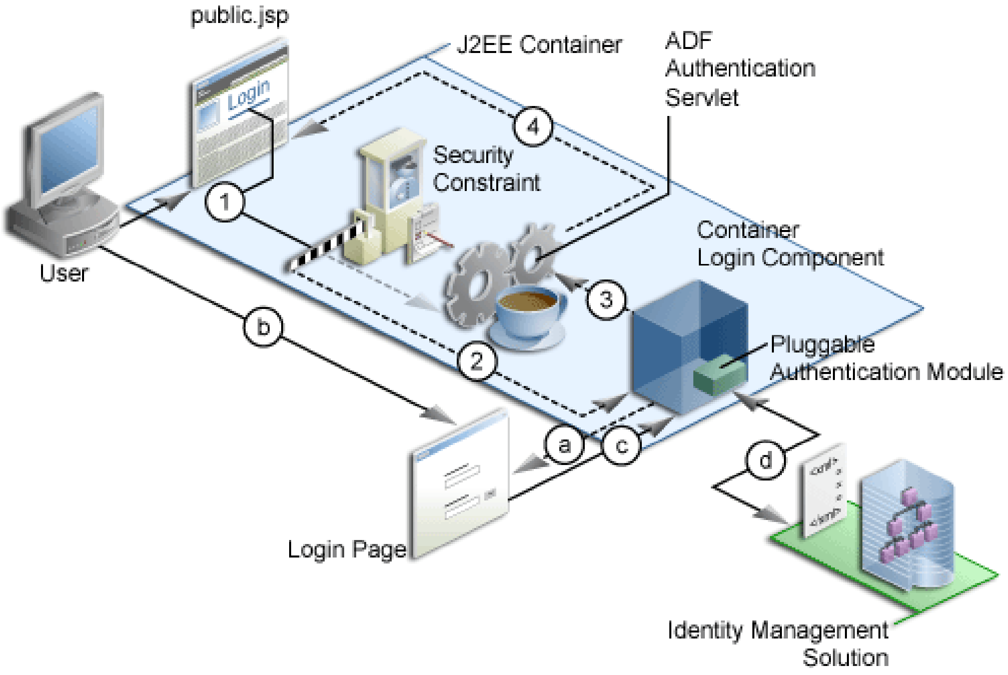
1. When the bounded task flow or web page (with ADF bindings) is requested, the ADF bindings servlet filter redirects the request to the ADF authentication servlet (in the figure, Step 1), storing the logical operation that triggered the login.
2. The ADF authentication servlet has a Java EE security constraint set on it, which results in the Java EE container invoking the configured login mechanism (in the figure, Step 2). Based on the container's login configuration, the user is prompted to authenticate:
   a. The appropriate login form is displayed for form-based authentication (in the figure, Step 2a).
   b. The user enters their credentials in the displayed login form (in the figure, Step 2b).
   c. The user posts the form back to the container's `j_security_check()` method (in the figure, Step 2c).
   d. The Java EE container authenticates the user, using the configured pluggable authentication module (in the figure, Step 2d).

3. Upon successful authentication, the container redirects the user back to the servlet that initiated the authentication challenge, in this case, the ADF authentication servlet (in the figure, Step 3).
4. On returning to the ADF authentication servlet, the servlet subsequently redirects to the originally requested resource (in the figure, Step 4).

   Whether or not the resource is displayed will depend on the user's access rights and on whether authorization for ADF Security is enforced, as explained in What Happens at Runtime: How ADF Security Handles Authorization.

Figure 47-28 illustrates the explicit authentication process when the user becomes authenticated starting with the login link on a public page.

Figure 47-28 ADF Security Explicit Authentication



In an explicit authentication scenario, an unauthenticated user (with only the `anonymous` user principal and `anonymous-role` principal) clicks the **Login** link on a public page (in the figure, Step 1). The login link is a direct request to the ADF authentication servlet, which is secured through a Java EE security constraint in the `web.xml` file.

In this scenario, the current page is passed as a parameter to the ADF authentication servlet. As with the implicit case, the security constraint redirects the user to the login page (in the figure, Step 2). After the container authenticates the user, as described in Step a through Step d in the implicit authentication case, the request is returned to the ADF authentication servlet (in the figure, Step 3), which subsequently returns the user to the public page, but now with new user and role principals in place.

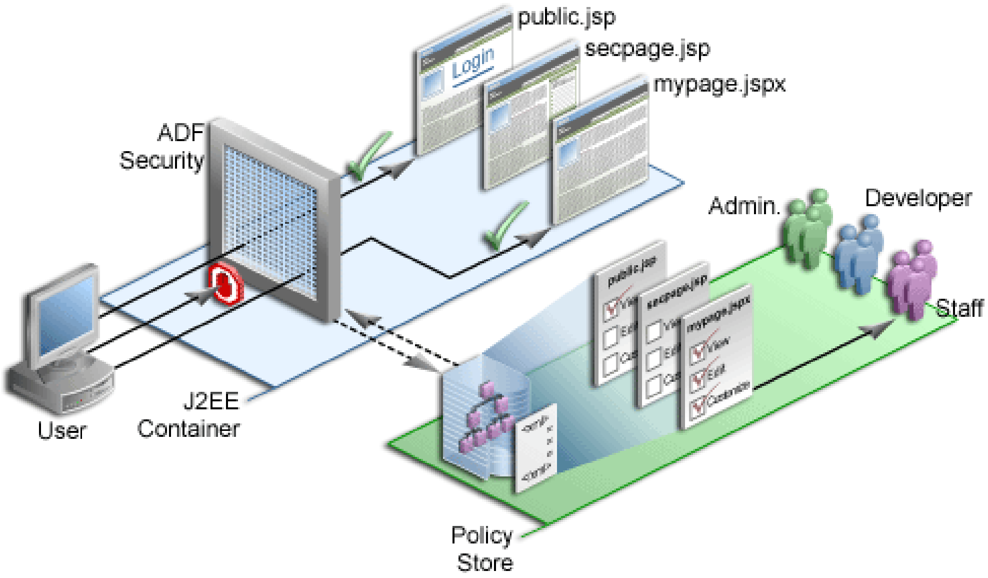
What Happens at Runtime: How ADF Security Handles Authorization

When ADF authorization is enabled, the ADF bounded task flows and web pages outside of a task flow that have an ADF page definition will be secure by default.

When a user attempts to access these web pages, ADF Security checks to determine whether the user has been granted access in the policy store. If the user is not yet authenticated, and the page is not granted to the `anonymous-role`, then the application displays the login page or form. If the user has been authenticated, but does not have permission, a security error is displayed. If you do not configure the policy store with appropriate grants, the pages will remain protected and therefore stay unavailable to the authenticated user.

Figure 47-29 illustrates the authorization process.

Figure 47-29 ADF Security Authorization



The user is a member of the application role *staff* defined in the policy store. Because the user has not yet logged in, the security context does not have a subject (a container object that represents the user). Instead, Oracle Platform Security Services provides ADF Security with a subject with the `anonymous` user principal (a unique definition of the user) and the `anonymous-role` principal.

With the `anonymous-role` principal, typically the user would be able to access only pages not defined by ADF resources, such as the `public.jsp` page, whereas all pages that are defined either by an ADF task flow or outside of a task flow using an ADF page definition file are secure by default and unavailable to the user. An exception to this security policy would be if you were to grant `anonymous-role` access to ADF resources in the policy store. In this case, the user would not be allowed immediate access to the page defined by an ADF resource.

When the user tries to access a web page defined by an ADF resource, such as `mypage.jspx` (which is specified by an ADF page definition, for example), the ADF Security enforcement logic intercepts the request and because all ADF resources are secured by default, the user is automatically challenged to authenticate (assuming that the `anonymous-role` is not granted access to the ADF resource).

After successful authentication, the user will have a specific subject. The security enforcement logic now checks the policy store to determine which role is allowed to view `mypage.jspx` and whether the user is a member of that role. In this example for mypage.jspx, the view privilege has been granted to the staff role and because the user is a member of this role, they are allowed to navigate to mypage.jspx.

Similarly, when the user tries to access secpage.jsp, another page defined by ADF resources, for which the user does not have the necessary view privilege, access is denied.

Users and roles are those already defined in the identity store of the resource provider. Application roles are defined in the policy store of the jazn-data.xml file.

Preparing the Secure Application for Deployment

Oracle ADF applications should be deployed to a target application server after testing. You can run an application in the Integrated WebLogic Server as well as deploy it to a standalone Oracle WebLogic Server

After testing in JDeveloper using Integrated WebLogic Server, you will eventually want to deploy the application to a standalone server. Initially, the server you target will be your staging environment where you can continue development testing using that server's identity store before deploying to the production environment. Thus, you will typically not migrate the test users you created to run with Integrated WebLogic Server. The steps you perform to migrate credentials (in the cwallet.sso file) and security policies (in the jazn-data.xml file) to standalone Oracle WebLogic Server will depend on the configured mode of the target server and whether you deploy using JDeveloper or a tool outside of JDeveloper.

> **Note:**
>
> For details about deploying from JDeveloper to a development environment, see Deploying Fusion Web Applications.

When the target server is configured for **development mode**, you can deploy directly from JDeveloper. In this case, JDeveloper automatically handles the migration of the policy store, system credentials, and identity store (users and groups) as part of the deployment process. Application security deployment properties are configured by default to allow the deployment process to overwrite the domain-level policy store and the system credentials. Additionally, the identity store deployment property is configured by default to migrate the identity store consisting of your test users. You can change this default deployment behavior in the Application Properties dialog, as described in How to Configure, Deploy, and Run a Secure Application in JDeveloper.

> **Note:**
>
> Note that migration of system credentials to Oracle WebLogic Server running in development mode will be performed only if the target server is configured to permit credential overwrite. For details about configuring Oracle WebLogic Server to support overwriting of system credentials, see the Configuring the OPSS Security Store chapter in *Securing Applications with Oracle Platform Security Services*.

When the target server is configured for **production mode**, you typically handle the migration task outside of JDeveloper using tools like Oracle Enterprise Manager. For details about using tools outside of JDeveloper to migrate the policy store to the domain-level in a production environment, see the Configuring the OPSS Security Store chapter in *Securing Applications with Oracle Platform Security Services*. Note that Oracle WebLogic Server running in production mode does not support the overwriting of system credentials under any circumstances.

Before you deploy the application, you will want to remove the `test-all` application role if you enabled the automatic grants feature in the Configure ADF Security wizard. Because the `test-all` role makes all ADF resources public, its presence increases the risk that your application may leave some resources unprotected. You must therefore remove the role before you migrate application-level policy store.

Additionally, when you prepare to deploy the application to Oracle WebLogic Server, you will want to remove the test identities that you created in the `jazn-data.xml` file. This will ensure that users you created to test security policies are not migrated to the domain-level identity store.

> **Best Practice:**
>
> If you deploy your application to the standalone environment, you must not migrate users and enterprise roles in your local identity store that are already configured for Oracle WebLogic Server. For example, if you were to deploy the identity store with the user `weblogic` and enterprise role `Administrators`, you would overwrite the default administration configuration on the target server. To ensure you avoid all possible conflicts, you can disable migration of the identity store, as described in How to Remove Test Users from the Application Identity Store.

How to Remove the test-all Role from the Application Policy Store

The overview editor for security policies provides the facility to display all resources with view grants made to ADF Security's built-in `test-all` role. You can use this feature in the overview editor to delete the `test-all` role grant and replace it with a grant to the roles that your application defines.

Alternatively, you could delete the `test-all` role using the overview editor for the `jazn-data.xml` file, by selecting the `test-all` role in the Application Roles page of the editor and clicking the **Delete Application Role** button. However, when you remove the `test-all` role this way, you will still need to create a grant to replace the ones that you delete. Because the overview editor lets you combine both of these tasks, the following procedure describes its usage.

Before you begin:

It may be helpful to have an understanding of the Oracle WebLogic Server. For more information, see Preparing the Secure Application for Deployment.

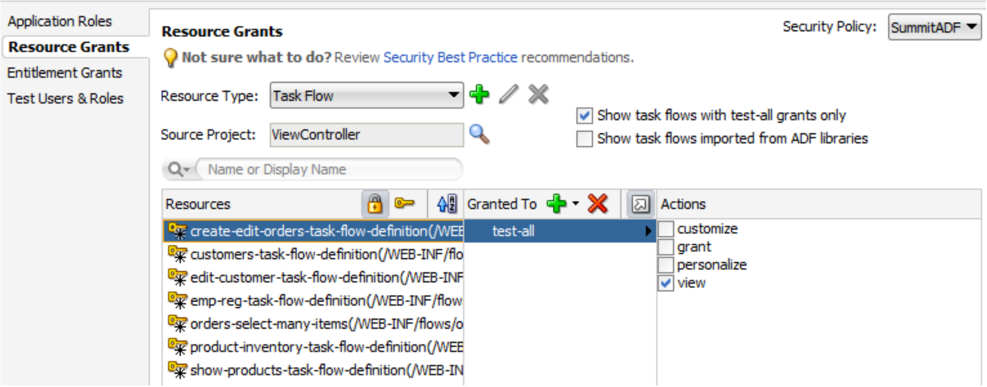
To remove the test-all application role and substitute custom application roles:

1. In the main menu, choose **Application** and then **Secure > Resource Grants**.

2. In the Resource Grants page of the overview editor for security policies, select **Task Flow** from the **Resource Type** dropdown list and then select the **Show task flows with test-all grants only** checkbox to view the list of task flows with grants to this built-in role.

   If no grant exists for the `test-all` role, then the **Resources** list in the overview editor will appear empty. The `test-all` role is defined only when enabled in the Configure ADF Security wizard. If it is enabled, you will see those task flows with `test-all` grants listed, as shown in Figure 47-30.

Figure 47-30 Showing Task Flows with test-all Grants in the Overview Editor



Application Roles
Resource Grants
Entitlement Grants
Test Users & Roles

Resource Grants

Security Policy: SummitADF

Not sure what to do? Review Security Best Practice recommendations.

Resource Type: Task Flow

Source Project: ViewController

☑ Show task flows with test-all grants only
☐ Show task flows imported from ADF libraries

Name or Display Name

Resources	Granted To	Actions
create-edit-orders-task-flow-definition(/WEB	test-all	☐ customize ☐ grant ☐ personalize ☑ view
customers-task-flow-definition(/WEB-INF/flo		
edit-customer-task-flow-definition(/WEB-INF		
emp-reg-task-flow-definition(/WEB-INF/flow		
orders-select-many-items(/WEB-INF/flows/o		
product-inventory-task-flow-definition(/WEB		
show-products-task-flow-definition(/WEB-IN		

3. In the **Resources** column, select the first task flow in the list.
4. In the **Granted to** column, select **test-all** and click the **Remove Grantee** icon.
5. In the **Granted to** column, click the **Add Grantee** icon and choose **Add Application Role** and then use the Select Application Roles dialog to add the desired role.
6. Repeat these steps to remove the `test-all` role and substitute your own application role for all remaining task flows.
7. In the Resource Grants page of the overview editor, select **Web Page** from the **Resource Type** dropdown list and repeat these steps to remove the `test-all` role for all web pages and their ADF page definitions.
8. With the **Show task flows/web pages with test-all grants only** checkbox selected, verify that the overview editor displays no resources with `test-all` grants.

How to Remove Test Users from the Application Identity Store

The standalone Oracle WebLogic Server that you will deploy to will have its own identity stored already configured. To ensure that you do not migrate test users and enterprise role groups you created in JDeveloper to the domain level, you should remove the test user realm from the `jazn-data.xml` file.

Alternatively, if you are deploying from JDeveloper, you can disable the migration of users and groups by deselecting the **Users and Groups** option in the Application Properties dialog, as described in How to Configure, Deploy, and Run a Secure Application in JDeveloper.

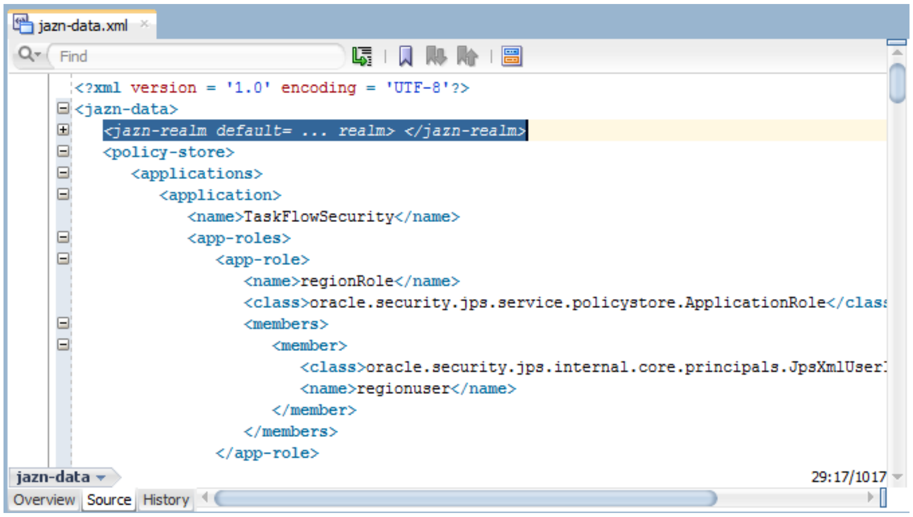
Before you begin:

It may be helpful to have an understanding of the Oracle WebLogic Server. For more information, see Preparing the Secure Application for Deployment.

To remove test users and enterprise role groups from the identity store:

1. In the main menu, choose **Application** and then **Secure > Users**.
2. In the editor window for the `jazn-data.xml` file, click the **Source** tab.
3. In the source for the `jazn-data.xml` file, click the **-** icon next to the **<jazn-realm>** element so the entire element appears collapsed as shown in Figure 47-31.

Figure 47-31 Selecting the <jazn-realm> Element in the XML Editor



```
<?xml version = '1.0' encoding = 'UTF-8'?>
<jazn-data>
    <jazn-realm default= ... realm> </jazn-realm>
    <policy-store>
        <applications>
            <application>
                <name>TaskFlowSecurity</name>
                <app-roles>
                    <app-role>
                        <name>regionRole</name>
                        <class>oracle.security.jps.service.policystore.ApplicationRole</class
                        <members>
                            <member>
                                <class>oracle.security.jps.internal.core.principals.JpsXmlUser]
                                <name>regionuser</name>
                            </member>
                        </members>
                    </app-role>
```

4. With the element selected, press **Delete** and save the file.

How to Secure Resource Files Using a URL Constraint

Resource files, including images, style sheets, and JavaScript libraries are files that the Fusion web application loads to support the individual pages of the application. ADF Security does not secure these files. Although securing such files is not a requirement for securing the web pages of the application, in some cases it may be desirable to protect resource files to fully harden your application. Please note that JavaScript files are downloaded to the browser and, as such, can be read at that time.

To serve up resource files, the ADF Faces framework relies on `org.apache.myfaces.trinidad.webapp.ResourceServlet`, which delegates to a resource loader. The Fusion web application's `web.xml` file contains a servlet mapping that maps the resource servlet to URL patterns. By default, JDeveloper uses the patterns `/adf/*` for MyFaces Trinidad Core, and `/afr/*` for ADF Faces.

Because ADF Security does not protect the resource servlet, you can protect the Java EE access paths for the resource servlet with a security constraint. To implement your own security for the resource files in the `web.xml` file, define a security constraint for the `/adflib/*`, `/adf/*` and `/afr/*` paths and assign a specific security role to them. This security constraint will require all users to be authenticated before they can access the first page of the Fusion web application.

The following example shows the security role `resource_role` and the constraint with the URL patterns mapped to this role. After you define the security role, the administrator for the target server must map this role to an Oracle WebLogic Server enterprise role (user group) or create an enterprise role with the same name (in which case no mapping is required).

```
<security-role>
    <description>Java EE role to map to an enterprise role. All users who will
      be allowed to run Fusion web app must be members of that role.     </
description>
    <role-name>resource_role</role-name>
</security-role>

<security-constraint>
    <web-resource-collection>
      <web-resource-name>resources</web-resource-name>
      <url-pattern>/adflib/*</url-pattern>
      <url-pattern>/adf/*</url-pattern>
      <url-pattern>/afr/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
      <role-name>resource_role</role-name>
    </auth-constraint>
</security-constraint>
```

Disabling ADF Security

You can disable ADF Security but that does not delete the created permission. This option can be used to run a previously secured application unsecured.

JDeveloper allows you to disable ADF Security when you want to temporarily run the application without enforcing authorization checks against the application policy store. This will allow you to run the application and access all resources without the protection provided by existing security policies.

How to Disable ADF Security

To disable ADF Security at the level of your application, run the wizard and choose one of these options:

- **ADF Authentication** disables ADF authorization but leaves the ADF authentication servlet enabled. For example, you may want to run your application in JDeveloper with authorization checking against security policies temporarily disabled. This option will require the user to log in the first time a page in the application is accessed by mapping the Java EE application root "/" to the `allPages` Java EE security constraint that will trigger user authentication. ADF resources will not be security-aware because authorization checking is not enforced. Thus, once the user is logged in, all web pages containing ADF resources will be available to the user.
- **Remove ADF Security Configuration** disables the ADF authentication servlet and disables authorization checking on ADF resources. In this case, you can run the application with no user authentication and no security for ADF resources in place.

You may select either option with the intention of reenabling ADF Security at any time. The wizard specifically does not alter the application policy store that contains the security policies that application developers defined for ADF resources. This means that you can return to the wizard at any time, select the **ADF Authentication and**

Authorization option, and reenable ADF Security against your application's existing policy store and identity store.

Before you begin:

It may be helpful to have an understanding of disabling ADF Security. For more information, see Disabling ADF Security.

To disable ADF authorization checking:

1. In the main menu, choose **Application** and then **Secure > Configure ADF Security**.
2. In the ADF Security page, select either the **ADF Authentication** option or the **Disable ADF Security Configuration** option. Click **Next**.

   After you run the wizard with either of these options, the ADF resources of your user interface projects will no longer be security-aware.
3. Click **Finish**.

What Happens When You Disable ADF Security

If you run the Configure ADF Security wizard with the **Remove ADF Security Configuration** option selected, it removes the ADF-specific metadata in the `web.xml` file and `adf-config.xml` file, as described in Table 47-2.

Similarly, running the wizard with the **ADF Authentication** option selected to disable only authorization checking performs the following updates:

- Leaves the ADF-specific metadata in the `web.xml` file unchanged and adds the `allPages` security constraint.

  With the `allPages` security constraint present, users will be expected to authenticate when they first access the application.
- Sets the `authorizationEnforce` parameter in the `<JaasSecurityContext>` element of the `adf-config.xml` file to `false`, as shown in the following example.

```
<JaasSecurityContext

initialContextFactoryClass="oracle.adf.share.security.JAASInitialContextFactory"
  jaasProviderClass="oracle.adf.share.security.providers.jps.JpsSecurityContext"
                             authorizationEnforce="false"
                             authenticationRequire="true"/>
```

The `adf-config.xml` file is located in the `/.adf/META_INF` folder of your workspace. In JDeveloper, you can locate the file in the Application Resources panel of the Applications window by expanding the **Descriptors-ADF META-INF** node. Note that if you view the `adf-config.xml` file in an editor outside of JDeveloper, you must save the workspace to see the wizard-applied changes in the file.

Advanced Topics and Best Practices

There are additional security configurations that go beyond the default ADF Region and task flow authorization. Then there are best practices that will help you to secure the ADF application to allow users to access the web pages you intend.

After you have completed the process of enabling ADF Security, you may want to customize your application to work with ADF Security in the user interface. For example, you can use Expression Language (EL) to render UI components in the web page based on evaluation of custom permissions that you define just for a group of UI components. Additionally, you can define methods within a managed bean to expose information, such as the user name and role membership, in your application.

Using Expression Language (EL) with ADF Security

You can use Expression Language (EL) to evaluate the policy directly in the UI, while the use of Java enables you to evaluate the policy from within a managed bean. ADF Security implements several convenience methods for use in EL expressions to access ADF resources in the security context. For example, you can use the EL expression and ADF Security Context API methods to determine whether the user is allowed to access a particular task flow. Good security practice dictates that your application should hide resources and capabilities for which the user does not have access. And for this reason, if the user is not allowed access to a particular task flow, you would evaluate the user's permission grant to determine whether or not to render the navigation components that initiate the task flow.

> **Note:**
>
> The ability to evaluate a policy is limited to the current request. For this reason, it is important to understand where the policy evaluation occurs, because evaluating the policy at anything other than the request scope can lead to unexpected results.

How to Evaluate Policies Using EL

The use of EL within a UI component allows for the component's attribute values to be defined dynamically, resulting in modification of the UI component at runtime. In the case of securing resources, the UI component attribute of interest is the `rendered` attribute, which allows you to show and hide components based on available permissions. By default, the `rendered` attribute is set to `true`. By dynamically changing this value based on the permission, you can set the UI component to be shown or hidden. For example, if the user has the appropriate permission, the `rendered` attribute should be set to `true` so that the UI component is shown. If they do not have permission, the attribute should be set to `false` and the UI component hidden from view.

To evaluate a policy using EL, you must use the ADF Security methods in the `securityContext` EL namespace. These methods let you access information in the ADF security context for a particular user or ADF resource.

Table 47-11 shows the EL expression that is required to determine whether a user has the associated permission. If the user has the appropriate permission, the EL expression evaluates to `true`; otherwise, it returns `false`.

Table 47-11 EL Expression to Determine View Permissions on ADF Resources

Expression	Expression action

`#{securityContext.taskflowViewable['`*`MyTaskFlow`*`']}`
For example:
`#{securityContext.taskflowViewable ['/WEB-INF/audit-expense-report.xml#audit-expense-report']}`
	Where *`MyTaskFlow`* is the WEB-INF node-qualified name of the task flow being accessed. Returns `true` if the user has access rights. Returns `false` if the user does not have sufficient access rights.

`#{securityContext.regionViewable['`*`MyPagePageDef`*`']}`	Where *`MyPagePageDef`* is the qualified name of the page definition file associated with the web page being accessed. Returns `true` if the user has access rights. Returns `false` if the user does not have sufficient access rights.

> **Note:**
>
> In the case of page permission, the value of the page definition can be specified dynamically by using late-binding EL within a managed bean, as described in What You May Need to Know About the valid-users Role.

Table 47-12 shows the EL expression that lets you get general information from the ADF security context not related to a particular ADF resource. For example, you can access the current user name when you want to display the user's name in the user interface. You can also check whether the current user is a member of certain roles or granted certain privileges. Your application may use this result to dynamically hide or show menus.

Table 47-12 EL Expression to Determine User Information in the ADF Security Context

Expression	Expression Action
`#{securityContext.userName}`	Returns the user name of the authenticated user.
`#{securityContext.authenticated}`	Returns `true` if the user is logged in. Returns `false` if the user is not logged in. This is useful for rendering a dynamic link for login/logout, or for rendering a "Welcome, *username*" message when the user has been authenticated. For an example that uses this expression, see Adding Login and Logout Links.
`#{securityContext.userInRole['`*`roleList`*`']}`	Where *`roleList`* is a comma-separated list of role names. Returns `true` if the user is in at least one of the roles. Returns `false` if the user is in none of the roles, or if the user is not currently authenticated.

Table 47-12 (Cont.) EL Expression to Determine User Information in the ADF Security Context

Expression	Expression Action
`#{securityContext.userInAllRoles['roleList']}`	Where *`roleList`* is a comma-separated list of role names. Returns `true` if the user is in all of the roles. Returns `false` if the user is not in all of the roles, or if the user is not currently authenticated.
`#{securityContext.userGrantedPermission['permission']}`	Where *`permission`* is a string containing a semicolon-separated concatenation of `permissionClass=<class>;target=<artifact_name>;action=<action>`. Returns `true` if the user has access rights. Returns `false` if the user does not have sufficient access rights. Note that the convenience methods `taskflowViewable` and `regionViewable` shown in Table 47-11 provide the same functionality.
`#{securityContext.userGrantedResource['resource']}`	Where *`resource`* is a string containing a semicolon-separated concatenation of `resourceName=<name>;resourceType=<type>;action=<action>`. Returns `true` if the user has access rights. Returns `false` if the user does not have sufficient access rights. You can use this expression to test the permission grant in the display property of a resource that is not contained in a task flow (like an ADF Faces panel or a menu item). You first create a custom resource type for the UI component that you want to protect. For more details, see How to Protect UI Components Using OPSS Resource Permissions and EL.

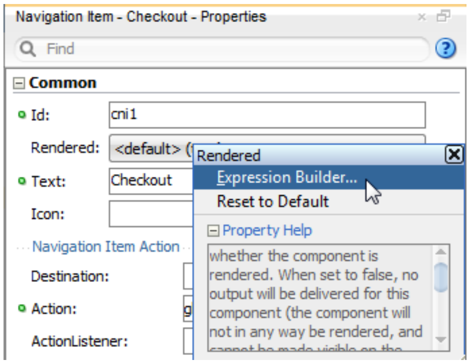
Before you begin:

It may be helpful to have an understanding of using EL. For more information, see Using Expression Language (EL) with ADF Security.

To associate the rendering of a navigation component with a user's granted permissions on a target task flow or page definition:

1. In the Applications window, double-click the page that contains the navigation component that you wish to conditionally render.
2. In the visual editor for the page, select the component that is used to navigate to the secured page.
3. In the Property window, click the **Property Menu** dropdown menu displayed menu next to the **Rendered** field and choose **Expression Builder**, as shown in Figure 47-32.

Figure 47-32 Binding the Rendered Property to Data



Navigation Item - Checkout - Properties

Find

Common

Id: cni1

Rendered: <default>

Text: Checkout

Icon:

Navigation Item Action

Destination:

Action:

ActionListener:

Rendered

Expression Builder...

Reset to Default

Property Help

whether the component is rendered. When set to false, no output will be delivered for this component (the component will not in any way be rendered, and

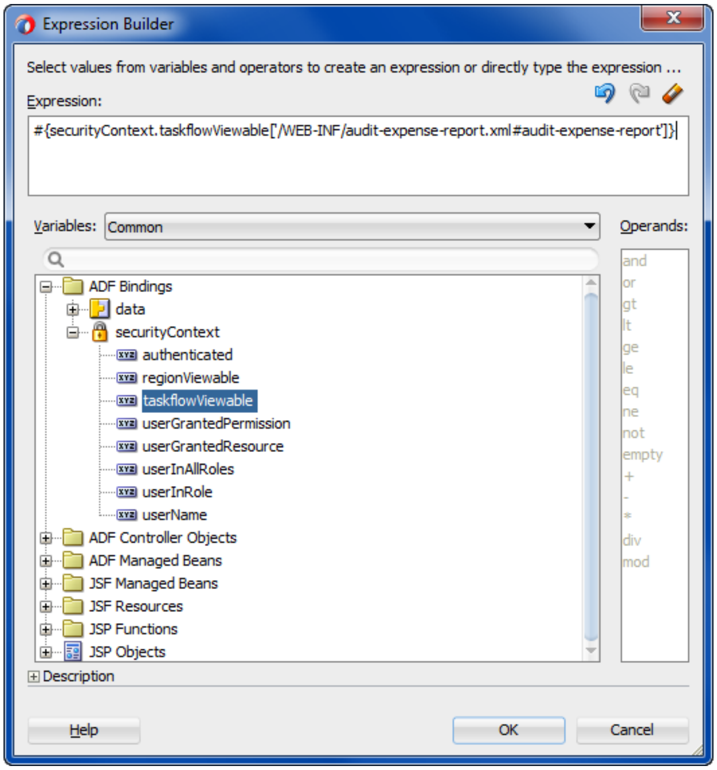
4. In the Expression Builder, expand the **ADF Bindings - securityContext** node and select the appropriate EL value, and then in the **Expression** field, enter the qualified name of the ADF resource that the user will attempt to access.

   For example, as shown in Figure 47-33, to limit access to a task flow that your application displays, you would create an expression like:

   ```
   #{securityContext.taskflowViewable
        ['/WEB-INF/audit-expense-report.xml#audit-expense-report']}
   ```

   In this example, the expression determines the user's access rights to view the target task flow `audit-expense-report`. If the user has the access rights, then the expression evaluates to `true` and the `rendered` attribute receives the value `true`.

Figure 47-33 Defining EL in the Expression Builder Dialog



Expression Builder

Select values from variables and operators to create an expression or directly type the expression ...

Expression:

```
#{securityContext.taskflowViewable['/WEB-INF/audit-expense-report.xml#audit-expense-report']}
```

Variables: Common

- ADF Bindings
  - data
  - securityContext
    - authenticated
    - regionViewable
    - taskflowViewable
    - userGrantedPermission
    - userGrantedResource
    - userInAllRoles
    - userInRole
    - userName
- ADF Controller Objects
- ADF Managed Beans
- JSF Managed Beans
- JSF Resources
- JSP Functions
- JSP Objects

> **Tip:**
>
> In the Expression Builder dialog, expand **Description** for additional information about any security EL method you select.

5. Click **OK**.

When you run the application, the component will be rendered or hidden based on the user's ability to view the target page.

What Happens When You Use the Expression Builder Dialog

When you use the Expression Builder to define an expression for the `rendered` attribute, JDeveloper updates the component definition in the open `.jspx` file. The component's `rendered` attribute appears with an expression that should evaluate to either `true` or `false`, as shown in the following example. In this example, the component is a navigation link with the link text `Checkout` defined by another expression. The page that contains the navigation link renders the component only when the user has sufficient rights to access the checkout task flow.

```
<af:commandNavigationItem
   text="#{res['global.nav.checkout']}"
   action="globalCheckout"
   id="cni3"
   rendered="#{securityContext.taskflowViewable
                    ['/WEB-INF/checkout-task-flow.xml#checkout-task-flow']}"
/>
```

What You May Need to Know About Delayed Evaluation of EL

The ability to evaluate a security permission is scoped to the request. If you want to evaluate permissions to access a target page from a managed bean that is scoped to a higher level than request (for example, a global menu that is backed by a managed bean), you must implement delayed EL evaluation (late-binding). By passing in the target page as a managed property of the bean, you ensure that the EL expression is evaluated only after the required binding information is available to the managed bean. Because EL is evaluated immediately when the page is executed, placing the EL expression directly in the properties of a UI component, backed by a managed bean, would result in an out-of-scope error.

The following example shows a property (`authorized`) of a managed bean that returns `true` or `false` based on a user's ability to view a named target page. In this case, the `_targetPageDef` variable is a managed property containing the name of the target page. Within the UI, the EL expression would reference the `authorized` property.

```
public boolean isAuthorized()
{
 if (_targetPageDef != null) {
  FacesContext fctx = FacesContext.getCurrentInstance();
  ADFContext adfCtx = ADFContext.getCurrent();
  SecurityContext secCtx = adfCtx.getSecurityContext();
  boolean hasPermission = secCtx.hasPermission(new RegionPermission
     (_targetPageDef, RegionPermission.VIEW_ACTION));
     if (hasPermission) {
        return hasPermission;
     }
```

```
          else {
             fctx.addMessage(null, new FacesMessage (
             FacesMessage.SEVERITY_WARN, "Access Permission not defined! " , null));
             return false;
          }
      }
```

How to Protect UI Components Using OPSS Resource Permissions and EL

Security expressions in the ADF Security EL namespace described in Table 47-12 let you check the user permission, authentication and role membership status before displaying UI components in your page. When you need fine-grained protection over UI components to secure functional parts of the application that are not part of task flow security or web page security, you use the EL value `userGrantedResource` and a resource grant that you define for a custom resource type. For instance, through a resource grant that you create for a menu item, the application may either enable or disable a Cancel Shipment menu item to limit who may perform customer order updates.

Resource grants for custom resources types rely on the `ResourcePermission` class, which is provided by Oracle Platform Security Services (OPSS) and packaged with the application.

You use the overview editor for security policies to create a security policy for the custom resource type. And, in your web page, you use the security expression to evaluate the user's access rights to the UI component projected by the resource grant.

> **Best Practice:**
>
> Custom resource types in combination with the `ResourcePermission` matcher class let you extend ADF Security to define custom actions for use in grants. This gives you the flexibility to define security policies to manage the user's ability to view UI components without having to overload the built-in actions defined by the ADF resources' permission classes. Be aware that you do not need to create custom resource types to manage access to web pages. This level of access is provided by the default ADF Security view permission that you work with in the overview editor for security policies.

To grant resource policies for a custom resource type:

1. Create the custom resource type for the UI component you want to protect.
2. Create the resource grant for the custom resource.
3. Associate the rendering of a UI component with a role's granted resource permission.

Creating the Custom Resource Type

You use the Create Resource Type dialog to create a resource type for the UI component that you want to protect. JDeveloper will match your resource type definition to the OPSS permission class `oracle.security.jps.ResourcePermission`.

This class allows you to specify custom actions that you can grant to functional parts of your application that are not protected by task flow security or web page security. For example, you might want to create a resource type to protect sensitive parts of the application accessed by the user through a menu UI component.

The Create Resource Type dialog lets you identify the type of resource you wish to protect (for example, the name `MenuProtection` would identify the menu UI component), a display name visible in the security policy store (for example, `Menu Protection`), and a description of how the resource type will be used in the resource grant (for example, `Resource permission to grant menu item permissions`). The dialog also lets you enter one or more custom action names that you will use to grant permission. For example, you might name the action `process` for use in a security policy that grants access to a protected menu item.

Before you begin:

It may be helpful to have an understanding of the ADF Permission class. For more information, see How to Protect UI Components Using OPSS Resource Permissions and EL.

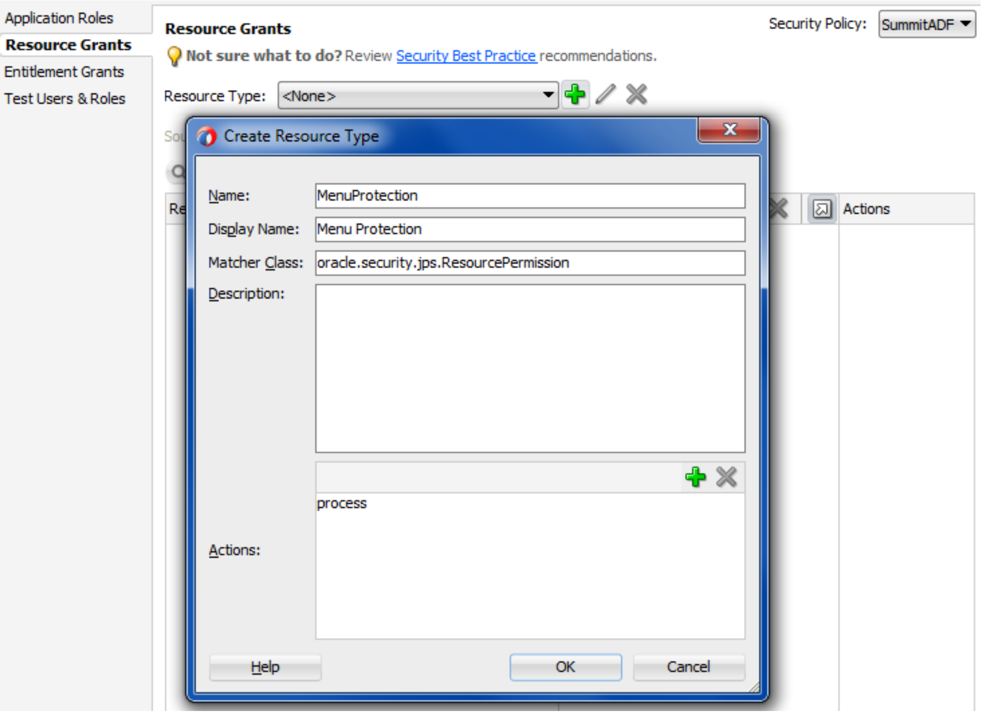
To create the custom resource type:

1. In the main menu, choose **Application** and then **Secure > Resource Grants**.
2. In the Resource Grants page of the overview editor for security policies, click **New Resource Type**.
3. In the Create Resource Type dialog, enter the name and display name that describes the UI component.

   For example, to protect menu items, you might enter `MenuProtection` for the name and `Menu Protection` for the display name.
4. In the **Actions** list, click **Add** and enter the name of the actions that your permission will grant and click **OK**.

   For example, when protecting a menu item, you might enter `process` to signify that permission to process the menu selection is being granted. You can enter multiple action names when you intent to grant specific actions to individual users or roles. The overview editor displays the custom permission grant, as shown in Figure 47-34.

Figure 47-34 Creating a Custom Resource Type



Name: MenuProtection
Display Name: Menu Protection
Matcher Class: oracle.security.jps.ResourcePermission
Description:
Actions: process

Creating a Resource Grant for a Custom Resource Type

You use the overview editor for the `jazn-data.xml` file to create the ADF Security policy for a custom resource. The finished source should look similar to the permission grant defined in the policy store, as shown in the following example.

```
<grant>
  <grantee>
    <principals>
      <principal>
         <name>AccountManagersadmin</name>
         <class>oracle.security.jps.service.policystore.ApplicationRole</class>
      </principal>
     </principals>
  </grantee>
  <permissions>
     <permission>
       <class>oracle.security.jps.ResourcePermission</class>
       <name>resourceType=MenuProtection,resourceName=CancelShipment</name>
       <actions>process</actions>
     </permission>
   </permissions>
</grant>
```

The `<permission>` target name identifies the custom resource type and name that you assigned the resource. For example, a resource name that lets users cancel shipments from a menu selection in their account might be named `CancelShipment`.

The actions are those that your custom resource type defines. For example, in the Menu Protection resource type defines the single action `process` to grant permission to process the menu selection.

Before you begin:

It may be helpful to have an understanding of the `ResourcePermission` class. For more information, see How to Protect UI Components Using OPSS Resource Permissions and EL.

You will need to complete this task:

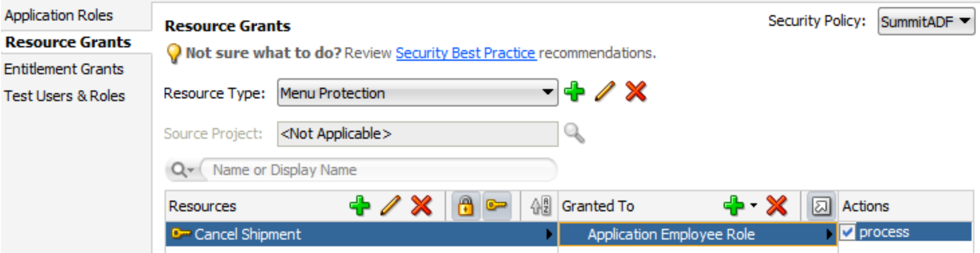
> Create the custom resource type, as described in Creating the Custom Resource Type.

To create the resource grant for a custom resource type:

1. In the main menu, choose **Application** and then **Secure > Resource Grants**.
2. In the Resource Grants page of the overview editor for security policies, select the custom resource from the **Resource Type** dropdown list.
3. In the **Resources** column, click the **Add Resource** icon.
4. In the Create Resource dialog, enter the name, display name, and description for the resource that the security policy will specifically protect and then click **OK**.
5. In Resource Grants page of the overview editor, in the **Granted to** column, click the **Add Grantee** icon and choose **Add Application Role**.
6. In the **Actions** column, select action that you defined for the custom resource type.

   The overview editor displays the resource grant, as shown in Figure 47-35.

Figure 47-35 Creating a Grant for a Custom Resource in the Overview Editor



Resource Grants

Security Policy: SummitADF

Not sure what to do? Review Security Best Practice recommendations.

Resource Type: Menu Protection

Source Project: <Not Applicable>

Resources	Granted To	Actions
Cancel Shipment	Application Employee Role	☑ process

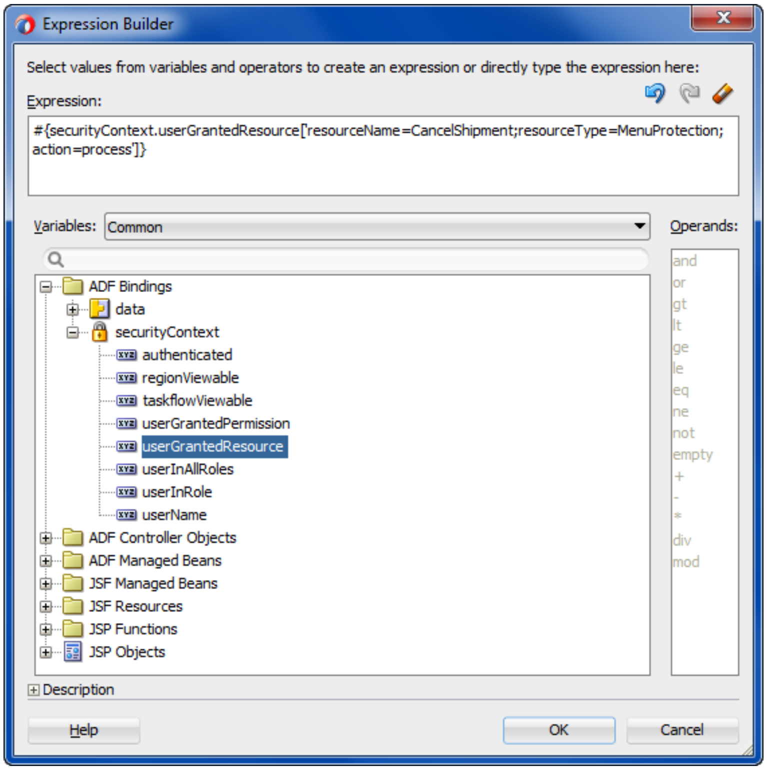
Associating the Rendering of a UI Component with a Resource Grant

You use the Expression Builder dialog for the UI component display property to define an EL expression that controls rendering the component. For example, to protect a menu item in an application that allows authorized users to cancel shipments, the application specifies a `userGrantedResource` expression on the **disabled** property of the menu item defined by `af:commandMenuItem`. As the following example shows, the expression tests whether the user has been granted the resource type and then either displays the menu item or, when the user has not been granted the resource, disables the menu item.

```
#{!securityContext.userGrantedResource

['resourceName=CancelShipment;resourceType=MenuProtection;action=process']}
```

Figure 47-36 shows how the expression appears in the Expression Builder dialog.

Figure 47-36   Defining EL in the Expression Builder Dialog



Select values from variables and operators to create an expression or directly type the expression here:

Expression:

```
#{securityContext.userGrantedResource['resourceName=CancelShipment;resourceType=MenuProtection;
action=process']}
```

Variables: Common

Before you begin:

It may be helpful to have an understanding of the `ResourcePermission` class. For more information, see How to Protect UI Components Using OPSS Resource Permissions and EL.

You will need to complete these tasks:

1. Create the custom resource type, as described in Creating the Custom Resource Type.
2. Create the ADF security policy using the custom permission, as described in Creating a Resource Grant for a Custom Resource Type.

To associate the rendering of a UI component with a user's granted resource:

1. In the Applications window, double-click the page that contains the UI component that you wish to conditionally render.
2. In the visual editor for the page, select the component that is used to navigate to the secured page.
3. In the Property window, click the **Property Menu** dropdown menu displayed menu next to the **Rendered** field and choose **Expression Builder**.
4. In the Expression Builder, expand the **ADF Bindings - securityContext** node and select **userGrantedResource**, and then, in the **Expression** field, enter a concatenated string that defines the permission.

   Enter the permission string as a semicolon-separated concatenation of `resourceName=`*`theResourceInstanceName`*`;resourceType=`*`customResourceTypeNa`* *me*;action=*actionName*. For example, to protect a menu item in an application that allows authorized users to cancel shipments, you would enter an expression similar to the one shown in the following example, where the resource type for userGrantedResource identifies the resource type used in the custom resource grant.

```
#{!securityContext.userGrantedResource

['resourceName=CancelShipment;resourceType=MenuProtection;action=process']}
```

In this example, the expression evaluates the permission based on the custom resource type named MenuProtection that you added to the application policy store.

5. Click **OK**.

How to Perform Authorization Checks for Entity Object Operations

In the Fusion web application, you can enable security for the data collection when you secure the operations of the entity object. The user must have sufficient privileges to view, update, or delete the data for the entire row set.

The oracle.jbo.ViewCriteriaRow API allows you to perform authorization checks for the standard operations (read, update, removeCurrentRow) at the level of entity rows. The following example shows how to use the authorization checking API, getSecurityHints() and allowsOperation(), on the entity row, to test whether the user has permission to delete data.

```
if(row.getSecurityHints().allowsOperation("removeCurrentRow").hasPermission())
        // code for the data
else
        // display error message
```

Note that the allowsOperation() method expects as input the secured operation name rather than the name of the action that secures it. The operation name for deleting an entity row is removeCurrentRow (where delete is the action name).

When you want to perform authorization checks for the entity object that underlies data bound components in a JSF page of the Fusion web application, you can use EL expressions. For example, the following EL expression returns true if the user has been granted the permission to perform the operation specified. The <*operationName*> used in the expression must be the read, update, or removeCurrentRow operation that you enabled for the entity object.

```
#{row.hints.allows.<operationName>}
```

Authorization checking expressions are evaluated in the context of a row collection. When you want to check entity permissions outside of the context of the row collection, you must use the userGrantedResource method as shown in Table 47-12.

Getting Information from the ADF Security Context

The implementation of security in a Fusion web application is by definition an implementation of the security infrastructure of the ADF Security framework. As such, the security context of the framework allows access to information that is required as you define the policies and the overall security for your application.

How to Determine Whether Security Is Enabled

Because the enforcement of ADF Security can be turned on and off at the container level independent of the application, you should determine whether ADF Security is enabled prior to making authorization checks. You can achieve this by calling the `isAuthorizationEnabled()` method of the ADF Security Context, as shown in the following example.

```
if (ADFContext.getCurrent().getSecurityContext().isAuthorizationEnabled()){
  //Authorization checks are performed here.
}
```

How to Determine Whether the User Is Authenticated

As the user principal in a Fusion web application is never `null` (that is, it is either `anonymous` for unauthenticated users or the actual user name for authenticated users), it is not possible to simply check whether the user principal is `null` to determine if the user has logged on or not. As such, you must use a method to take into account that a user principal of `anonymous` indicates that the user has not authenticated. You can achieve this by calling the `isAuthenticated()` method of the ADF Security Context, as shown in the following example.

```
// ============ User's Authenticated Status =============
private boolean _authenticated;
public boolean isAuthenticated() {
  _authenticated =
ADFContext.getCurrent().getSecurityContext().isAuthenticated();
  return _authenticated;
}
```

How to Determine the Current User Name

Fusion web applications support the concept of public pages that, while secured, are available to all users. Furthermore, components on the web pages, such as portlets, require knowledge of the current user identity. As such, the user name in a Fusion web application will never be `null`. If an unauthenticated user accesses the page, the user name `anonymous` will be passed to page components.

You can determine the current user's name by evaluating the `getUserName()` method of the ADF security context, as shown in the following example. This method returns the string `anonymous` for all unauthenticated users and the actual authenticated user's name for authenticated users.

```
// ============ Current User's Name/PrincipalName =============
     public String getCurrentUser() {
      _currentUser = ADFContext.getCurrent().getSecurityContext().getUserName();
      return _currentUser;
     }
```

Because the traditional method for determining a user name in a Faces-based application (`FacesContext.getCurrentInstance().getExternalContext().getRemoteUser()`) returns `null` for unauthenticated users, you need to use additional logic to handle the public user case if you use that method.

How to Determine Membership of a Java EE Security Role

As Fusion web applications are JavaServer Faces-based applications, you can use the isUserInRole(roleName) method of the Faces external context, as shown in the following example, to determine whether a user is in a specified role. Because ADF Security is based around JAAS policies, you should not need to use Java EE security roles to secure pages associated with ADF security-aware resources based on role membership. However, you might use the method to check the role for a page that is not associated with an ADF security-aware resource.

In this example, a convenience method (checkIsUserInRole) is defined. The use of this method within a managed bean enables you to expose membership of a named role as an attribute, which can then be used in EL.

```
public boolean checkIsUserInRole(String roleName){
        return
(FacesContext.getCurrentInstance().getExternalContext().isUserInRole(roleName));
}

public boolean isCustomer() {
        return (checkIsUserInRole("Application Customer Role"));
 }
```

How to Determine Permission Using Java

To evaluate the security policies from within Java, you can use the hasPermission method of the ADF security context. This method takes a permission object (defined by the resource and action combination) and returns true if the user has the corresponding permission.

In the following example, a convenience function is defined to enable you to pass in the name of the page and the desired action, returning true or false based on the user's permissions. Because this convenience function is checking page permissions, the RegionPermission class is used to define the permission object that is passed to the hasPermission method.

```
private boolean TestPermission (String PageName, String Action)  {
  Permission p = new RegionPermission("view.pageDefs." + PageName + "PageDef",
                                       Action);
  if (p != null) {
     return ADFContext.getCurrent().getSecurityContext().hasPermission(p);
  }
  else {
     return (true);
}
```

As it is possible to determine the user's permission for a target page from within a backing bean, you can use this convenience method to dynamically alter the result of a Faces navigation action. In the following example, you can see that a single command button can point to different target pages depending on the user's permission. By checking the view permission from the most secured page (the manager page) to the least secured page (the public welcome page), the command button's backing bean will apply the appropriate action to direct the user to the page that corresponds to their permission level. The backing bean that returns the appropriate action uses a convenience method defined in the following example.

```
//Button Definition
<af:button text="Goto Your Group Home page"
  binding="#{backing_content.commandButton1}"
  id="commandButton1"
  action="#{backing_content.getSecureNavigationAction}"/>

//Backing Bean Code
    public String getSecureNavigationAction() {
      String ActionName;
      if (TestPermission("ManagerPage", "view"))
        ActionName = "goToManagerPage";
      else if (TestPermission("EmployeePage", "view"))
        ActionName = "goToEmployeePage";
      else
        ActionName = "goToWelcomePage";
      return (ActionName);
    }
```

Best Practices for Working with ADF Security

These best practices summarize the rules that govern enforcement of security by the ADF Security framework. Understanding these best practices will help you to secure the application to allow users to access the web pages you intend.

Do build your application with ADF Security enabled from the start.

When you enable security, you essentially lock down the application and you will be required to make explicit permission grants to specific ADF security-aware resources you create. Knowing about these resources and making grants on them as you build the application will enable you to iteratively test security to ensure that you structure your application in a way that achieves the desired result.

Do define permission grants for bounded task flows.

Pages that the user accesses within the process of executing a bounded task flow will not be individually permission-checked and will run under the permission grants of the task flow. This means that any page that you add to the task flow should not have its own page definition-level security defined. Upon requesting a flow, the user will be allowed either to view all the pages of the task flow or to view none of the pages, depending on their level of access.

Do not define permission grants for individual pages of a bounded task flow.

It is important to realize that task flows do not prevent users from accessing pages directly. Any web page that is located in a directory that is publicly accessible can be reached from a browser using a URL. To ensure that pages referenced by a bounded task flow cannot be accessed directly, remove all permission grants that exist for their associated page definition file. When pages require additional security within the context of a bounded task flow, wrap those pages in a sub-task flow with additional grants defined on the nested task flow.

Do use task flows to reduce the number of access points exposed to end users.

When you use task flows you can reduce the number of access points that you expose to end users. For example, configure an unbounded task flow to display one page that provides navigation to the remaining pages in your application. Use bounded task flows for the remaining pages in the application. By setting the URL Invoke property of these bounded task flows to `url-invoke-disallowed`, your application has one access point (the page on the unbounded task flow). For more information about the URL Invoke property, see How to Call a Bounded Task Flow Using a URL.

Do specify an alternative task flow for unauthorized users.

End users who attempt to invoke a bounded task flow in a region for which they do not have the necessary permission will see a region that renders as blank in place of the requested task flow. The region when left blank may confuse end users as it does not provide feedback as to why the requested task flow does not appear. Consider configuring an alternative task flow that does not require authorization when this scenario occurs. For more information configuring an alternative task flow, see Handling Access to Secured Task Flows by Unauthorized Users.

Do define permission grants for individual pages outside of a bounded task flow.

Page-level security is checked for pages that have an associated page definition binding file *only* if the page is directly accessed or if the page is accessed in an *unbounded* task flow. There is a one-to-one relationship between the page definition file and the web page it secures.

If you want to secure a page that uses no ADF bindings, you can create an empty page definition binding file for the page.

Do define custom permissions to render UI component based on the user's access rights.

Custom ADF permission classes let you extend ADF Security to define custom actions for use in grants. This gives you the flexibility to define security policies to manage the user's ability to view UI components without having to overload the built-in actions defined by the ADF resources' permission classes.

Do define entity object attribute permissions to manage the user's access rights to row-level data displayed by UI components.

Entity objects and entity object attributes both define permission classes that let you define permissions for the read, update, and delete operations that the entity object initiates on its data source. In the case of these data model project components, you must explicitly grant permissions to an application role in order to opt into ADF Security authorization. However, once you enable authorization for an entity object, all rows of data defined by the entity object will be protected by the grant. At this level of granularity, your table component would render in the web page either with all data visible or with no data visible—depending on the user's access rights. As an alternative to securing the entire collection, you can secure individual columns of data. This level of granularity is supported by permissions you set on the individual attributes of entity objects. When entity objects are secured, users may see only portions of the data that the table component displays.

Do use task flow or page-level permission grants to avoid exposing row-level create/insert operations to users with view-only permission.

The correct way to control access to a page that should allow only certain users to update new rows in a table is to use task flow or page-level permission grants. However, as an alternative, it is possible to secure table buttons corresponding to particular operations by specifying an EL expression to test the user's access rights to view the button. When the custom permission is defined and the `userGrantedPermission` expression is set on the `Rendered` property of the button, only users with sufficient privileges will see the button. This may be useful in a case where the user interface displays a page that is not restricted and view-only permission for row-level data is defined for the entity object. In this case, when viewed by the user, the Delete button for the editable table associated with the entity object will appear disabled. However, in the case of an input table, the user interface does not disable the button for the `CreateInsert` operation even though the user may not have update permission.

Do not use JDeveloper as a user identity provisioning tool.

JDeveloper must not be used as an identity store provisioning tool, and you must be careful not to deploy the application with user identities that you create for testing purposes. Deploying user identities with the application introduces the risk that malicious users may gain unintended access. Instead, always rely on the system administrator to configure user identities through the tools provided by the domain-level identity management system. You should delete all users and groups that you create in the `jazn-data.xml` file before deploying the application.

Do not allow users to access a web page by its file name.

When you deploy the Fusion web application, you should always permit users to access the web page from a view activity defined in the ADF Controller configuration file. Do not allow users to access the JSPX file directly by its physical name (for example, similar to the file name `AllDepartments.jspx`.

Assuming the view activity is named `AllDepartments`, then there are two ways to call the page:

1. `localhost:7101/myapp/faces/AllDepartments`
2. `localhost:7101/myapp/faces/AllDepartments.jspx`

The difference is that the call 1) is in the context of the ADF Controller task flow, which means that navigation on the page will work and any managed beans that are referenced by the page will be properly instantiated. The call in 2) also serves the page, however, the page may not function fully. This may be considered a security breach.

To prevent direct JSPX file access, move the JSPX file under the `/public_html/WEB-INF` directory so that direct file access is no longer possible. To access a document, users will have to call its view activity name.

Note that this suggestion does not protect documents that are unprotected in ADF Security. It only helps to lock down access to the physical file itself.

Thus, the following security guidelines still apply:

1. Apply ADF Security permissions to all JSPX documents associated with view activities defined in the `adfc-config.xml` file.
2. Move all JSPX documents in the user interface project under the `/public_html/WEB-INF` directory to prevent direct file access.
3. Limit the pages in the `adfc-config.xml` to the absolute minimum and place all other pages into bounded task flows.
4. Make bounded task flows inaccessible from direct URL access (which is the default configuration setting for new task flows).
5. Apply ADF Security permissions to bounded task flows.

48

Testing and Debugging ADF Components

This chapter describes the tools for logging and testing an application that uses Oracle ADF. It contains debugging procedures for setting breakpoints using the ADF Declarative Debugger. Finally, it explains how to write and run regression tests for your ADF Business Components-based business services.
This chapter includes the following sections:

About ADF Debugging

ADF debugging is the process of identifying, separating, and fixing specific contributing factors leading to failures at runtime.

Like any debugging task, debugging the web application's interaction with **Oracle Application Development Framework** (Oracle ADF) is a process of isolating specific contributing factors. However, in the case of web applications, generally this process does not involve compiling Java source code. Your web pages contain no Java source code, as such, to compile. In fact, you may not realize that a problem exists until you run and attempt to use the application. For example, these failures are only visible at runtime:

- A page not found servlet error
- The page is found, but the components display without data
- The page fails to display data after executing a method call or built-in operation (like Next or Previous)
- The page displays, but a method call or built-in operation fails to execute at all
- The page displays, but unexpected validation errors occur

The failure to display data or to execute a method call arises from the interaction between the web page's components and the **ADF Model** layer. When a runtime failure is observed during ADF lifecycle processing, the sequence of preparing the model, updating the values, invoking the actions, and, finally, rendering the data failed to complete.

Fortunately, most failures in the web application's interaction with Oracle ADF result from simple and easy-to-fix errors in the declarative information that the application defines or in the EL expressions that access the runtime objects of the page's ADF **binding container**.

In your databound Fusion web application, you should examine the declarative information and EL expressions as likely contributing factors when runtime failures are observed. To understand editing the declarative files, see Correcting Simple Oracle ADF Compilation Errors, and Correcting Simple Oracle ADF Runtime Errors.

One of the most useful diagnostic tools is the ADF Logger. You use this logging mechanism in JDeveloper to capture runtime traces messages. With ADF logging enabled, JDeveloper displays the application trace in the Message Log window. The trace includes runtime messages that may help you to quickly identify the origin of an application error. Read Using the ADF Logger, to configure the ADF Logger to display detailed trace messages.

Supported Oracle ADF customers can request Oracle ADF source code from Oracle Worldwide Support. This can make debugging **ADF Business Components** framework code a lot easier. Read Using ADF Source Code with the Debugger, to understand how to configure JDeveloper to use the Oracle ADF source code.

If the error cannot be easily identified, you can utilize the ADF Declarative Debugger in JDeveloper to set breakpoints. When a breakpoint is reached, the execution of the application is paused and you can examine the data that the ADF binding container has to work with, and compare it to what you expect the data to be. Depending on the types of breakpoints, you may be able to use the step functions to move from one breakpoint to another. For more information about the debugger, read Using the ADF Declarative Debugger.

JDeveloper provides integration with JUnit for your Fusion web application through a wizard that generates regression test cases. Read Regression Testing with JUnit, to understand how to write test suites for your application.

Correcting Simple Oracle ADF Compilation Errors

Simple Oracle ADF compilation errors occur when you modify the Oracle declarative files and it does not conform to the XML schema defined by Oracle ADF.

When you create web pages and work with the ADF **data controls** to create the **ADF binding** definitions in JDeveloper, the Oracle ADF declarative files you edit must conform to the XML schema defined by Oracle ADF. When an XML syntax error occurs, the JDeveloper XML compiler immediately displays the error in the Structure window.

Although there is some syntax checking during design time, the JDeveloper compiler is currently limited by an inability to resolve EL expressions. EL expressions in your web pages interact directly with various runtime objects in the web environment, including the web page's ADF binding container. At present, errors in EL expressions can be observed only at runtime. Thus, the presence of a single typing error in an object-access expression will not be detected by the compiler, but will manifest at runtime as a failure to interact with the binding container and a failure to display data in the page. For information about debugging runtime errors, see Correcting Simple Oracle ADF Runtime Errors.

Tip:

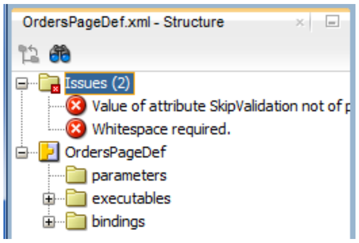
The Expression Builder is a dialog that helps you build EL expressions by providing lists of objects, managed beans, and properties. It is particularly useful when creating or editing ADF databound EL expressions because it provides a hierarchical list of ADF binding objects and their valid properties from which you can select. You should use the Expression Builder to avoid introducing typing errors. For details, see Creating ADF Data Binding EL Expressions.

The following example illustrates two simple compilation errors contained in a **page definition file**: `fals` instead of `false` and `ChangeEventPolicy="ppr"/` instead of `ChangeEventPolicy="ppr"/>` (that is, the attribute is missing a closing angle bracket).

```
<?xml version="1.0" encoding="UTF-8" ?>
<pageDefinition xmlns="http://xmlns.oracle.com/adfm/uimodel"
                Package="oracle.summit.view.pageDefs"
                SkipValidation="fals">
  <parameters/>
  <executables>
    <iterator Binds="InventoryForOrderItem" RangeSize="-1"
DataControl="BackOfficeAppModuleDataControl" id="InventoryForOrderItemIterator"
              ChangeEventPolicy="ppr"/
  ...
</pageDefinition>
```

During compilation, the Structure window displays the XML errors in the page, as shown in Figure 48-1.

Figure 48-1 The Structure Window Displays XML Errors

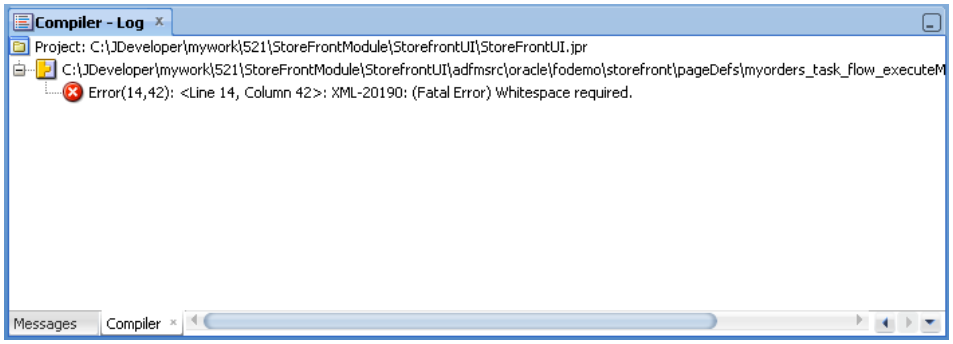


OrdersPageDef.xml - Structure

Issues (2)
Value of attribute SkipValidation not of p
Whitespace required.
OrdersPageDef
parameters
executables
bindings

The Compiler-Log window also displays the compilation errors in the page, as shown in Figure 48-2.

Figure 48-2   The Compiler Window Displays XML Compile Errors



```
Project: C:\JDeveloper\mywork\521\StoreFrontModule\StorefrontUI\StoreFrontUI.jpr
C:\JDeveloper\mywork\521\StoreFrontModule\StorefrontUI\adfmsrc\oracle\fodemo\storefront\pageDefs\myorders_task_flow_executeM
  Error(14,42): <Line 14, Column 42>: XML-20190: (Fatal Error) Whitespace required.
```

To view and correct schema validation errors:

1. In the main menu, choose **View** and then **Structure** to open the Structure window or choose **View** and then **Log** to open the Log Window.
2. In either window, double-click the error message.
3. In the XML editor, locate the highlighted lines.

   The highlighted lines will be lines with errors.
4. Correct any errors.

   After an error has been corrected, the corresponding error message will be automatically removed from the Structure window.
5. Optionally, you can recompile the project by choosing **Run > Make** and checking to see whether the compiler still produces the error message.

Correcting Simple Oracle ADF Runtime Errors

Simple Oracle ADF runtime errors occur when the failures of the ADF Model layer cannot be detected with the help of JDeveloper compiler.

Failures of the ADF Model layer cannot be detected by the JDeveloper compiler, in part because the page's data-display and method-execution behavior relies on the declarative ADF **page definition files**. The ADF Model layer utilizes those declarative files at runtime to create the objects of the ADF binding container.

To go beyond simple schema validation, you will want to routinely run and test your web pages to ensure that none of the following conditions exists:

- The project dependency between the data model project and the user interface project is disabled.

  By default, the dependency between projects is enabled whenever you create a web page that accesses a data control in the data model project. However, if the dependency is disabled and remains disabled when you attempt to run the application, an internal servlet error will be generated at runtime:

  ```
  oracle.jbo.NoDefException: JBO-25002: Definition
  model.DataControls.dcx of type null not found
  ```

To correct the error, double-click the user interface project, and select the **Dependencies** node in the dialog. Make sure that the ***ModelProjectName*.jpr** option appears selected in the panel.

- Page definition files have been renamed, but the `DataBindings.cpx` file still references the original page definition file names.

  While JDeveloper does not permit these files to be renamed within the IDE, if a page definition file is renamed outside of JDeveloper and the references in the `DataBindings.cpx` file are not also updated, an internal servlet error will be generated at runtime:

  ```
  oracle.jbo.NoDefException: JBO-25002: Definition
  oracle.<path>.pageDefs.<pagedefinitionName> of type Form Binding
  Definition not found
  ```

  To correct the error, open the `DataBindings.cpx` file and use the source editor to edit the page definition file names that appear in the `<pageMap>` and `<pageDefinitionUsages>` elements.

- The web page file (`.jsp` or `.jspx`) has been renamed, but the `DataBindings.cpx` file still references the original file name of the same web page.

  The page controller uses the page's URL to determine the correct page definition to use to create the ADF binding container for the web page. If the page's name from the URL does not match the `<pageMap>` element of the `DataBindings.cpx` file, an internal servlet error will be generated at runtime:

  ```
  javax.faces.el.PropertyNotFoundException: Error testing property
  <propertyname>
  ```

  To correct the error, open the `DataBindings.cpx` file and use the source editor to edit the web page file names that appear in the `<pageMap>` element.

- Bindings have been renamed in the web page EL expressions, but the page definition file still references the original binding object names.

  The web page may fail to display information that you expect to see. To correct the error, compare the binding names in the page definition file and the EL expression responsible for displaying the missing part of the page. Most likely the mismatch will occur on a **value binding**, with the consequence that the component will appear but without data. Should the mismatch occur on an **iterator binding** name, the error may be more subtle and may require deep debugging to isolate the source of the mismatch.

- Bindings in the page definition file have been renamed or deleted, and the EL expressions still reference the original binding object names.

  Because the default error-handling mechanism will catch some runtime errors from the ADF binding container, this type of error can be very easy to find. For example, if an iterator binding named `findUsersByNameIter` was renamed in the page definition file, yet the page still refers to the original name, this error will display in the web page:

  ```
  JBO-25005: Object name <iterator> for type Iterator Binding Definition
  is invalid
  ```

  To correct the error, right-click the name in the web page and choose **Go to Page Definition** to locate the correct binding name to use in the EL expression.

- EL expressions were written manually instead of using the expression picker dialog and invalid object names or property names were introduced.

This error may not be easy to find. Depending on which EL expression contains the error, you may or may not see a servlet error message. For example, if the error occurs in a binding property with no runtime consequence, such as displaying a label name, the page will function normally but the label will not be displayed. However, if the error occurs in a binding that executes a method, an internal servlet error `javax.faces.el.MethodNotFoundException: methodname` will display. Or, in the case of an incorrectly typed property name on the method expression, the servlet error `javax.faces.el.PropertyNotFoundException: propertyname` will display.

If this list of typical errors does not help you to find and fix a runtime error, you can initiate debugging within JDeveloper to find the contributing factor. For an Oracle ADF application, start setting ADF declarative breakpoints to find the problem. Using the ADF Declarative Debugger to set ADF declarative breakpoints is described in Using the ADF Declarative Debugger, and Setting ADF Declarative Breakpoints. This process involves pausing the execution of the application as it proceeds through the application and examining data. You can also use the ADF Declarative Debugger to set Java code breakpoints, as described in Setting Breakpoints in Java Code and Groovy Script.

Reloading Oracle ADF Metadata in Integrated WebLogic Server

Simply changing the ADF medatata files or class files will not have any effect on runtime after you have deployed the application to the Integrated WebLogic Server. You must recompile the project and refresh the web browser.

JDeveloper support for hot reloading of Oracle ADF metadata is an alternative to quitting the running application, editing your project's XML definition files, redeploying, and rerunning the application in Integrated WebLogic Server to view the latest changes.

Changes that you make to the Fusion web application projects will not be picked up automatically by an application that you have deployed to Integrated WebLogic Server. You can, however, reload metadata from the data model project and user interface project any time you want to synchronize the running application with changes you have made to your application's XML definition files.

To reload metadata so your changes are reflected in the deployed Fusion web application, you must recompile the project and refresh the web browser.

Metadata that JDeveloper will hot reload in Integrated WebLogic Server, include:

- In the data model project, changes to the definition files of business components.
- In the user interface project, changes to the binding definitions in the page definition files and changes to task flows in the task flow definition files.

This support makes it possible to make incremental changes and test them.

Validating ADF Controller Metadata

ADF Controller recovers metadata and then validation is performed. For example, in an application, the grammar in ADF Controller metadata is validated by enabling the enable-grammar-validationsetting in the adf-config.xml file.

Basic validation is performed when **ADF Controller** retrieves metadata. The most serious errors, for example, a task flow that is missing a default activity, result in parsing exceptions.

The `enable-grammar-validation` setting in `adf-config.xml` allows you to validate the grammar in ADF Controller metadata before deploying an application. When `enable-grammar-validation` is set to `true`, ADF Controller metadata is validated against ADF Controller XSDs. For example, invalid characters in ADF Controller metadata, such as a slash (/) in a **view activity** ID, are flagged as exceptions.

By default, `enable-grammar-validation` is set to `false`. For performance reasons, it should be set to `true` only during application development or when troubleshooting an application.

Using the ADF Logger

Logging is important in debugging and tracing errrors. In the ADF framework, you can use the ADF logger to identify and investigate application errors.

If you are not able to easily find the error in either your web page or its corresponding page definition file, you can use the JDeveloper debugging tools to investigate where your application failure occurs.

You configure the logging session by editing the `logging.xml` configuration file using the editor for Oracle Diagnostic Logging Configuration. Logging configuration can be set at any time, even while the application is running in JDeveloper.

Unlike many files in JDeveloper, you cannot directly open the `logging.xml` file. Instead you use menu commands at these locations to open the Oracle Diagnostic Logging Configuration editor:

- Right-click an active server instance in the Application Servers window.
- Click the Actions dropdown menu displayed in the Log window's debugger process panel, after you have started the application in debug mode.

You use the Oracle Diagnostic Logging Configuration editor to set the desired logging level to control the level and number of messages that are displayed. You can set the logging level for both persistent and transient loggers and declare handlers for each logger.

JDeveloper creates diagnostic log files in the Oracle Diagnostic Logging (ODL) format, used by Oracle Fusion Middleware components. Log file naming and the format of the contents of log files conform to an Oracle standard. By default, the diagnostic messages are in text format. For information about the ODL format, see Understanding ODL Messages and ODL Log Files in *Administering Oracle Fusion Middleware*.

In the editor, as an alternative to the default ODL format, you can configure the Java Logger to display Java diagnostic messages.

After you have created a log, you can view and filter the log messages with Oracle Diagnostic Log Analyzer. This tool allows you to set filters for different log levels, define message time frames, and search on message text.

You can then use the ADF Declarative Debugger to set breakpoints and examine the application. For additional information, see Setting ADF Declarative Breakpoints, and Setting Breakpoints in Java Code and Groovy Script.

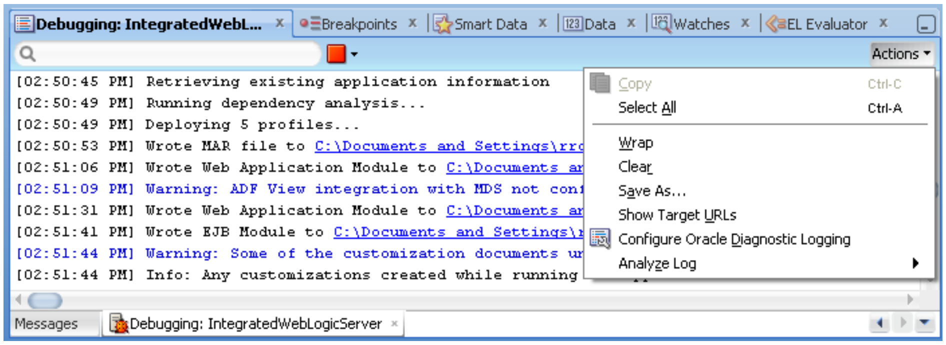
How to Set ADF Logging Levels

You can use the Oracle Diagnostic Logging Configuration editor to configure the logging levels specified in the `logging.xml` configuration file. The file can be configured before and while the application is running in Integrated WebLogic Server. The changes will apply without the need to restart the server.

When Integrated WebLogic Server is running, you can define both persistent and transient loggers. When Integrated WebLogic Server is not running, you can only define persistent loggers. The transient loggers will last only for the session and will not be entered in the `logging.xml` configuration file. If the server is not running, you must explicitly save the configuration changes to the `logging.xml` file for the updates to take effect in the next server run.

You can access the Oracle Diagnostic Logging Configuration editor from the Application Servers window or from the Log window, which is shown in Figure 48-3.

Figure 48-3   Log Window with Toolbar

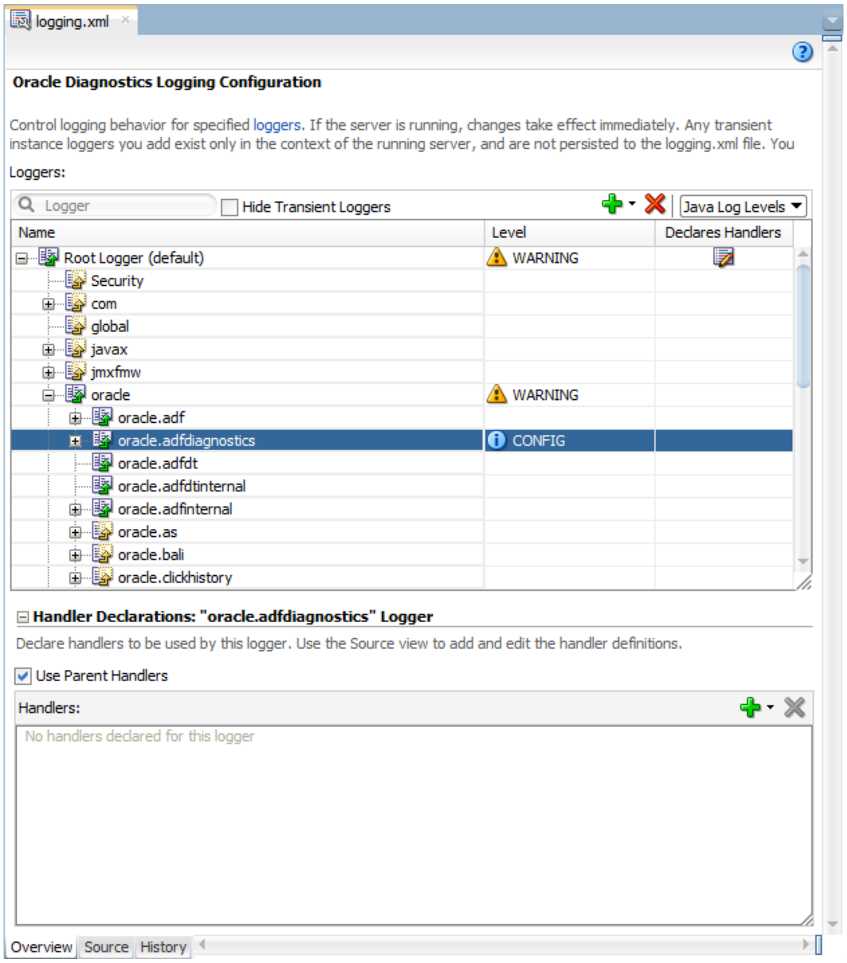


```
[02:50:45 PM] Retrieving existing application information
[02:50:49 PM] Running dependency analysis...
[02:50:49 PM] Deploying 5 profiles...
[02:50:53 PM] Wrote MAR file to C:\Documents and Settings\rrc
[02:51:06 PM] Wrote Web Application Module to C:\Documents ar
[02:51:09 PM] Warning: ADF View integration with MDS not conf
[02:51:31 PM] Wrote Web Application Module to C:\Documents ar
[02:51:41 PM] Wrote EJB Module to C:\Documents and Settings\r
[02:51:44 PM] Warning: Some of the customization documents ur
[02:51:44 PM] Info: Any customizations created while running
```

However, while the server is running, when you access the editor via the Log window menu, then the editor has the ability to add transient loggers.

Figure 48-4 shows the Oracle Diagnostic Logging Configuration while the server is running.

Figure 48-4 Editor for Oracle Diagnostic Logging Configuration



Oracle Diagnostics Logging Configuration

Control logging behavior for specified loggers. If the server is running, changes take effect immediately. Any transient instance loggers you add exist only in the context of the running server, and are not persisted to the logging.xml file. You

Loggers:

Logger □ Hide Transient Loggers Java Log Levels

Name	Level	Declares Handlers
Root Logger (default)	WARNING	✓
Security		
com		
global		
javax		
jmxfmw		
oracle	WARNING	
oracle.adf		
oracle.adfdiagnostics	CONFIG	
oracle.adfdt		
oracle.adfdtinternal		
oracle.adfinternal		
oracle.as		
oracle.bali		
oracle.clickhistory		

Handler Declarations: "oracle.adfdiagnostics" Logger

Declare handlers to be used by this logger. Use the Source view to add and edit the handler definitions.

☑ Use Parent Handlers

Handlers:

No handlers declared for this logger

You can only use JDeveloper menu commands to open the `logging.xml` configuration file and launch the editor for Oracle Diagnostic Logging Configuration. However, you may find the following information about the location of the configuration file useful.

> **Note:**
>
> You can declare and add log handler definitions by clicking the **Source** tab and entering them in the XML editor.

If you are using Integrated WebLogic Server in JDeveloper on the Windows platform, you can find the `logging.xml` configuration file in a location similar to:

```
C:\Users\username\AppData\Roaming\JDeveloper\<your_JDev_system_folder>\Def
aultDomain\config\fmwconfig\servers\DefaultServer
```

The log files for Integrated WebLogic Server are in a location similar to:

```
C:\Users\username\AppData\Roaming\JDeveloper\<your_JDev_system_folder>\Def
aultDomain\servers\DefaultServer\logs
```

The log files for a standalone WebLogic Server instance are in a location similar to:

`$domain_home/servers/<your_servername>/logs`

You can configure logging levels before a test run from the Application Servers window, or, while the application is running, from the Log window's run panel toolbar, or, during a debug session, from the Log window's debug process toolbar.

To configure the log levels:

1. In the main menu, choose **Window** and then **Application Servers**.
2. In the Application Servers window, expand **Application Servers**, and right-click **IntegratedWebLogicServer** and choose **Configure Oracle Diagnostic Logging for "IntegratedWebLogicServer"**.

   Or, after you start the application, from the Log window's Running: IntegratedWebLogicServer panel, choose **Actions - Configure Oracle Diagnostic Logging.** The Running Log window is only visible while you run the application.

   Or, while you debug the application, from the Log window's debug process panel, choose **Actions - Configure Oracle Diagnostic Logging.** The debug process panel is only visible in the Log window after you have started the application in debug mode.
3. In the editor for Oracle Diagnostics Logging Configuration, select **ODL Log Levels** or **Java Log Levels** for the logger type you want to view.
4. If you want to see persistent loggers only, select **Hide Transient Loggers**.
5. To add a logger:
   a. If the server is running, click the **Add** icon dropdown menu and choose **Add Persistent Logger** or **Add Transient Logger**. If the server is not running, click **Add** to add a persistent logger. You cannot add a transient logger.
   b. In the Add Logger dialog, enter a logger name.
   c. Select the logging level.
   d. Click **OK**.
6. For any logger, including a newly created logger, you can specify its handlers by selecting from a list of available handlers by clicking the **Add** icon for the **Handler Declarations** section.

   Or, you can select **Use Parent Handlers** to assign its parent's handler to the logger. By default, a logger uses its parent's handler.

> **Note:**
>
> You can declare and add log handler definitions by clicking the **Source** tab and entering them in the XML editor.

How to Create an Oracle ADF Debugging Configuration

Oracle ADF leverages the Java Logging API (`java.util.logging.Logger`) to provide logging functionality when you run a debugging session. Java Logging is a standard

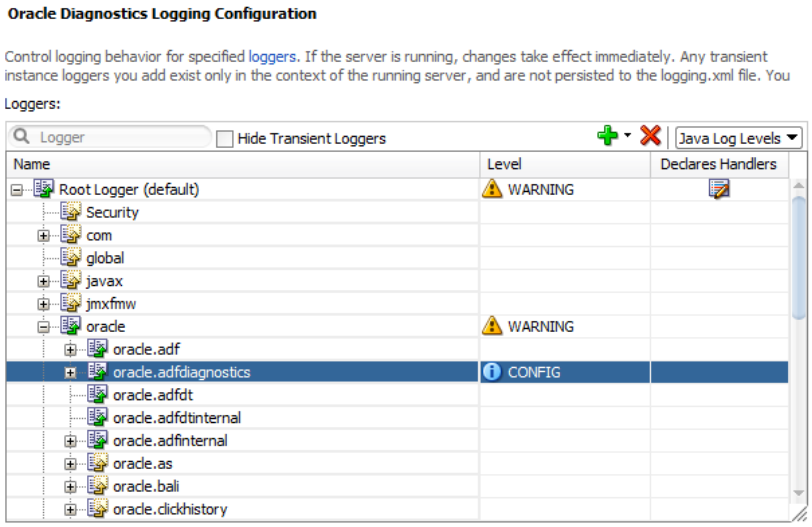
API that is available in the Java platform at http://docs.oracle.com/javase/7/docs/api/java/util/logging/package-summary.html.

To log a complete hierarchy tree of ADF event messages, you must configure a Java log level that is not more restrictive than `CONFIG` (or an ODL log level that is not more restrictive than `Notification:16`) for the single package:

- `oracle.adfdiagnostics` will log events generated by source code for the ADF Model data binding layer, ADF Controller source code, ADF Business Components, ADF Desktop Integration, and Oracle ADF internal classes.

Figure 48-5 shows specific the logger configured to enable the most detailed Oracle ADF log messages. Note that the `oracle.adfdiagnostics` package is set to CONFIG and not to a higher, more restrictive log level (where SEVERE, WARNING, and INFO are Java log levels higher than CONFIG and FINE, FINER, and FINEST are lower levels than CONFIG). Enabling a logger for the `oracle.adfdiagnostics` package ensures that the logger records a complete hierarchy of ADF events.

Figure 48-5 Configuring a Logger for Oracle ADF Debugging



Oracle Diagnostics Logging Configuration

Control logging behavior for specified loggers. If the server is running, changes take effect immediately. Any transient instance loggers you add exist only in the context of the running server, and are not persisted to the logging.xml file. You

Loggers:

Logger □ Hide Transient Loggers Java Log Levels

Name	Level	Declares Handlers
Root Logger (default)	WARNING	✓
Security		
com		
global		
javax		
jmxfmw		
oracle	WARNING	
oracle.adf		
oracle.adfdiagnostics	CONFIG	
oracle.adfdt		
oracle.adfdtinternal		
oracle.adfinternal		
oracle.as		
oracle.bali		
oracle.clickhistory		

For backward compatibility, the Java system property `jbo.debugoutput` set to the value `console` or `ADFLogger` (to route diagnostics through the standard Logger implementation, which can be controlled in a standard way through the `logging.xml` file) is supported. The easiest way to set this system property while running your application inside JDeveloper is to edit your project properties and in the Run/Debug page, select a run configuration and click **Edit**. Then add the string `-Djbo.debugoutput=console` to the **Java Options** field.

To create an ADF Model debugging configuration:

1. In the main menu, choose **Window** and then **Application Servers**.
2. In the Application Servers window, expand **Application Servers**, and right-click **IntegratedWebLogicServer** and choose **Configure Oracle Diagnostic Logging**.

   Or, after you start the application, from the Log window's Running: IntegratedWebLogicServer panel, choose **Actions - Configure Oracle Diagnostic Logging.** The Running Log window is only visible while you run the application.

Or, while you debug the application, from the Log window's debug process panel, choose **Actions - Configure Oracle Diagnostic Logging.** The debug process panel is only visible in the Log window after you have started the application in debug mode.

3. In the editor for Oracle Diagnostics Logging Configuration, select **ODL Log Levels** or **Java Log Levels** for the logger type you want to view.
4. Expand the **oracle** node and select a log level that are not more restrictive than `CONFIG` (for Java Log) or that are not more restrictive than `Notification` (for ODL Log) for the following package:
   - **oracle.adfdiagnostics** will log events generated by source code for the ADF Model data binding layer, ADF Controller source code, ADF Business Components, and Oracle ADF internal classes.

     Enabling a logger for this package ensures that the logger records a complete hierarchy of ADF events.
5. Select the Java log level **FINEST** for other desired Oracle ADF loggers.

To create an ADF view Javascript logging configuration:

1. In the Applications window, double-click the application or project **web.xml** file.
2. In the source editor, add the following elements to the file:

```
<context-param>
  <param-name>
     oracle.adf.view.rich.LOGGER_LEVEL
  </param-name>
  <param-value>
      FINE
  </param-value>
</context-param>
```

How to Turn On Diagnostic Logging for Non-Oracle ADF Loggers

For non-Oracle ADF loggers, before you use the actual debugger, running the application with Java log level set to FINE, FINER, or FINEST will enable framework diagnostics logging. The debug diagnostic messages can be helpful to see what happens when the problem occurs. To enable debug diagnostic messages, use the Oracle Diagnostics Logging Configuration Editor to configure the desired loggers with a Java log level set to FINEST. If you have configured diagnostic logging for the supported loggers, JDeveloper will direct debug diagnostics messages to the JDeveloper Log window.

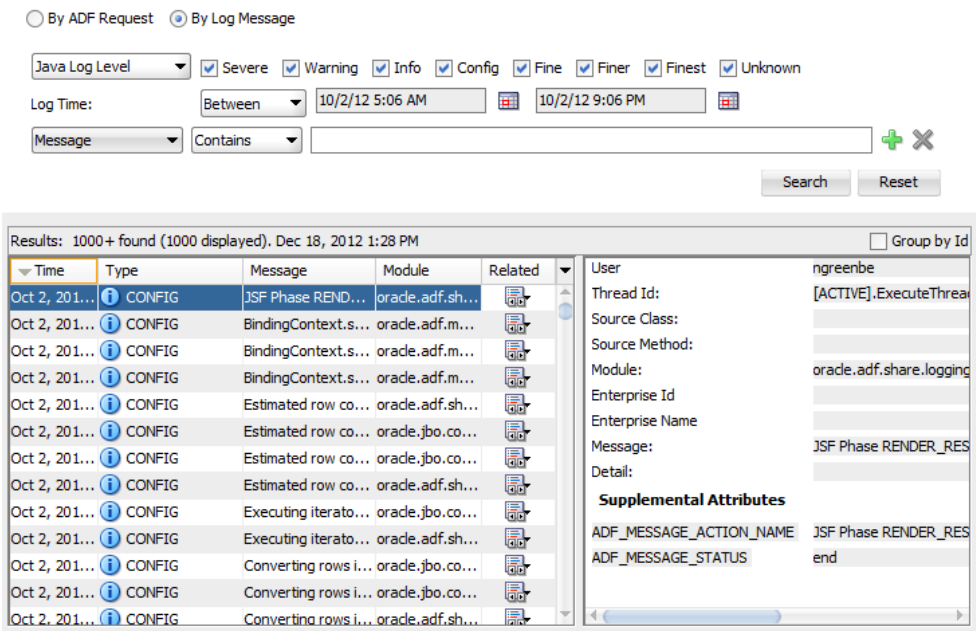
Currently, Oracle ADF does not support logging with Java log level FINE, FINER, or FINEST. For more details about Oracle ADF loggers, see How to Create an Oracle ADF Debugging Configuration.

How to Use the Log Analyzer to View Log Messages

You can use Oracle Diagnostic Log Analyzer to view the log entries of a log file. The log analyzer allows you to filter the entries by log level, entry type, log time, and entry content (using one or more search criteria). You can also order the messages and show and hide columns for better viewing.

Figure 48-6 shows Oracle Diagnostic Log Analyzer set to view Java Log levels.

Figure 48-6 Oracle Diagnostic Log Analyzer Displays Java Log Messages



○ By ADF Request ● By Log Message

Java Log Level ☑ Severe ☑ Warning ☑ Info ☑ Config ☑ Fine ☑ Finer ☑ Finest ☑ Unknown

Log Time: Between 10/2/12 5:06 AM 10/2/12 9:06 PM

Message Contains

Results: 1000+ found (1000 displayed). Dec 18, 2012 1:28 PM □ Group by Id

Time	Type	Message	Module	Related
Oct 2, 201...	CONFIG	JSF Phase REND...	oracle.adf.sh...	
Oct 2, 201...	CONFIG	BindingContext.s...	oracle.adf.m...	
Oct 2, 201...	CONFIG	BindingContext.s...	oracle.adf.m...	
Oct 2, 201...	CONFIG	BindingContext.s...	oracle.adf.m...	
Oct 2, 201...	CONFIG	Estimated row co...	oracle.adf.sh...	
Oct 2, 201...	CONFIG	Estimated row co...	oracle.jbo.co...	
Oct 2, 201...	CONFIG	Estimated row co...	oracle.jbo.co...	
Oct 2, 201...	CONFIG	Estimated row co...	oracle.adf.sh...	
Oct 2, 201...	CONFIG	Executing iterato...	oracle.jbo.co...	
Oct 2, 201...	CONFIG	Executing iterato...	oracle.adf.sh...	
Oct 2, 201...	CONFIG	Converting rows i...	oracle.jbo.co...	
Oct 2, 201...	CONFIG	Converting rows i...	oracle.jbo.co...	
Oct 2, 201...	CONFIG	Converting rows i...	oracle.adf.sh...	

User	ngreenbe
Thread Id:	[ACTIVE].ExecuteThrea
Source Class:	
Source Method:	
Module:	oracle.adf.share.logging
Enterprise Id	
Enterprise Name	
Message:	JSF Phase RENDER_RES
Detail:	

Supplemental Attributes

ADF_MESSAGE_ACTION_NAME	JSF Phase RENDER_RES
ADF_MESSAGE_STATUS	end

You can also use the log analyzer on log files created in other test runs. For instance, you can analyze the log sent to you by another developer for another application.

Viewing Diagnostic Messages in the Log Analyzer

You can configure logging levels before a test run from the Application Servers window or during a debug session from the Log window toolbar. The level you specify will determine the type and quantity of log messages.

In the case of ADF events, all messages are generated for the ODL log at the level `Notification` or for the Java log at the level `Info`. Fewer ADF messages will be generated at the `Incident Error/Severe` and `Error/Warning` levels.

After you select the log level for the messages you wish to view, you can use the Search panel of the By Log Message page to filter the messages to display from the log file.

> **Note:**
>
> For further details about search criteria that you can specify to search on ADF-specific messages, see Sorting Diagnostic Messages By ADF Events.

You can start the log analyzer before a test run from the **Tools** menu or during a debug session from the log window toolbar.

Before you begin:

It may be helpful to have an understanding of logging. For more information, see Using the ADF Logger.

You will need to complete these tasks:

1. Set logging levels, as described in How to Set ADF Logging Levels.
2. Enable logging, as described in How to Turn On Diagnostic Logging for Non-Oracle ADF Loggers.
3. Create a log file, either from your test run or from another source.

To start the log analyzer:

1. In the main menu, choose **Tools** and then **Oracle Diagnostic Log Analyzer**.

   Or, from the Log window **Action** menu, choose **Analyze Log** and then either **Current in Console** or **Open Selected** (to browse log files in the server log directory).
2. In the editor for Oracle Diagnostic Log Analyzer, select **By Log Message**.
3. In the By Log Message page, in the **Log** field, enter the path and name of the log file or click **Browse Log Files** to navigate to the log file.

   > **Tip:**
   >
   > The Choose Log File dialog helps you to navigate to the directory that contains the log files generated by JDeveloper. Click the **Browse Log Files** icon next to the **Log** text field, and then click the **Server Logs** icon from the scroll list. From the list of log files, you can select more than one log file to analyze at a time.
4. From the dropdown list, select either **ODL Log Level** or **Java Log Level**.
5. Select the corresponding checkbox for each type of log entry you want to view. You must select at least one type.

   The available ODL log level types are:

   - INCIDENT ERROR
   - ERROR
   - WARNING
   - NOTIFICATION - corresponds to ADF event messages
   - TRACE
   - UNKNOWN.

   The available Java log level types are:

   - SEVERE
   - WARNING
   - INFO
   - CONFIG - corresponds to ADF event messages
   - FINE
   - FINER
   - FINEST
   - UNKNOWN

6. Specify a time period for the entries you want to view. You can select the most recent *period* or a range.
7. To filter the results, use the Search panel to query the log for a text pattern. For additional search criteria, click **Add**. The supported search criteria include:
   - **Enterprise Name:** Filters the log by the name of the enterprise.
   - **Detail:** Filters text in statements from the stack where the method was invoked.
   - **EnterpriseID:** Filters the log by the ID of the enterprise.
   - **Message:** Filters text in the logged messages.
   - **ADF Message Data:** Filters the log for data related to ADF lifecycle phase names, **view object** names, view object query statements, data control names, binding container names, and iterator binding names logged during the execution of ADF events.
   - **Source Method:** Filters the log by the method where the message is logged. For example, you can filter on the method `execute` to view all messages logged for view object query execution or ADF lifecycle phase execution.
   - **Application:** Filters the log by the application name where the message is logged. This is useful when the application is running in a composite application and you want to view messages for a specific application.
   - **Source Class:** Filters the log by the fully qualified class name of the method where the message is logged. To see more messages, enter a partial package name. For example, you can enter the partial package name `oracle.adf` or the full package name `oracle.jbo` to filter for all classes related to Oracle ADF.
   - **Module:** Filters the log by the fully qualified package name of the class where the message is logged. This is same package as the source class.
   - **Message Id:** Filters the log by the ID of the logged messages. Many messages share the same ID. For example, message ID `ADFC-52008` might have four `INFO` messages and one `Warning` message. You can select **Group by Id** in the log analyzer Results panel to group messages by their common ID.
8. To initiate the filters and display the log messages, click **Search**.
9. To order the results by the message ID, select the **Group by Id** checkbox.
10. To group the messages by time period or by request, in the **Related** column, select either **Related by Time** or **Related by Request**.

    When you select **Related by** option, the log analyzer groups the messages in the Results Messages panel according to the selected criteria (time, HTTP request, or enterprise name).
11. To open the ADF request (if any) that the message is a part of, in the **Related** column, select **Related by ADF Request**.

    When you select **Related by ADF Request**, the log analyzer displays the related request in the By ADF Request page.
12. To show or hide columns in the **Results** section, click the dropdown list to the right of the column headers and select among the list of displayed columns to change the visibility of a column.

Using the Log Analyzer to Analyze the ADF Request

Because Oracle instrumented the Oracle ADF source code to generate log messages during the execution of the ADF lifecycle phases, you can use the log analyzer to investigate the details of the active (or previous) page request in your running application. Specifically, the By ADF Request page of the log analyzer lets you view ADF event messages in a hierarchical list, organized by the sequence of their execution. It also provides a graphical representation of the duration of each event. When you run your application and start the log analyzer with ADF logging configured, you can use this page to quickly identify whether a component of your application is contributing to a performance bottleneck due to unusually long execution times.

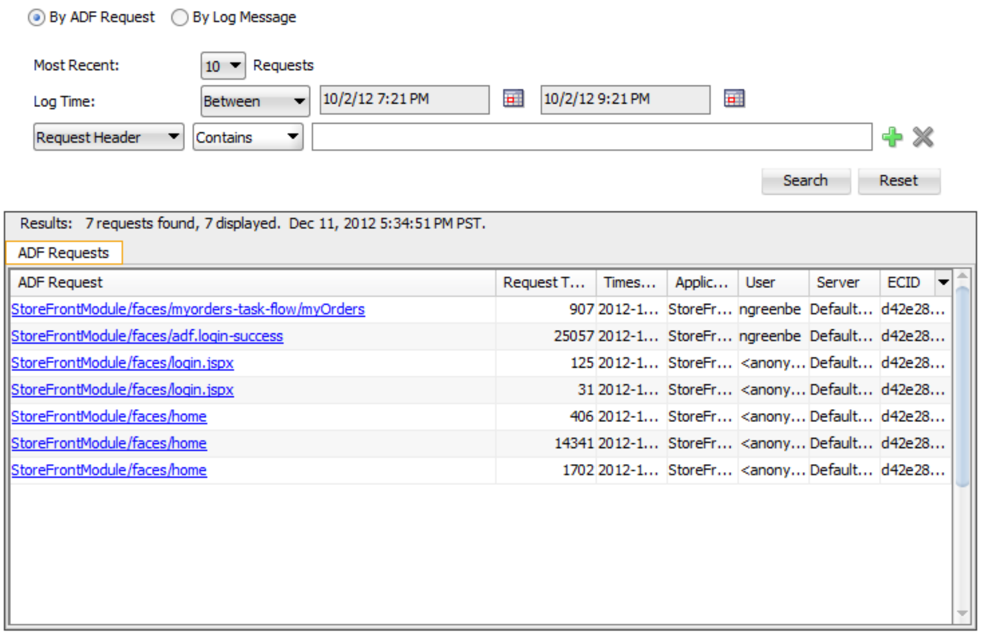
> **Note:**
>
> In contrast to the By Log Message page, the By ADF Request page of the log analyzer displays a hierarchical view of ADF event messages. The difference between these two pages is that the By ADF Request page focuses only on ADF page requests made when a page or **region** is submitted. For details about the ADF page lifecycle, see Understanding the Fusion Page Lifecycle .

You can enter search criteria in the By ADF Request page to display one or more specific requests from the log file. You can combine any of the following search criteria to filter the log file:

- The number of requests to display
- The timestamp for the request recorded in the log
- The logged-in user name, enterprise name, request header (to specify a search based on the page or region name), message, source class, module, server, detail, enterprise ID, ADF message data, application name in a composite application, source method, message ID, or ECID (a globally unique ID associate with every top-level task; sub-tasks invoked by that top-level task operate with an execution context that shares the same ECID)

After the request is completed, the log analyzer displays a summary of the ADF requests in a single table. Figure 48-7 shows the summary table that displays the list of ADF web requests.

Figure 48-7 Oracle Diagnostic Log Analyzer Displays Summary of ADF Web Requests



● By ADF Request ○ By Log Message

Most Recent: 10 Requests

Log Time: Between 10/2/12 7:21 PM 10/2/12 9:21 PM

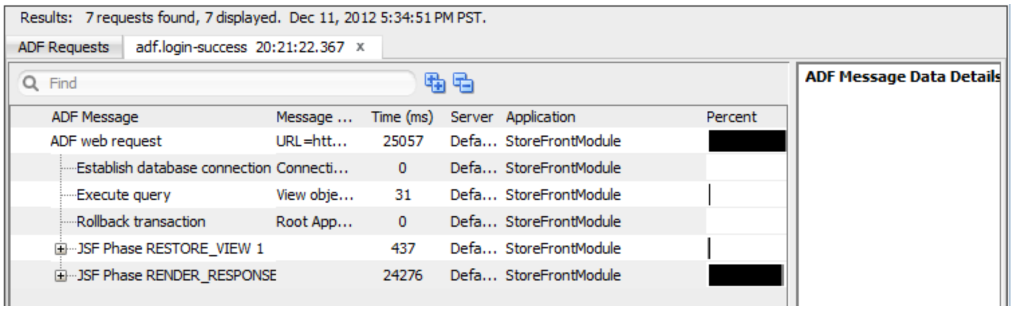
Request Header Contains

Results: 7 requests found, 7 displayed. Dec 11, 2012 5:34:51 PM PST.

ADF Request	Request T...	Times...	Applic...	User	Server	ECID
StoreFrontModule/faces/myorders-task-flow/myOrders	907	2012-1...	StoreFr...	ngreenbe	Default...	d42e28...
StoreFrontModule/faces/adf.login-success	25057	2012-1...	StoreFr...	ngreenbe	Default...	d42e28...
StoreFrontModule/faces/login.jspx	125	2012-1...	StoreFr...	<anony...	Default...	d42e28...
StoreFrontModule/faces/login.jspx	31	2012-1...	StoreFr...	<anony...	Default...	d42e28...
StoreFrontModule/faces/home	406	2012-1...	StoreFr...	<anony...	Default...	d42e28...
StoreFrontModule/faces/home	14341	2012-1...	StoreFr...	<anony...	Default...	d42e28...
StoreFrontModule/faces/home	1702	2012-1...	StoreFr...	<anony...	Default...	d42e28...

For each ADF request displayed in the summary table, you can click the request link to view details about the request event hierarchy. Figure 48-8 shows the ADF Requests panel with the JSF lifecycle Restore View and Render Response phases displayed and their duration in milliseconds. The bar graph for the root node of the request event hierarchy (**ADF web request**) displays the total execution time.

Figure 48-8 Oracle Diagnostic Log Analyzer Displays ADF Web Request Event Hierarchy



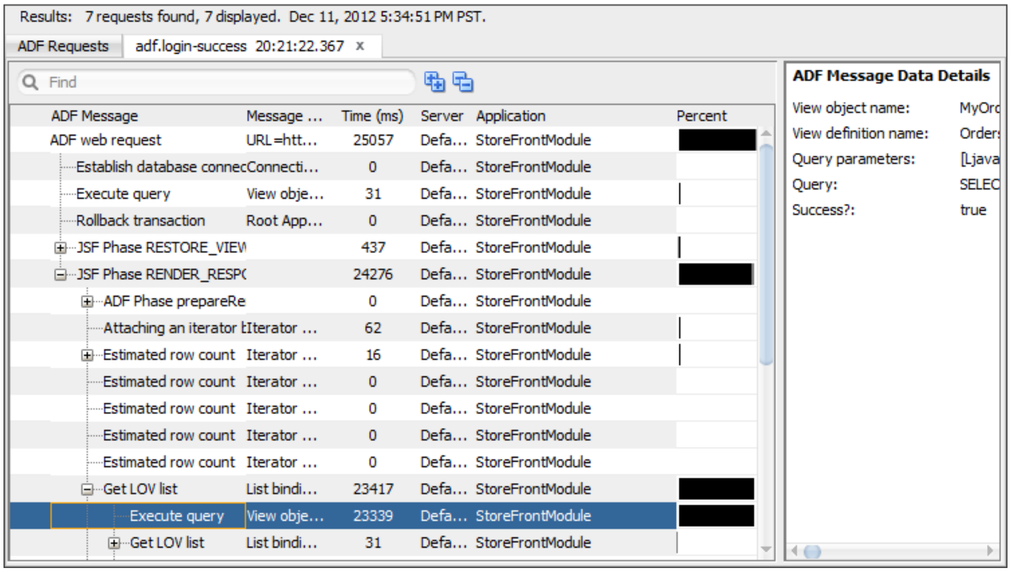
Results: 7 requests found, 7 displayed. Dec 11, 2012 5:34:51 PM PST.

ADF Message	Message ...	Time (ms)	Server	Application	Percent
ADF web request	URL=htt...	25057	Defa...	StoreFrontModule	█
Establish database connection	Connecti...	0	Defa...	StoreFrontModule	
Execute query	View obje...	31	Defa...	StoreFrontModule	|
Rollback transaction	Root App...	0	Defa...	StoreFrontModule	
⊞ JSF Phase RESTORE_VIEW 1		437	Defa...	StoreFrontModule	|
⊞ JSF Phase RENDER_RESPONSE		24276	Defa...	StoreFrontModule	█

> **Note:**
>
> The **Percentage Request Time** bar graphs (black and gray) indicate which portion of the request's execution time resulted from ADF source code that was instrumented to generate ADF event messages (shown in black) and which portion resulted from ADF source code that is uninstrumented and therefore cannot generate ADF event messages (shown in gray). Additionally, note that the individual phases of the request do not sum to equal the total request time. This is due to the fact that only those phases of the lifecycle that are useful are represented in the log analyzer.

To examine the request in more detail, you can expand the tree for any ADF lifecycle node to further investigate where in the application the performance bottleneck occurred. Drilling down and then selecting the ADF event node in the request message panel gives you details about the component associated with each ADF event. For instance, expanding the **JSF Phase RENDER_RESPONSE** node displays all ADF events generated during that phase. Figure 48-9 shows the **JSF Phase RENDER_RESPONSE** node expanded with a long request duration bar graph for the **Get LOV list** node and the **Execute query** node. The **Execute query** node has been selected to reveal detailed ADF data in the ADF Message Data Details panel, including the view object's name and query statement. By drilling down and selecting the ADF event with the long execution time as indicated by the bar graph, you can obtain, for example, the name of the view object in the data model project that should be tuned for improved performance.

Figure 48-9 Oracle Diagnostic Log Analyzer Displays ADF Event Messages with ADF Data



Results: 7 requests found, 7 displayed. Dec 11, 2012 5:34:51 PM PST.

ADF Message	Message ...	Time (ms)	Server	Application	Percent
ADF web request	URL=htt...	25057	Defa...	StoreFrontModule	■
Establish database connec	Connecti...	0	Defa...	StoreFrontModule	
Execute query	View obje...	31	Defa...	StoreFrontModule	|
Rollback transaction	Root App...	0	Defa...	StoreFrontModule	
⊞ JSF Phase RESTORE_VIEW		437	Defa...	StoreFrontModule	|
⊟ JSF Phase RENDER_RESP(		24276	Defa...	StoreFrontModule	■
⊞ ADF Phase prepareRe		0	Defa...	StoreFrontModule	
Attaching an iterator t	Iterator ...	62	Defa...	StoreFrontModule	|
⊞ Estimated row count	Iterator ...	16	Defa...	StoreFrontModule	|
Estimated row count	Iterator ...	0	Defa...	StoreFrontModule	
Estimated row count	Iterator ...	0	Defa...	StoreFrontModule	
Estimated row count	Iterator ...	0	Defa...	StoreFrontModule	
Estimated row count	Iterator ...	0	Defa...	StoreFrontModule	
⊟ Get LOV list	List bindi...	23417	Defa...	StoreFrontModule	■
Execute query	View obje...	23339	Defa...	StoreFrontModule	■
⊞ Get LOV list	List bindi...	31	Defa...	StoreFrontModule	|

ADF Message Data Details

View object name:	MyOrd
View definition name:	Orders
Query parameters:	[Ljava
Query:	SELEC
Success?:	true

Before you begin:

It may be helpful to have an understanding of logging. For more information, see Using the ADF Logger.

You will need to complete these tasks:

1. Set logging levels, as described in How to Set ADF Logging Levels.

   To log a complete hierarchy tree of ADF event messages, you must configure an ODL log level that is not more restrictive than `NOTIFICATION:16` or a Java log level that is not more restrictive than `CONFIG` for the following single package:

   - `oracle.adfdiagnostics` will log events generated by source code for the ADF Model data binding layer, ADF Controller source code, ADF Business Components, and Oracle ADF internal classes.

   > **Tip:**
   >
   > The default log level for the Root Logger displayed by the editor for Oracle Diagnostics Logging Configuration ensures that ADF event messages are logged.

2. Enable logging, as described in How to Turn On Diagnostic Logging for Non-Oracle ADF Loggers.
3. Create a log file, either from your test run or from another source.

To display ADF request messages in the log analyzer:

1. In the main menu, choose **Tools** and then **Oracle Diagnostic Log Analyzer**.

   Or, from the Log window **Action** menu, choose **Analyze Log** and then either **Current in Console** or **Open Selected** (to browse log files in the server log directory).

2. In the editor for Oracle Diagnostic Log Analyzer, click the **By ADF Request** tab.
3. In the By ADF Request page, specify how many of the most recent request you want to display.

   The default displays only the most recent request.

4. Specify a time period for the entries you want to view. You can select the most recent *period* or a range.
5. To filter the request to display, use the search fields to query the log for a text pattern. For additional search fields, click the **Add Row** icon.
6. To initiate the filters and display matching ADF requests, click **Search**.
7. In the ADF Requests panel, click the desired request link to view the event hierarchy and look for ADF events that display long execution times as indicated by the **Time (ms)** bar graphs.
8. Select the desired ADF event and examine the ADF Message Data Details panel for details about the ADF component associated with the ADF event.
9. Examine the component in your application and determine whether optimization is possible.

Sorting Diagnostic Messages By ADF Events

Oracle instrumented the Oracle ADF source code to generate log messages during the execution of the ADF lifecycle phases and during operations executed in the ADF Model data binding layer, ADF Controller source, and ADF Business Components source. Combined, the log analyzer refers to these messages as ADF events. You can use the log analyzer to investigate ADF events in your running application. The By Log Message page of the log analyzer lets you view ADF event messages in a flat list, organized by time of execution, with the option to switch to the By ADF Request page to view the ADF events in a hierarchical list, organized by the sequence of their execution.

ADF event messages contain useful information that helps you identify which ADF components in your application generated the event. For example, you can search the log for ADF event messages to identify the components related to displaying data in the page, executing queries, or initiating actions:

- Executing iterator binding: Displays the names of the iterators executed to manage displaying data in the page. This can be useful for diagnosing slow query updates.
- Execute query: Displays the name of the view object associated with the executed query. This can be useful when you want to view the query statement, bind parameters, and name of the view object.
- Executing method binding: Displays the names of the Java methods executed on the bound data source. This can be useful for diagnosing slow method execution.

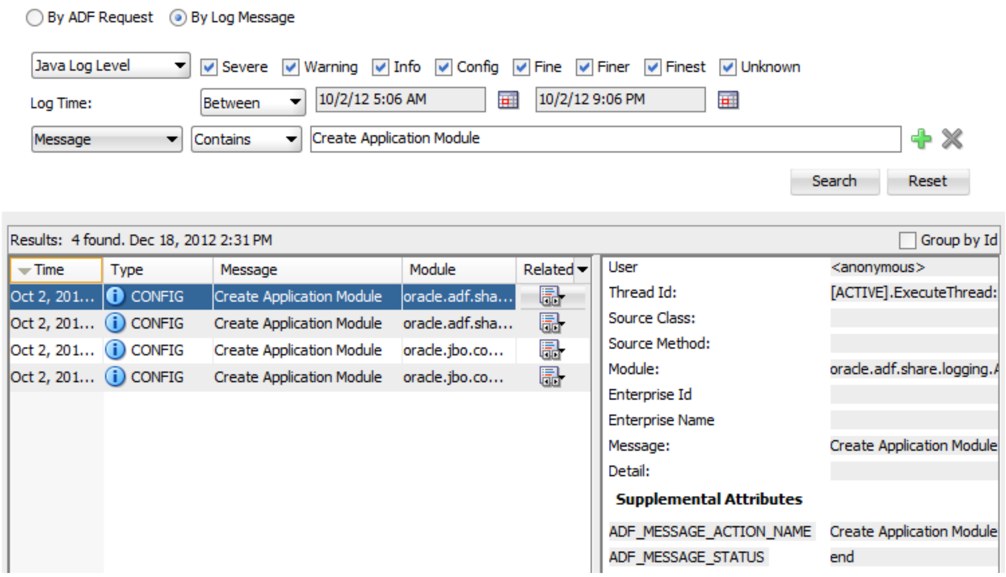
After you display an ADF event message in the log analyzer, you can organize the event in the context of other logged messages. You can select options from the **Related** column to display:

- All messages leading up to the ADF event (related by time)
- All messages in the same web request as the ADF event (related by request)
- Only ADF event messages in the same web request (related by ADF request)

> **Tip:**
>
> The **Related by ADF Request** option displays detailed ADF data for the ADF event messages. This is the view to use, for example, when you want to display the query statement associated with the `Execute query` message.

Figure 48-10 shows the log analyzer search result for the ADF event message `Create Application Module`. The Results panel displays all messages that match the search criteria and the bottom panel displays detailed information about the component.

Figure 48-10 Oracle Diagnostic Log Analyzer Displays ADF Event Messages



○ By ADF Request ● By Log Message

Java Log Level ☑ Severe ☑ Warning ☑ Info ☑ Config ☑ Fine ☑ Finer ☑ Finest ☑ Unknown

Log Time: Between 10/2/12 5:06 AM 10/2/12 9:06 PM

Message Contains Create Application Module

Search Reset

Results: 4 found. Dec 18, 2012 2:31 PM □ Group by Id

Time	Type	Message	Module	Related
Oct 2, 201...	CONFIG	Create Application Module	oracle.adf.sha...	
Oct 2, 201...	CONFIG	Create Application Module	oracle.adf.sha...	
Oct 2, 201...	CONFIG	Create Application Module	oracle.jbo.co...	
Oct 2, 201...	CONFIG	Create Application Module	oracle.jbo.co...	

User	<anonymous>
Thread Id:	[ACTIVE].ExecuteThread:
Source Class:	
Source Method:	
Module:	oracle.adf.share.logging.A
Enterprise Id	
Enterprise Name	
Message:	Create Application Module
Detail:	
Supplemental Attributes	
ADF_MESSAGE_ACTION_NAME	Create Application Module
ADF_MESSAGE_STATUS	end

When you select **Related by ADF Request** in the **Related** column of the Results panel, the log analyzer switches to display the By ADF Request page with the ADF event messages arranged hierarchically to show their execution dependencies. The By ADF Request page of the log analyzer is the preferred way to diagnose performance issues. For details about the By ADF Request page, see Using the Log Analyzer to Analyze the ADF Request. In the By Log Message page, the elapsed time is information that you can leave visible or hide from the Results panel.

Before you begin:

It may be helpful to have an understanding of logging. For more information, see Using the ADF Logger.

You will need to complete these tasks:

1. Set logging levels, as described in How to Set ADF Logging Levels.

   To log ADF event messages, do not configure an ODL log level that is more restrictive than `NOTIFICATION:16` or a Java log level that is more restrictive than `CONFIG` for the following package:

   - `oracle.adfdiagnostics` will log events generated by source code for the ADF Model data binding layer, ADF Controller source code, ADF Business Components, and Oracle ADF internal classes.

   > **Tip:**
   >
   > The default log level for the Root Logger displayed by the editor for Oracle Diagnostics Logging Configuration ensures ADF event messages are logged.

2. Enable logging, as described in How to Turn On Diagnostic Logging for Non-Oracle ADF Loggers.

3. Create a log file, either from your test run or from another source.

To display messages related by ADF events:

1. In the main menu, choose **Tools** and then **Oracle Diagnostic Log Analyzer**.

   Or, from the Log window **Action** menu, choose **Analyze Log** and then either **Current in Console** or **Open Selected** (to browse log files in the server log directory).

2. In the editor for Oracle Diagnostic Log Analyzer, click the **By Log Message** tab.

3. In the By Log Message page, select the desired logger type, log levels, and log time.

   To search the log for ADF event messages, you must minimally select log level **Notification** (for ODL log level) or **Info** (for Java log level).

4. In the search dropdowns, choose the search criteria **Message** and **Contains**, and then enter any of the following ADF event messages and click **Search**.

   - `Executing iterator binding` - this can be useful for diagnosing slow query updates.
   - `Executing method binding` - this can be useful for diagnosing slow method execution.
   - `Execute query` - this can be useful when you want to view the query statement, bind parameters, and name of the view object.

   You can also filter the log on these additional ADF event messages:

   - `Refreshing binding container`
   - `Attaching an iterator binding to a datasource`
   - `Converting rows into hierarchical nodes`
   - `Estimated row count`
   - `Get LOV list`
   - `Filter LOV list`
   - `Validate Entity`
   - `Lock Entity's Parent`
   - `Lock Entity`
   - `Before posting the entity's changes`
   - `Posting the entity's changes`
   - `Posting in batches`
   - `Before committing the entity's changes`
   - `After committing the entity's changes`
   - `Before rolling back the entity's changes`
   - `After rolling back the entity's changes`
   - `Entity notifying an event`
   - `Entity notification name`
   - `Removing Entity`

- `Updating audit columns`
- `Applying Effective Date change`
- `Entity DML`
- `Entity read all attributes`
- `Create Application Module`
- `Create nested Application Module`
- `Passivating Application Module`
- `Activating Application Module`
- `Establish database connection`
- `Commit transaction`
- `Rollback transaction`
- `Validate transaction`
- `Validate value`

Examine the data portion of the Results panel for the ADF event information.

5. To view a hierarchical sequence of ADF events, with the desired ADF event message selected in the Results panel, in the **Related** column, click the icon for the desired event row and choose **Related By ADF Request** from the dropdown menu.

   The editor for Oracle Diagnostic Log Analyzer displays the By ADF Request page for the selected ADF event. Examine the bottom portion of the Results panel for additional ADF data for the ADF event. For example, you can see the query statement associated with the `Execute query` message in the ADF Data area of the Results panel.

What You May Need to Know About ADF Loggers and Log Levels

By default, the level is set to `WARNING` for all Oracle loggers. For Oracle ADF packages set `level="FINE"` for detailed logging diagnostics.

For the ADF view layer packages `oracle.adf.view.faces` and `oracle.adfinternal.view.faces`, edit these elements:

```
<logger name="oracle.adf" level="FINE"/>
<logger name="oracle.adfinternal" level="FINE"/>
```

For the ADF Model layer packages, edit these elements:

```
<logger name="oracle.adf" level="FINE"/>
<logger name="oracle.jbo" level="FINE"/>
```

For the **ADF Controller** layer packages, edit these elements:

```
<logger name="oracle.adf.controller" level="FINE"/>
<logger name="oracle.adfinternal.controller" level="FINE"/>
```

Alternatively, you can create a debug configuration in JDeveloper that you can choose when you start a debugging session.

The following example shows the portion of the `logging.xml` file where you can change the granularity of the log messages. Note in the example that the log for `oracle.adf.faces` has been changed to `FINE` to display more messages.

```
</logging_configuration>
...
  <loggers>
      <logger name="oracle.adf" level="INFO"/>
      <logger name="oracle.adf.faces" level="FINE"/>
      <logger name="oracle.adf.controller" level="INFO"/>
      <logger name="oracle.bc4j" level="INFO"/>
      <logger name="oracle.adf.portal" level="INFO"/>
      <logger name="oracle.vcr" level="INFO"/>
      <logger name="oracle.portlet" level="INFO"/>
      <logger name="oracle.adfinternal" level="INFO"/>
      <logger name="oracle.adfdt" level="INFO"/>
      <logger name="oracle.adfdtinternal" level="INFO"/>
   </loggers>
</logging_configuration>
```

For the latest information about the different levels of the Java Logging system, go to `http://www.oracle.com/technetwork/java/index.html`. Normally, the Java logging system supports the following log levels:

- SEVERE (most restrictive, highest log level)
- WARNING
- INFO
- CONFIG (the highest low level supported for Oracle ADF)
- FINE
- FINER
- FINEST (least restrictive, lowest log level)

What You May Need to Know About ADF Logging and Log Output

By default, loggers are hierarchical, so a logger that is a child of another logger will inherit the log level of its parent and may generate an unexpectedly large log output. As a result, when you enable loggers in the editor for Oracle Diagnostic Logging Configuration, you may want to set one log level for the parent logger and another log level for child loggers. This is especially important when you use settings `FINE`, `FINER`, `FINEST` that can be useful to diagnosis certain conditions like memory leaks, where detailed log messages are beneficial.

For example, when debugging memory leaks, you might set the `oracle.adf.share.ADFContext` logger to `FINEST`. This setting will automatically trigger the child `oracle.adf.share.ADFContext.allocationLogger` logger and may result in an allocation buffer exceeded condition for the log output file. Therefore, the solution to enable the messages needed to fix memory leaks is to make the parent level `FINEST`, while turning down the child logger to `INFO`.

```
<logger name="oracle.adf.share.ADFContext" level="FINE"/>
<logger name="oracle.adf.share.ADFContext.allocationLogger" level="INFO"/>
```

What You May Need to Know About ADF Logging and Oracle WebLogic Server

After you have deployed the Fusion web application to Oracle WebLogic Server, the operations performed by the application are logged directly to the Managed Server where the application is running:

`DOMAIN_HOME/servers/`*`server_name`*`/logs/`*`server_name`*`-diagnostic.log`

The log files for the different Managed Servers are also available from the Oracle WebLogic Server Administration Console. To verify the logs, access the Oracle WebLogic Server Administration Console `http://<admin_server_host>:<port>/console` and click **Diagnostics-Log Files**.

This log's granularity and logging properties can be changed using Oracle Enterprise Manager Fusion Middleware Control (Fusion Middleware Control). Fusion Middleware Control is a web browser-based, graphical user interface that you can use to monitor and administer a farm.

When the Fusion web application is deployed to a high availability environment, you can receive warning diagnostic messages specific to high availability by setting the level to `FINE`.

For details about using Fusion Middleware Control to change the log settings of Managed Servers and Oracle ADF, see Understanding ODL Messages and ODL Log Files in *Administering Oracle Fusion Middleware*.

Using the Oracle ADF Model Tester for Testing and Debugging

You can use the Oracle ADF Model Tester to test and validate your ADF Business Components even before you have started to build any custom user interface.

The Oracle ADF Model Tester (also referred to as the tester) is a Java application that you launch from JDeveloper when you want to interact with the business objects of the ADF Business Components data model project. The Oracle ADF Model Tester runs outside of JDeveloper and provides a full UI for testing and examining the data model project. You can run the tester to examine the view instances of the ADF **application module**, navigate the hierarchical relationship of view links, and execute custom methods from the application module's client interface, view object interface, and view row interface. The tester also interacts with the ADF Declarative Debugger to allow you to set breakpoints on the custom methods of these interfaces.

Additionally, the tester simulates many features that the user interface might expose by allowing you to view, insert, and update the contents of business objects in the database specified by the application module's configuration file (`bc4j.xcfg`). Specifically, you can use the tester to verify many aspects of the data model design, including a **master-detail relationship** between view instances, view instances and their attributes, view instance query result sets, search forms using **view criteria**, validation rules defined for attribute values, and dropdown lists on LOV-defined attributes (list of values). For more information about ways to interact with the tester to test your business objects, see Testing View Object Instances Using the Oracle ADF Model Tester. Additional information about testing with the Oracle ADF Model Tester also appears in sections specific to each business object throughout the chapters in the Building Your Business Services part of this book.

How to Run in Debug Mode and Test with the Oracle ADF Model Tester

Often you will find it useful to analyze and debug custom code in the service methods of your client interface implementation classes. When you use the Oracle ADF Model Tester, you can do this without needing to run the application with the user interface. You can use the Oracle ADF Model Tester as a testing tool to complement your debugging process.

> **Note:**
>
> The Oracle ADF Model Tester that you run in debug mode will not inherit your JDeveloper IDE Java options. To ensure that specific run/debug Java options are used with the tester, you must edit the run configuration for the data model project. You can modify the default run confirmation in the Run/ Debug page of the Project Properties dialog.

Before you begin:

It may be helpful to have an understanding of the using the runtime testing tools. For more information, see Using the Oracle ADF Model Tester for Testing and Debugging.

You will need to complete these tasks:

1. Set the desired Java options for your preferred run configuration, as described in the Running and Debugging Java Projects chapter in *Developing Web User Interfaces with Oracle ADF Faces*.
2. Set breakpoints in the custom methods of your client interface, as described in Setting ADF Declarative Breakpoints.

To launch the Oracle ADF Model Tester and go into debug mode:

1. In the Applications window, right-click the desired application module and choose **Debug**.
2. In the Oracle ADF Model Tester, open the method testing panel for the desired client interface, as described in How to Test Custom Service Methods Using the Oracle ADF Model Tester.
3. In the method panel, select the desired method from the dropdown list, enter values to pass as method parameters, and click **Execute**.

   Return to JDeveloper to step through your code using the ADF Declarative Debugger. When you complete method execution, the method panel displays the return value (if any) and test result. The result displayed in the Oracle ADF Model Tester will indicate whether or not the method executed successfully.

How to Run the Oracle ADF Model Tester and Test with a Specific Application Module Configuration

When you right-click the application module in the Applications window and choose **Run** or **Debug**, JDeveloper will run the Oracle ADF Model Tester using the default configuration defined for the application module. If you want to test your business components with a different application module configuration (which can specify a different data source and its own set of runtime parameters), you can do so from the overview editor for the `bc4j.xcfg` file. The file is not visible in the Applications window and is only accessible from the Configurations page of the overview editor for the application module.

To run the Oracle ADF Model Tester with a specific application module configuration:

1. In the Applications window, double-click the application module that you want to test.
2. In the overview editor, click the **Configurations** navigation tab and then click the **bc4j.xcfg** configuration file link.
3. In the overview editor, select the configuration from the **Configurations** list.
4. Right-click the selected configuration and choose **Run** or **Debug** to launch the Oracle ADF Model Tester.

What Happens When You Run the Oracle ADF Model Tester in Debug Mode

The Oracle ADF Model Tester behaves as any other Java program. Specifically, it does not inherit Java options that you may have specified for the JDeveloper IDE. The tester instead uses the run configuration that you have specified for the data model project and any Java options that you may have set in that configuration.

JDeveloper lets you run the Oracle ADF Model Tester in two modes: either in debug mode or non-debug mode. When run in debug mode, the tester interacts with the ADF Declarative Debugger so that you execute custom methods using breakpoints you insert in custom Java code and Groovy script of the client interfaces. For instance, if you set a breakpoint on a method in the client interface and execute that method in the tester, then in debug mode, you can step through the code before the tester returns a success/fail result. In non-debug mode, the tester will immediately return a result to indicate whether the method executed successfully. Additionally, in either debug or non-debug mode, the tester can display runtime artifacts from the system catalog created at runtime for the application module.

How to Verify Runtime Artifacts in the Oracle ADF Model Tester

When you want to run the Oracle ADF Model Tester, but do not require the use of the ADF Declarative Debugger you can display information about the runtime artifacts from the application module's system catalog. The system catalog displays business object metadata and other information that you may find useful when you need to compare business objects.

To launch the Oracle ADF Model Tester without debugging:

1. In the Applications window, right-click the desired application module and choose **Run**.
2. In the Oracle ADF Model Tester, choose **Create - Create SysCat AM**.
3. In the data model tree, expand the **SysCatAMDefs**, right-click **ViewDefs**, and choose **Show Table**.
4. In the data viewer, scroll vertically to locate the desired view instance in the **SCName** (system catalog name) field.

   Exposing the system catalog in the tester allows access to metadata and other information specific to the runtime objects without running the debugger. For example, you can check whether a view instance has a custom Java implementation class or not.

How to Refresh the Oracle ADF Model Tester with Application Changes

The Oracle ADF Model Tester is a highly interactive tool. When you run the tester and determine a change is needed in the data model project, you can return to JDeveloper to edit the desired **application module instance** and refresh the Oracle ADF Model Tester data model to display the changes. This way you can verify your changes without needing to rerun the tester.

To reload application metadata in the Oracle ADF Model Tester:

1. In the data model project, edit your business objects and save the changes in the JDeveloper.
2. Recompile the data model project.

   For example, you can right-click the data model project in the Applications window and choose **Make** to complete the recompile step.

   Although the metadata changes that you make are not involved in compiling the project, the compile step is necessary to copy the metadata to the class path and to allow the Oracle ADF Model Tester to reload it.
3. In the Oracle ADF Model Tester, in the toolbar, click **Reload application metadata**.

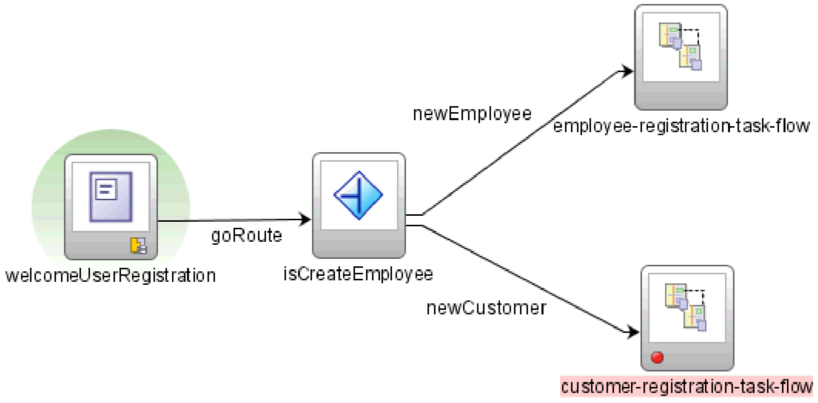
   Alternatively, you can choose **Reload Application** from the **File** menu of the Oracle ADF Model Tester.

Using the ADF Declarative Debugger

ADF Declarative Debugger allows you to place breakpoints on task flow activities, page definition bindings, page definition executables, and of course Java source code, too. When paused at a breakpoint during debugging runtime, several targeted new windows provide the ADF application developer a detailed and intuitive understanding of the application based on ADF runtime objects.

The ADF Declarative Debugger provides declarative breakpoints that you can set at the ADF object level (such as task flows, page definition executables, method and action bindings, ADF lifecycle phases), as well as standard breakpoints in custom Java code and Groovy script. ADF declarative breakpoints provide a high-level object view for debugging ADF applications. For example, you can break before a task flow activity to see what parameters would be passed to the task flow, as shown in Figure 48-11. To perform the same function using only Java breakpoints would require you to know which class or method to place the breakpoint in. ADF declarative breakpoints should be the first choice for ADF applications.

Figure 48-11 ADF Declarative Breakpoint on a Task Flow Activity



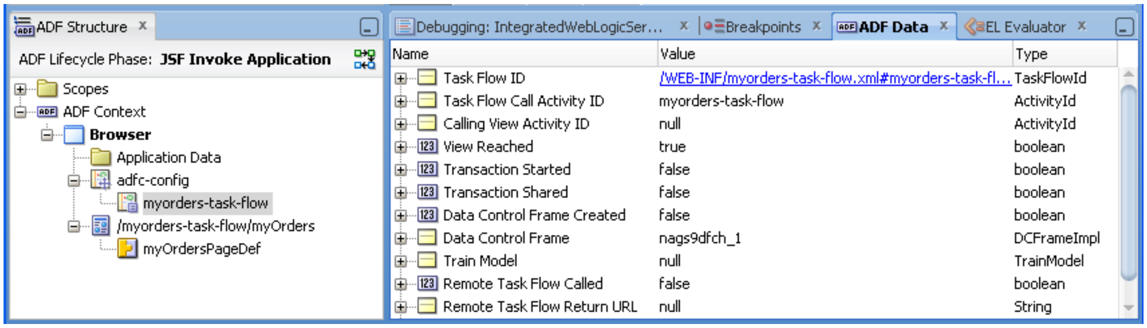
The ADF Declarative Debugger also supports standard Java code breakpoints. You can set Java code breakpoints in any ADF application. You may be able to use Java code breakpoints when an ADF declarative breakpoint does not break in the place you want.

The ADF Declarative Debugger is built on top of the Java debugger, so it has the features and behaviors of the Java debugger. But instead of needing to know the Java class or method, you can set ADF declarative breakpoints in visual editors.

The ADF Declarative Debugger provides standard debugging features such as the ability to examine variable and stack data. When an application pauses at any breakpoint (ADF Declarative or Java code breakpoint), you can examine the application status using a variety of windows. You can check where the break occurs in the Breakpoints window. You can check the call stack for the current thread using the Stack window. When you select a line in the Stack window, information in the Data window, Watches window, and all Inspector windows is updated to show relevant data. You can use the Data window to display information about arguments, local variables, and static fields in your application.

The ADF Structure window displays the runtime structure of the project. The ADF Data window automatically changes its display information based on the selection in the ADF Structure window. For example, if a task flow node is selected, the ADF Data window displays the relevant debugging information for task flows, as shown in Figure 48-12.

Figure 48-12 ADF Structure Window and ADF Data Window for a Task Flow Selection



ADF Lifecycle Phase: **JSF Invoke Application**

- Scopes
- ADF Context
  - **Browser**
    - Application Data
    - adfc-config
      - myorders-task-flow
    - /myorders-task-flow/myOrders
      - myOrdersPageDef

Name	Value	Type
Task Flow ID	/WEB-INF/myorders-task-flow.xml#myorders-task-fl...	TaskFlowId
Task Flow Call Activity ID	myorders-task-flow	ActivityId
Calling View Activity ID	null	ActivityId
View Reached	true	boolean
Transaction Started	false	boolean
Transaction Shared	false	boolean
Data Control Frame Created	false	boolean
Data Control Frame	nags9dfch_1	DCFrameImpl
Train Model	null	TrainModel
Remote Task Flow Called	false	boolean
Remote Task Flow Return URL	null	String

You can mix ADF declarative breakpoints with standard Java and Groovy breakpoints as needed in your debugging session. Although you can use step functions to advance the application from Java code breakpoint to Java code breakpoint, the step functions for ADF declarative breakpoints have more constraints and limitations. For information about using step functions on ADF declarative breakpoints, see Table 48-3.

For information on how to use ADF declarative breakpoints, see Setting ADF Declarative Breakpoints.

For information on how to use Java breakpoints on classes and methods, see Setting Breakpoints in Java Code and Groovy Script.

In a JSF application (including Fusion web applications), when a breakpoint breaks, you can use the EL Evaluator to examine the value of an EL expression. The EL Evaluator has the browse function that helps you select the correct expression to evaluate. See How to Use the EL Expression Evaluator.

Whether you plan to use ADF declarative breakpoints or standard Java and Groovy breakpoints, you can use the ADF Declarative Debugger with Oracle ADF source code. You can obtain Oracle ADF source code with Debug libraries. For information about loading source code. see Using ADF Source Code with the Debugger.

Using ADF Source Code with the Debugger

If you have valid Oracle ADF support, you can obtain complete source code for Oracle ADF by opening a service request with Oracle Worldwide Support. You can request a specific version of the Oracle ADF source code. You may be given download and password information to decrypt the source code ZIP file. Contact Oracle Worldwide Support for more information.

Adding Oracle ADF source code access to your application debugging session will:

- Provide access to the JDeveloper Quick Javadoc feature in the source editor. Without the source code, you will have only standard Javadoc.
- Enhance the use of Java code breakpoints by displaying the Oracle source code that's being executed when the breakpoint is encountered. You can also set breakpoints easier by clicking on the margin in the source code line you want to break on. Without the source code, you will have to know the class, method, or line number in order to set a breakpoint within Oracle code.
- For Java code breakpoints set within the source code, you will be able to see the values of all local variables and member fields in the debugger.

The ADF source code ZIP file may be delivered within an encrypted "outer" ZIP file to protect its contents during delivery. The "outer" ZIP name is sometimes a variant of the service request number.

After you have received or downloaded the "outer" ZIP archive file, unzip it with the provided password to access the actual source code ZIP file. The ADF source code ZIP name should be a variant of the Oracle ADF version number and build number. For example, the Oracle ADF source ZIP file may have a format similar to `adf_vvvv_nnnn_source.zip`, where *vvvv* is the version number and *nnnn* is the build number.

After you have access to the Oracle ADF source code ZIP file, extract its contents to a working directory.

How to Set Up the ADF Source User Library

You create a name for the source user library and then associate that name with the source ZIP file.

To add the ADF source zip file to the user library

1. In the main menu, choose **Tools** and then **Manage Libraries**.
2. In the Manage Libraries dialog, with the **Libraries** tab selected, click **New**.
3. In the Create Library window, enter a library name for the source that identifies the type of library.
4. Select the **Source Path** node in the tree structure. Click **Add Entry**.

> **Note:**
>
> Do not enter a value for the class path. You need to provide a value only for the source path.

5. In the Select Path Entry window, browse to the directory where the file was extracted and select the source ZIP file. Click **Select**.
6. In the Create Library window, verify that the source path entry has the correct path to the source ZIP file, and deselect **Deployed by Default**. Click **OK**.
7. Click **OK**.

How to Add the ADF Source Library to a Project

After the source library has been added to the list of available user libraries, add it to the project you want to debug.

To add the ADF source zip file to the project:

1. In the Applications window, double-click the project to which you want to add the ADF library or right-click the project and choose **Project Properties**.
2. In the Project Properties dialog, select **Libraries and Classpaths**.
3. Click **Add Library**.

4. In the Add Library dialog, under the **Users** node, select the source library you want to add and click **OK**.

   The source library should appear in the **Classpath Entries** section in the Project Properties dialog.

5. Click **OK**.

How to Use the EL Expression Evaluator

When the application is paused at a breakpoint, you can use the EL expression evaluator to enter an EL expression for evaluation. You can enter arbitrary EL expressions for evaluation within the current context. If the EL expression no longer applies within the current context, the value will be evaluated to `null`.

The EL Evaluator is different from the Watches window in that EL evaluation occurs only when stopped at a breakpoint, not when stopped at subsequent debugging steps.

The EL Evaluator is available for debugging any JSF application.

> ⚠ **Caution:**
>
> Be wary when you are evaluating EL expressions that you do not indirectly change application data and therefore the behavior of the application. For example, if you evaluate `#{foo.bar}`, the corresponding `getBar()` method modifies application data.

Before you begin:

It may be helpful to have an understanding of the ADF Declarative Debugger. For more information, see Using the ADF Declarative Debugger.

You will need to complete this task:

Set the desired breakpoints in the application, as described in Setting ADF Declarative Breakpoints. The application must be a JSF application. It does not need to be an ADF application.

To use the EL Evaluator:

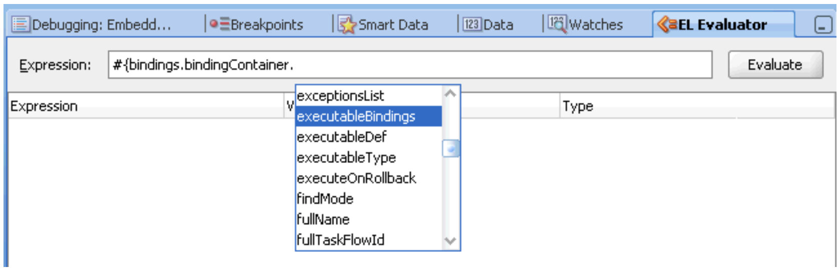
1. Start the debugging process by using any of the following techniques.
   - In the main menu, choose **Run** and then **Debug *<project_name>***.
   - From the Applications window, right-click the project, `adfc-config.xml`, task flow, or page and choose **Debug**.
   - From the task flow diagrammer, right-click an activity and choose **Debug**. Only task flows that do not use page fragments can be run.
2. When the breakpoint is reached, the EL Evaluator should appear as a tab in the debugger window area. Click the **EL Evaluator** tab to bring it forward. If it does not appear, in the main menu, choose **Window** and then **Debugger > EL Evaluator**.

> **Note:**
>
> Be sure that the application has actually hit a breakpoint by checking the Breakpoints window or checking that there is an **Execution Point** icon (red right arrow) next to the object breakpoint. Depending on where you set the breakpoint, an application may appear to be stopped when in fact it is waiting for user input at the page.

3. Enter an EL expression in the **Expression** input field.

   When you click in the field after entering #{ or after a period, a discovery function provides a selectable list of expression items, as shown in Figure 48-13. Auto-completion will be provided for easy entry. You can evaluate several EL expressions at the same time by separating them with semicolons.

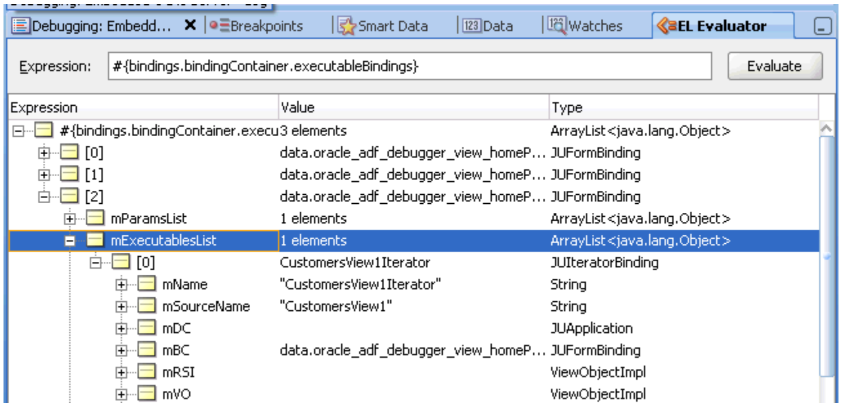
Figure 48-13 Using the Discovery Function of the EL Evaluator



Expression: #{bindings.bindingContainer.

4. When you finish entering the EL expression, click **Evaluate** and the expression is evaluated, as shown in Figure 48-14.

Figure 48-14 EL Expression Evaluated



Expression: #{bindings.bindingContainer.executableBindings}

Expression	Value	Type
#{bindings.bindingContainer.execu	3 elements	ArrayList<java.lang.Object>
[0]	data.oracle_adf_debugger_view_homeP...	JUFormBinding
[1]	data.oracle_adf_debugger_view_homeP...	JUFormBinding
[2]	data.oracle_adf_debugger_view_homeP...	JUFormBinding
mParamsList	1 elements	ArrayList<java.lang.Object>
mExecutablesList	1 elements	ArrayList<java.lang.Object>
[0]	CustomersView1Iterator	JUIteratorBinding
mName	"CustomersView1Iterator"	String
mSourceName	"CustomersView1"	String
mDC		JUApplication
mBC	data.oracle_adf_debugger_view_homeP...	JUFormBinding
mRSI		ViewObjectImpl
mVO		ViewObjectImpl

How to View and Export Stack Trace Information

If you are unable to determine what the problem is and you cannot resolve it yourself, typically your next step is to ask someone else for assistance. Whether you post a question in the OTN JDeveloper Discussion Forum or open a service request on Metalink, including the stack trace information in your posting is extremely useful to anyone who will need to assist you further to understand exactly where the problem is occurring.

JDeveloper's Stack window makes communicating this information easy. Whenever the debugger is paused, you can view the Stack window to see the program flow as a stack of method calls that got you to the current line. You can use the Preferences dialog to set the Stack window preference to include the line number information, as well as the class and method name that will be there by default. Finally, the Stack window context menu option **Export** lets you save the current stack information to an external text file whose contents you can then post or send to whomever might need to help you diagnose the problem.

Before you begin:

It may be helpful to have an understanding of the ADF Declarative Debugger. For more information, see Using the ADF Declarative Debugger.

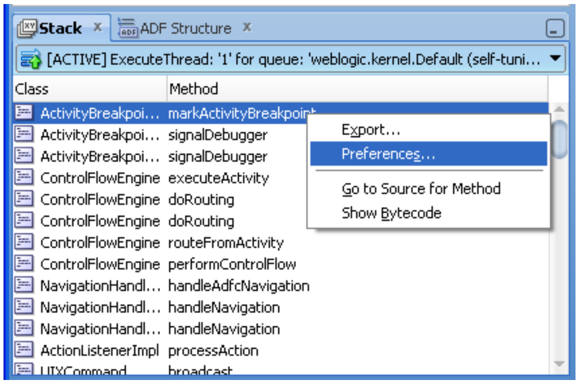
You will need to complete this task:

> Set the desired breakpoints in the application, as described in Setting ADF Declarative Breakpoints. The application must be a JSF application. It does not need to be an ADF application.

To save stack information to a text file:

1. Start the debugging process by using any of the following techniques.
   - In the main menu, choose **Run** and then **Debug *<project_name>***.
   - From the Applications window, right-click the project, `adfc-config.xml`, task flow, or page and choose **Debug**.
   - From the task flow diagrammer, right-click an activity and choose **Debug**. Only task flows that do not use page fragments can be run.
2. When the breakpoint is reached and the debugger is paused, the Stack window should appear as a window to the side of the debugger window area. Click the **Stack** tab to bring it forward. If the Stack window does not appear, in the main menu, choose **Window** and then **Debugger > Stack**.
3. When you want to change the Stack window preference for example to include the line number information in the export file, right-click the Stack window background and choose **Preferences**, as shown in Figure 48-15.

Figure 48-15 Stack Window Context Menu



[ACTIVE] ExecuteThread: '1' for queue: 'weblogic.kernel.Default (self-tuni...

Class	Method
ActivityBreakpoi...	markActivityBreakpoint
ActivityBreakpoi...	signalDebugger
ActivityBreakpoi...	signalDebugger
ControlFlowEngine	executeActivity
ControlFlowEngine	doRouting
ControlFlowEngine	doRouting
ControlFlowEngine	routeFromActivity
ControlFlowEngine	performControlFlow
NavigationHandl...	handleAdfcNavigation
NavigationHandl...	handleNavigation
NavigationHandl...	handleNavigation
ActionListenerImpl	processAction
UIXCommand	broadcast

4. Right-click the Stack window background and choose **Export**.
5. In the Export Stack dialog, name the file and choose the location where you want to save the file. Click **OK**.

Setting ADF Declarative Breakpoints

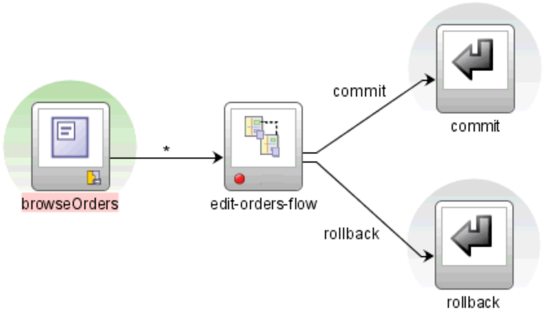
You can declaratively set breakpoints on task flow activities, page definition bindings, page definition executables, Java source code, and so forth using the ADF Declarative Debugger features.

You use the ADF Declarative Debugger features in JDeveloper to declaratively set breakpoints on **ADF task flow** activities, page definition executables, method, action, and value bindings, ADF Lifecycle phases, and contextual events. Instead of needing to know all the internal constructs of the ADF code, such as method names and class names, you can set breakpoints at the highest level of object abstraction.

You can add breakpoints to task flow activities in the task flow diagrammer or you can launch the Create ADF Task Flow Activity Breakpoint dialog from the Breakpoints window. In the task flow diagrammer, you can select a task flow activity and use the context menu to toggle or disable breakpoints on that activity, or press the F5 button. After the application pauses at the breakpoint, you can view the runtime structure of the objects in the ADF Structure window as a tree structure. The ADF Data window displays a list of data for a given object selected in the ADF Structure window.

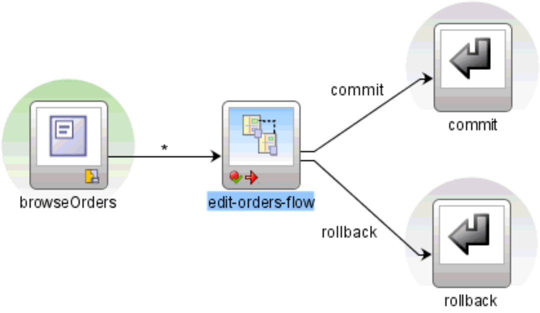
For example, when you set a breakpoint on a **task flow call activity** in the Browse Orders task flow, a red dot icon appears in the call activity, as shown in Figure 48-16.

Figure 48-16 ADF Declarative Breakpoint on a Task Flow Activity



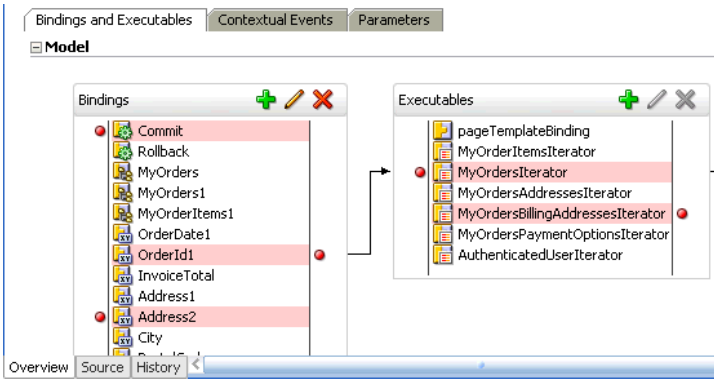
When the breakpoint is reached, the application is paused and the icon changes, as shown in Figure 48-17.

Figure 48-17 Application Paused at an ADF Declarative Breakpoint



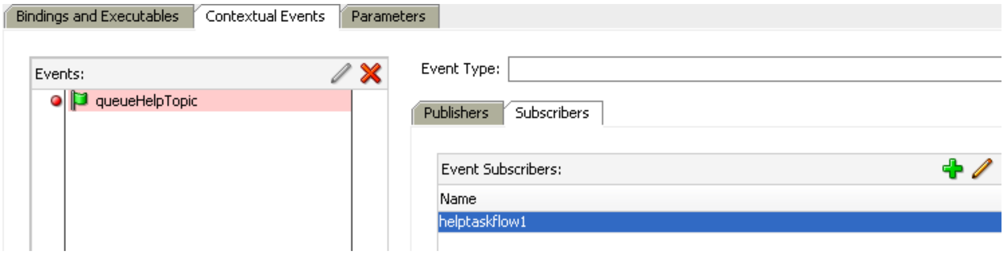
Similarly, you can set Before and After breakpoints in the page definition file. You set breakpoints for supported value bindings (see Table 48-1 for the list of supported value binding) and for executables by clicking on the left or right margin next to the item or by selecting from the context menu. Clicking on the left margin adds a **Before page definition** breakpoint, and clicking on the right margin adds an **After page definition** breakpoint. Again, a red dot icon that indicates the breakpoint is set, as shown in Figure 48-18.

Figure 48-18 ADF Declarative Breakpoints on ADF Bindings in the Page Definition File



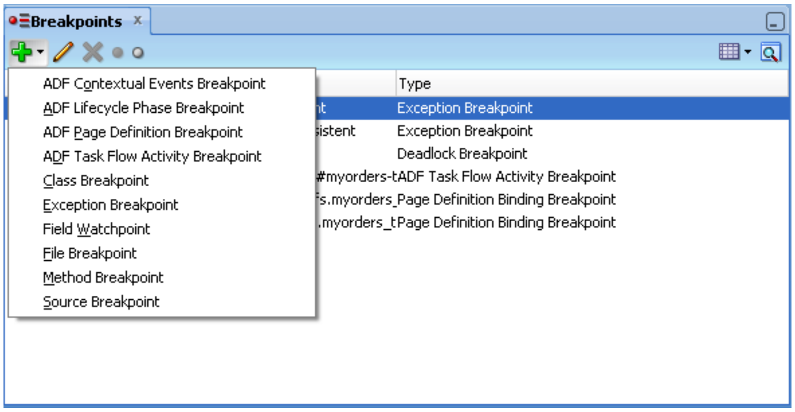
The page definition file also lets you set breakpoints on contextual events that your page or region within a page raises at runtime, as shown in Figure 48-19.

Figure 48-19 ADF Declarative Breakpoint on a Contextual Event in the Page Definition File



You can also set Before and After breakpoints on all the ADF lifecycle phases. You can launch the Create ADF Lifecycle Phase Breakpoint dialog from the Breakpoints window, as shown in Figure 48-20.

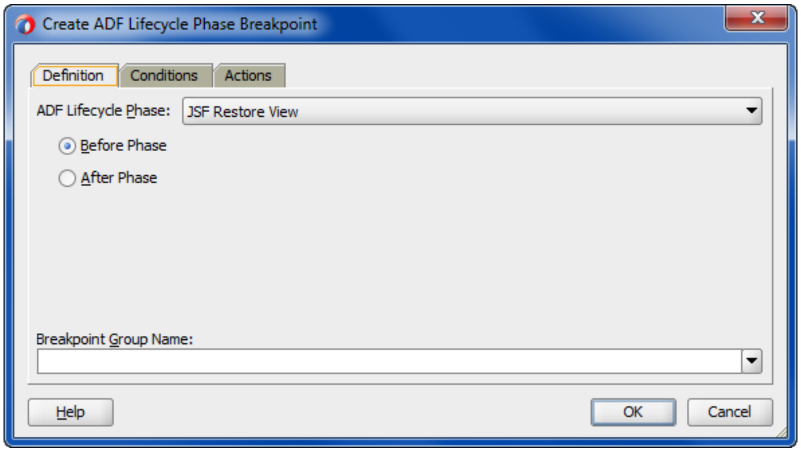
Figure 48-20 Breakpoints Window Add Breakpoint Icon Dropdown Menu


Breakpoints
ADF Contextual Events Breakpoint
ADF Lifecycle Phase Breakpoint
ADF Page Definition Breakpoint
ADF Task Flow Activity Breakpoint
Class Breakpoint
Exception Breakpoint
Field Watchpoint
File Breakpoint
Method Breakpoint
Source Breakpoint

	Type
nt	Exception Breakpoint
sistent	Exception Breakpoint
	Deadlock Breakpoint
#myorders-t	ADF Task Flow Activity Breakpoint
s.myorders_	Page Definition Binding Breakpoint
.myorders_t	Page Definition Binding Breakpoint

The Create ADF Lifecycle Phase Breakpoint dialog allows you to select different lifecycle breakpoint options, as shown in Figure 48-21.

Figure 48-21 Create ADF Lifecycle Phase Breakpoint Dialog



ADF Lifecycle Phase: JSF Restore View

● Before Phase

○ After Phase

Breakpoint Group Name:

Alternatively, you can use the ADF Lifecycle Breakpoints dialog from the ADF Structure window or the task flow diagrammer to set ADF lifecycle phase breakpoints. For more information about ADF lifecycle phases, see Understanding the Fusion Page Lifecycle .

You can define both ADF declarative breakpoints and standard Java code breakpoints when using the ADF Declarative Debugger. Depending on your debugging scenario, you may only need to use the declarative breakpoints to debug the application. Or you may find it necessary to add additional breakpoints in Java code that are not available declaratively. For information on Java code breakpoints, see How to Set Java Breakpoints on Classes and Methods. Table 48-1 lists the available ADF Declarative Debugger breakpoint locations.

Table 48-1 ADF Declarative Debugger Breakpoints

ADF Area	Breakpoint Type	Where Inserted	JDeveloper Context Menu Command	Description
ADF lifecycle phase	Before ADF lifecycle phase Break after ADF lifecycle phase	ADF Structure window **ADF Lifecycle Breakpoints** toolbar button Task flow diagrammer Breakpoints window **Add** button		A Before breakpoint pauses debugging before the ADF lifecycle phase. An After breakpoint pauses debugging after the ADF lifecycle phase. The ADF lifecycle JSF Render Response and Prepare Render phase Before and After breakpoints are executed in the following order: • Before `jsfRenderResponse`. • Before `prepareRender`. (`prepareRender` phase executes). • After `prepareRender`. (`jsfRenderResponse` phase executes). • After `jsfRenderResponse`.
ADF page definition - bindings and executables	Break before/Break after executable: • Iterator • Region instantiation	Page definition overview editor **Bindings and Executables** tab, Executables section Breakpoints window **Add** button	Toggle Breakpoint or F5 Disable Breakpoint	Pauses debugging before or after executable is refreshed. For task flow bindings, this represents two times per lifecycle: first, during `prepareModel` (initial region creation), and then again during `prepareRender` (where dynamic regions swap their corresponding task flow ID).
	Break before/Break after action binding: • `methodAction` • Built-in operations	Page definition overview editor **Bindings and Executables** tab, Bindings section Breakpoints window **Add** button	Toggle Breakpoint or F5 Disable Breakpoint	Pauses debugging before or after binding is executed.
	Break before/Break after attribute value binding	Page definition overview editor **Bindings and Executables** tab, Bindings section Breakpoints window **Add** button	Toggle Breakpoint or F5 Disable Breakpoint	Pauses debugging before or after the attribute's `setInputValue()` ADF source code method is executed. New values will be the parameters to `setInputValue()`.
	Break before/Break after **table binding**	Page definition overview editor **Bindings and Executables** tab, Bindings section Breakpoints window **Add** button	Toggle Breakpoint or F5 Disable Breakpoint	Pauses debugging before or after the ADF hierarchical binding's `updateValuesFromRows()` source code method is executed. New values will be the parameters to `updateValuesFromRows()`.

Table 48-1 (Cont.) ADF Declarative Debugger Breakpoints

ADF Area	Breakpoint Type	Where Inserted	JDeveloper Context Menu Command	Description
	Break before/Break after tree binding	Page definition overview editor **Bindings and Executables** tab, Bindings section Breakpoints window **Add** button	Toggle Breakpoint or F5 Disable Breakpoint	Pauses in either of these two cases: 1. When making a node selection in the tree, pauses debugging before or after the ADF hierarchical binding's `updateValuesFromRows()` source code method is executed. New values will be the parameters to `updateValuesFromRows()`. 2. When expanding the tree, pauses debugging before or after the ADF tree collection model's `modifyExpanded()` source code method is executed.
ADF page definition - contextual events	Break before/Break after contextual events	Page Definition overview editor **Contextual Events** tab, Events section Breakpoints window **Add** button	Toggle Breakpoint or F5 Disable Breakpoint	Pauses debugging either before the event is dispatched or after the event, just before the event is consumed by a subscriber to the event.
ADF task flow	Break before activity	Task flow diagrammer Breakpoints window **Add** button	Toggle Breakpoint or F5 Disable Breakpoint	Pauses debugging before the activity executes within the JSF Invoke Application phase. The activity where the declarative breakpoint is defined has not yet been executed. An exception are view activities; they pause within the JSF Render Response phase after the view activity is executed, but before the new page is rendered. By pausing at that point, the view activity values can be inspected using the ADF Structure and ADF Data windows.

The ADF Declarative Debugger uses the standard debugger icons and notations for setting, toggling, and indicating the status of ADF declarative breakpoints.

When an ADF declarative breakpoint is set, it appears as a red dot icon in the task flow activity, in the page definition breakpoint margins, or in the ADF Lifecycle Breakpoints window, as shown in Figure 48-22, Figure 48-23, Figure 48-24, and Figure 48-25.

Figure 48-22 ADF Declarative Breakpoint Enabled on a Task Flow Activity

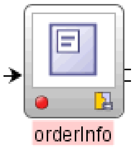
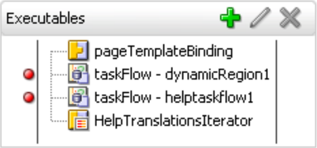


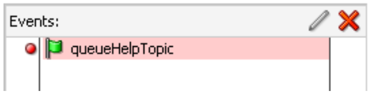
Figure 48-23 ADF Declarative Breakpoints Enabled in the Page Definition Executables



Executables

- pageTemplateBinding
- ● taskFlow - dynamicRegion1
- ● taskFlow - helptaskflow1
- HelpTranslationsIterator

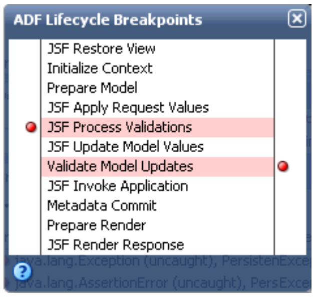
Figure 48-24 ADF Declarative Breakpoint Enabled on a Contextual Event in the Page Definition File



Events:

- ● queueHelpTopic

Figure 48-25 ADF Lifecycle Phase Breakpoints Enabled in the ADF Lifecycle Breakpoints Window



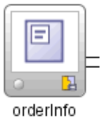
ADF Lifecycle Breakpoints

- JSF Restore View
- Initialize Context
- Prepare Model
- JSF Apply Request Values
- ● JSF Process Validations
- JSF Update Model Values
- Validate Model Updates ●
- JSF Invoke Application
- Metadata Commit
- Prepare Render
- JSF Render Response

java.lang.Exception (uncaught), PersistenExce
java.lang.AssertionError (uncaught), PersExce

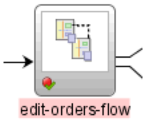
When an ADF task flow or page definition declarative breakpoint is disabled, the red icon becomes a gray icon, as shown in Figure 48-26.

Figure 48-26 ADF Declarative Breakpoint Disabled



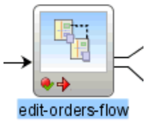
When an ADF task flow declarative breakpoint is active, the red dot icon has a green checkmark, as shown in Figure 48-27.

Figure 48-27 ADF Declarative Breakpoint Active



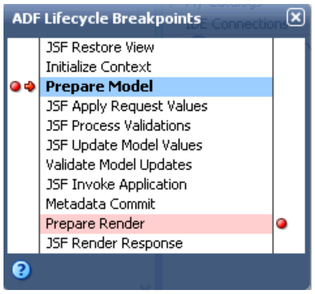
When the application is paused at an ADF declarative breakpoint, an **Execution Point** icon appears, as shown in Figure 48-28.

Figure 48-28 Application Paused at an Execution Point on a Task Flow



When the application is paused at an ADF lifecycle declarative breakpoint, an **Execution Point** icon appears next to the lifecycle phase in the ADF Lifecycle Breakpoints window, as shown in Figure 48-29. The name of the current ADF lifecycle phase is also displayed in the ADF Structure window.

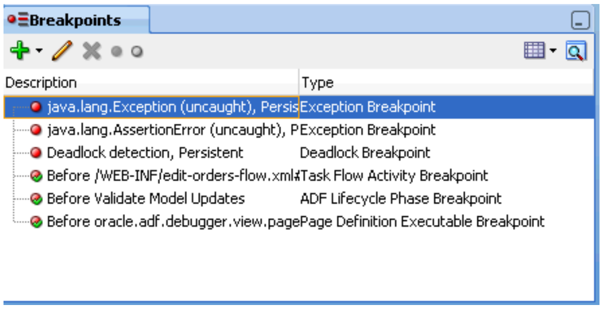
Figure 48-29 Application Paused at an Execution Point on an ADF Lifecycle Phase


ADF Lifecycle Breakpoints

- JSF Restore View
- Initialize Context
- **Prepare Model**
- JSF Apply Request Values
- JSF Process Validations
- JSF Update Model Values
- Validate Model Updates
- JSF Invoke Application
- Metadata Commit
- Prepare Render
- JSF Render Response

The Breakpoints window list all breakpoints, including ADF declarative breakpoints, as shown in Figure 48-30.

Figure 48-30 Breakpoints Window Showing ADF Declarative and Java Code Breakpoints


Breakpoints

Description	Type
java.lang.Exception (uncaught), Persis	Exception Breakpoint
java.lang.AssertionError (uncaught), P	Exception Breakpoint
Deadlock detection, Persistent	Deadlock Breakpoint
Before /WEB-INF/edit-orders-flow.xml#	Task Flow Activity Breakpoint
Before Validate Model Updates	ADF Lifecycle Phase Breakpoint
Before oracle.adf.debugger.view.page	Page Definition Executable Breakpoint

The Breakpoints window has a toolbar that includes buttons to add, edit, delete, enable, and disable breakpoints, as shown in Figure 48-30. The **Add Breakpoint** icon dropdown menu includes functions to create and manage ADF contextual events breakpoints, ADF lifecycle phase breakpoints, ADF page definition breakpoints (for ADF bindings and executables), ADF task flow activity breakpoints, and standard Java code breakpoints.

You can use the Breakpoints window to view the location of the ADF declarative breakpoint in its corresponding source file:

- Double-click a contextual event breakpoint to open the overview editor for the corresponding page definition file. You can then click the Contextual Events tab to view the location of the breakpoint.
- Double-click an ADF lifecycle phase breakpoint to open the ADF Lifecycle Breakpoints window which displays all ADF lifecycle execution points.
- Double-click a task flow activity breakpoint to open the task flow diagrammer for the corresponding task flow.
- Double-click an ADF binding breakpoint to open the overview editor for the corresponding page definition file.

To manage how the debugger handles a breakpoint, you can open the Edit ADF Breakpoint dialog for individual breakpoints that appear in the Breakpoints window. Or, you can select multiple ADF declarative breakpoints and customize the behavior of their common fields.

Table 48-2 lists how an ADF declarative breakpoint will appear in the Breakpoints window under the Description and Type columns.

Table 48-2 Breakpoints Window Display of ADF Declarative Breakpoints

Declarative Breakpoint Type	Description Column	Type Column
Before/After contextual event	Before *page definition@event name* After *page definition@event name*	Contextual events breakpoint
Before ADF lifecycle phase	Before *adf lifecycle phase*	ADF lifecycle phase breakpoint
After ADF lifecycle phase	After *adf lifecycle phase*	ADF lifecycle phase breakpoint
Before/After page definition executable: • Iterator	Before *page definition@executable id* After *page definition@executable id*	Page definition executable breakpoint
Before/After page definition action binding: • `methodAction` • Built-in Operations	Before *page definition@binding id* After *page definition@binding id*	Page definition binding breakpoint
Before/After page definition attribute value binding	Before *page definition@binding id* After *page definition@binding id*	Page definition binding breakpoint
Before/After page definition table binding	Before *page definition@binding id* After *page definition@binding id*	Page definition binding breakpoint

Table 48-2 (Cont.) Breakpoints Window Display of ADF Declarative Breakpoints

Declarative Breakpoint Type	Description Column	Type Column
Before/After page definition tree binding	Before *page definition@binding id* After *page definition@binding id*	Page definition binding breakpoint
Before ADF task flow activity	Before *task flow document#task flow id@activity id*	Task flow activity breakpoint

Table 48-3 lists the step commands that can be used with ADF declarative breakpoints.

Table 48-3 ADF Declarative Debugger Step Commands

ADF Debugger Step Commands	Description
Find Execution Point	Supported for declarative breakpoints to display the current execution point open and active within the corresponding editor.
Step Over (F8)	Supported for task flow activity declarative breakpoints to step from activity to activity within a task flow. If user interaction is required (for example, page displayed), once it is received (for example, button selected), processing will resume and then will pause before the next task flow activity. Supported for page definition executable breakpoints. The application will step to the next page definition executable breakpoint. Supported for ADF lifecycle phase declarative breakpoints to step to the next Before or After ADF lifecycle phase location.
Step Into (F7)	Supported only for task flow activity declarative breakpoints defined on task flow call activities. Task flow activity declarative breakpoints pause the application just before the activity is executed. The Step Into function provides the ability to pause debugging just prior to executing the called task flow default activity. This action would be the same as placing a task flow activity declarative breakpoint on the called task flow default activity.
Step Out (Shift F7)	Supported for task flow activity declarative breakpoints to step out of the current called task flow and back into the caller (if any). If user interaction is required (for example, page displayed) once user interaction received (for example, button selected), processing will resume and will pause before the next user interaction or activity within the calling task flow.
Continue Step (Shift F8)	Not supported for declarative breakpoints.
Step to End of Method	Not supported for declarative breakpoints.
Run to Cursor	Not supported for declarative breakpoints.
Pop Frame	Not supported for declarative breakpoints, as it is for Java code, to return to a previous point of execution.

How to Set and Use Task Flow Activity Breakpoints

After you have created a task flow diagram, you can set ADF declarative breakpoints on task flow activities.

Before you begin:

It may be helpful to have an understanding of when to use ADF declarative breakpoints. For more information, see Setting ADF Declarative Breakpoints.

To set a breakpoint on a task flow activity:

1. Open the task flow in the task flow diagrammer, or from the Breakpoints window, click the **Add** icon and select **ADF Task Flow Activity Breakpoint**.

   If the Breakpoints window is not already displayed, in the main menu, choose **Window** and then **Breakpoints**.

2. Set the task flow activity breakpoint.

   - If you use the task flow diagrammer, right-click and choose **Toggle Breakpoint** from the context menu, or press F5.

     A breakpoint icon appears on the task flow activity.

   - If you launched the Create ADF Task Flow Activity Breakpoint dialog from the Breakpoints window, click **Browse** to select a task flow definition, select the task flow from the **Task Flow** dropdown list, select the task flow activity from the **Activity** dropdown list, and click **OK**.

3. Optionally, configure a breakpoint's settings to manage the debugger:

   a. In the main menu, choose **Window** and then **Breakpoints**.

   b. In the Breakpoints window, select the task flow activity breakpoint you want to configure and click the **Edit** icon.

   c. In the Edit ADF Task Flow Activity Breakpoint dialog, click the **Conditions** tab, specify the conditions which apply to the breakpoint. The conditions must be valid for the breakpoint to occur.

   d. Click the **Actions** tab, specify the actions that you want the debugger to take when the breakpoint occurs and click **OK**.

      For example, the usual action for a breakpoint is to halt the program you are debugging, but you may want the debugger to beep and log information to the Log window without halting the program.

4. Start the debugging process by using any of the following techniques.

   - In the main menu, choose **Run** and then **Debug *<project_name>***.

   - From the Applications window, right-click the project, `adfc-config.xml`, task flow, or page and choose **Debug**.

   - From the task flow diagrammer, right-click an activity and choose **Debug**. Only task flows that do not use page fragments can be run.

5. When the application is paused at a breakpoint, an **Execution Point** icon (red right arrow) appears next to the breakpoint icon on the task flow activity. You can examine the application using various debugger windows.

> **Note:**
>
> Be sure that the application has actually hit a breakpoint by checking the Breakpoints window or checking that there is an **Execution Point** icon (red right arrow) next to the breakpoint. Depending on where you set the breakpoint, an application may appear to be stopped when in fact it is waiting for user input at the page.

Task flow activity declarative breakpoints pause the application just before the task flow activity is executed (except for view activities).

6. The ADF Structure window and the ADF Data window appear by default, as well as several debugger windows. You can examine the runtime structure in the ADF Structure window and its corresponding data in the ADF Data window. See How to Use the ADF Structure Window, and How to Use the ADF Data Window.
7. In the ADF Structure window, select a node and view pertinent information in the ADF Data window.

   Task flow activity declarative breakpoints pause the application just before the task flow activity is executed. You can use the **Step Into** (F7) function to pause the application just prior to executing the called task flow default activity.
8. Continue debugging the application as required, using the step functions as described in Table 48-3. The key step function is **Step Into** (F7).

   When the application is paused, you can remove or disable existing breakpoints and set new breakpoints.

How to Set and Use Page Definition Executable Breakpoints

If your page definition has executables, you can set breakpoints to pause the application before or after these executables. For example, you can set breakpoints to pause the application when iterators are refreshed.

> **Note:**
>
> If you are setting an After iterator breakpoint to pause the application after a view object query has been executed, be aware that the application may pause at this breakpoint multiple times. Also be aware that it may pause at this breakpoint even when the query has not been executed. If you need to know whether the query has been executed, select the relevant ADF Business Components in the ADF Structure window and view their corresponding data in the ADF Data window. For more information on how to use these windows, see How to Use the ADF Structure Window, and How to Use the ADF Data Window.
>
> For more information about using Java code breakpoints on view object query execution, see What You May Need to Know About Oracle ADF Breakpoints.

Before you begin:

It may be helpful to have an understanding of when to use ADF declarative breakpoints. For more information, see Setting ADF Declarative Breakpoints.

To set a breakpoint on an executable in the page definition file:

1. In the Applications window, double-click the page definition file that contains the executable in which you want to set a breakpoint.
2. In the overview editor, click the **Bindings and Executables** tab, select an executable from the **Executables** list, and click in the breakpoint margin to the left of the item.

   A breakpoint icon appears in the margin next to the item.
3. Optionally, configure a breakpoint's settings to manage the debugger:
   a. In the main menu, choose **Window** and then **Breakpoints**.
   b. In the Breakpoints window, select the executable breakpoint you want to configure and click the **Edit** icon.
   c. In the Edit ADF Page Definition Binding Breakpoint dialog, click the **Conditions** tab, specify the conditions which apply to the breakpoint. The conditions must be valid for the breakpoint to occur.
   d. Click the **Actions** tab, specify the actions that you want the debugger to take when the breakpoint occurs and click **OK**.

      For example, the usual action for a breakpoint is to halt the program you are debugging, but you may want the debugger to beep and log information to the Log window without halting the program.
4. Start the debugging process by using any of the following techniques.
   - In the main menu, choose **Run** and then **Debug <*project_name*>**.
   - From the Applications window, right-click the project, `adfc-config.xml`, task flow, or page and choose **Debug**.
   - From the task flow diagrammer, right-click an activity and choose **Debug**. Only task flows that do not use page fragments can be run.
5. When the application is paused at a breakpoint, an **Execution Point** icon (red right arrow) appears in the margin next to the breakpoint icon of the executable item. You can examine the application using several debugger windows.

   The application pauses when the executable binding is refreshed. If this is a taskFlow executable, the pause occurs in the `prepareModel` and the `prepareRender` lifecycles.

   > **Note:**
   >
   > Be sure that the application has actually hit a breakpoint by checking the Breakpoints window or checking that there is an **Executable Point** icon (red right arrow) next to the breakpoint. Depending on where you set the breakpoint, an application may appear to be stopped when in fact it is waiting for user input at the page.
6. The ADF Structure window and the ADF Data window appear by default, as well as several debugger windows. You can examine the runtime structure in the ADF

Structure window and its corresponding data in the ADF Data window. See How to Use the ADF Structure Window, and How to Use the ADF Data Window.

7. Select a node in the ADF Structure window and view pertinent information in the ADF Data window.

8. When the application is paused, you can remove or disable existing breakpoints and set new breakpoints.

How to Set and Use Page Definition Action Binding Breakpoints

You can set breakpoints in the page definition file on action bindings and `methodAction` bindings. The application pauses when the binding is executed.

Before you begin:

It may be helpful to have an understanding of when to use ADF declarative breakpoints. For more information, see Setting ADF Declarative Breakpoints.

To set a breakpoint on an action binding in the page definition file:

1. In the Applications window, double-click the page definition file that contains the binding in which you want to set a breakpoint.

2. In the overview editor, click the **Bindings and Executables** tab, select a `methodAction` binding or built-in operation item from the **Bindings** list, and click in the breakpoint margin to the left of the item.

   A breakpoint icon appears next to the item.

3. Optionally, configure a breakpoint's settings to manage the debugger:

   a. In the main menu, choose **Window** and then **Breakpoints**.

   b. In the Breakpoints window, select the action binding breakpoint you want to configure and click the **Edit** icon.

   c. In the Edit ADF Page Definition Binding Breakpoint dialog, click the **Conditions** tab, specify the conditions which apply to the breakpoint. The conditions must be valid for the breakpoint to occur.

   d. Click the **Actions** tab, specify the actions that you want the debugger to take when the breakpoint occurs and click **OK**.

      For example, the usual action for a breakpoint is to halt the program you are debugging, but you may want the debugger to beep and log information to the Log window without halting the program.

4. Start the debugging process by using any of the following techniques.

   - In the main menu, choose **Run** and then **Debug *<project_name>***.
   - From the Applications window, right-click the project, `adfc-config.xml`, task flow, or page and choose **Debug**.
   - From the task flow diagrammer, right-click an activity and choose **Debug**. Only task flows that do not use page fragments can be run.

5. When the application is paused at a breakpoint, an **Execution Point** icon (red right arrow) appears next to the breakpoint icon on the action binding item. You can examine the application using several debugger windows.

   The application is paused when the binding is executed.

> **Note:**
>
> Be sure that the application has actually hit a breakpoint by checking the Breakpoints window or checking that there is an **Execution Point** icon (red right arrow) next to the breakpoint. Depending on where you set the breakpoint, an application may appear to be stopped when in fact it is waiting for user input at the page.

6. The ADF Structure window and the ADF Data window appear by default, as well as several debugger windows. You can examine the runtime structure in the ADF Structure window and its corresponding data in the ADF Data window. See How to Use the ADF Structure Window, and How to Use the ADF Data Window.
7. Select a node in the ADF Structure window and view pertinent information in the ADF Data window.
8. When the application is paused, you can remove or disable existing breakpoints and set new breakpoints.

How to Set and Use Page Definition Value Binding Breakpoints

If the page definition has one of these values bindings, you can set breakpoints to pause the application:

- Attribute value binding
- Tree value binding
- Table value binding

Before you begin:

It may be helpful to have an understanding of when to use ADF declarative breakpoints. For more information, see Setting ADF Declarative Breakpoints.

To set a breakpoint on a value binding in the page definition file:

1. In the Applications window, double-click the page definition file that contains the binding in which you want to set a breakpoint.
2. In the overview editor, click the **Bindings and Executables** tab, select an attribute, tree, or table binding from the **Bindings** list, and click in the breakpoint margin to the left of the item. A breakpoint icon appears next to the value binding.
3. Optionally, configure a breakpoint's settings to manage the debugger:
   a. In the main menu, choose **Window** and then **Breakpoints**.
   b. In the Breakpoints window, select the binding breakpoint you want to configure and click the **Edit** icon.
   c. In the Edit ADF Page Definition Binding Breakpoint dialog, click the **Conditions** tab, specify the conditions which apply to the breakpoint. The conditions must be valid for the breakpoint to occur.
   d. Click the **Actions** tab, specify the actions that you want the debugger to take when the breakpoint occurs and click **OK**.

For example, the usual action for a breakpoint is to halt the program you are debugging, but you may want the debugger to beep and log information to the Log window without halting the program.

4. Start the debugging process by using any of the following techniques.
   - In the main menu, choose **Run** and then **Debug *<project_name>***.
   - From the Applications window, right-click the project, `adfc-config.xml`, task flow, or page and choose **Debug**.
   - From the task flow diagrammer, right-click an activity and choose **Debug**. Only task flows that do not use page fragments can be run.
5. When the application is paused at a breakpoint, an **Execution Point** icon (red right arrow) appears next to the breakpoint icon on the attribute value binding. You can examine the application using several debugger windows.

   The application pauses before an appropriate method of the ADF source code executes, as described in Table 48-1. New values will be the parameters that go into the method.

   > **Note:**
   >
   > Be sure that the application has actually hit a breakpoint by checking the Breakpoints window or checking that there is an **Execution Point** icon (red right arrow) next to the breakpoint. Depending on where you set the breakpoint, an application may appear to be stopped when in fact it is waiting for user input at the page.

6. The ADF Structure window and the ADF Data window appear by default, as well as several debugger windows. You can examine the runtime structure in the ADF Structure window and its corresponding data in the ADF Data window. See How to Use the ADF Structure Window, and How to Use the ADF Data Window.
7. Select a node in the ADF Structure window and view pertinent information in the ADF Data window.
8. Continue debugging the application as required, using the step functions as described in Table 48-3. The key step function is Step Over (F8).

   When the application is paused, you can remove or disable existing breakpoints and set new breakpoints.

How to Set and Use Page Definition Contextual Event Breakpoints

If the page definition defines contextual events, you can set breakpoints on the contextual events to pause the application.

Before you begin:

It may be helpful to have an understanding of when to use ADF declarative breakpoints. For more information, see Setting ADF Declarative Breakpoints.

To set a breakpoint on a contextual event in the page definition file:

1. In the Applications window, double-click the page definition file that contains the binding in which you want to set a breakpoint.

2. In the overview editor, click the **Contextual Events** tab, select a contextual event from the **Events** list, and click in the breakpoint margin to the left of the item. A breakpoint icon appears next to the contextual event.
3. Optionally, configure a breakpoint's settings to manage the debugger:
   a. In the main menu, choose **Window** and then **Breakpoints**.
   b. In the Breakpoints window, select the contextual event breakpoint you want to configure and click the **Edit** icon.
   c. In the Edit ADF Contextual Events Breakpoint dialog, click the **Conditions** tab, specify the conditions which apply to the breakpoint. The conditions must be valid for the breakpoint to occur.
   d. Click the **Actions** tab, specify the actions that you want the debugger to take when the breakpoint occurs and click **OK**.

      For example, the usual action for a breakpoint is to halt the program you are debugging, but you may want the debugger to beep and log information to the Log window without halting the program.
4. Start the debugging process by using any of the following techniques.
   - In the main menu, choose **Run** and then **Debug *<project_name>***.
   - From the Applications window, right-click the project, `adfc-config.xml`, task flow, or page and choose **Debug**.
   - From the task flow diagrammer, right-click an activity and choose **Debug**. Only task flows that do not use page fragments can be run.
5. When the application is paused at a breakpoint, an **Execution Point** icon (red right arrow) appears next to the breakpoint icon on the contextual event. You can examine the application using several debugger windows.

   The application pauses before the contextual event is raised.

   > **Note:**
   >
   > Be sure that the application has actually hit a breakpoint by checking the Breakpoints window or checking that there is an **Execution Point** icon (red right arrow) next to the breakpoint. Depending on where you set the breakpoint, an application may appear to be stopped when in fact it is waiting for user input at the page.

6. The ADF Structure window and the ADF Data window appear by default, as well as several debugger windows. You can examine the runtime structure in the ADF Structure window and its corresponding data in the ADF Data window. See How to Use the ADF Structure Window, and How to Use the ADF Data Window.
7. Select a node in the ADF Structure window and view pertinent information in the ADF Data window.
8. Continue debugging the application as required, using the step functions as described in Table 48-3. The key step function is Step Over (F8).

   When the application is paused, you can remove or disable existing breakpoints and set new breakpoints.

How to Set and Use ADF Lifecycle Phase Breakpoints

You can set both Before and After ADF lifecycle phase breakpoints on any of the ADF lifecycle phases. For each phase, you can set Before only, After only, or both. You can set breakpoints on as many phases as you want.

You can create the breakpoint and customize the options using the Create ADF Lifecycle Phase Breakpoint dialog from the Breakpoints window menu. Or You can create breakpoints with the default options using the ADF Lifecycle Breakpoints window. After a lifecycle breakpoint has been set, you can edit the options using the Edit ADF Lifecycle Phase Breakpoint dialog, which is also launched from the Breakpoints window.

You can set ADF lifecycle breakpoints on any of the ADF lifecycle phases:

- JSF Restore View
- Initialize Content
- Prepare Model
- JSF Apply Request Values
- JSF Process Validations
- JSF Update Model Values
- Validate Model Updates
- JSF Invoke Application
- Metadata Commit
- Prepare Render
- JSF Render Response

Before you begin:

It may be helpful to have an understanding of when to use ADF declarative breakpoints. For more information, see Setting ADF Declarative Breakpoints.

To set or manage an ADF lifecycle phase breakpoint from the Breakpoints window:

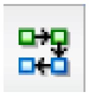
1. In the main menu, choose **Window** and then **Breakpoints**.
2. In the Breakpoints window, click **Add** and choose **ADF Lifecycle Phase Breakpoint**.
3. In the Create ADF Lifecycle Phase Breakpoint dialog **Definition** tab:
   - Select the ADF lifecycle phase where you want to set a breakpoint
   - Select **Before Phase** or **After Phase** breakpoint
4. In the **Conditions** tab, select the options you want and click **OK**.
5. In the **Actions** tab, select the options you want and click **OK**.

To set an ADF Lifecycle Phase Breakpoint using the breakpoint icon:

1. In the Applications window, double-click any task flow configuration file and in the overview editor click the **Diagram** tab.

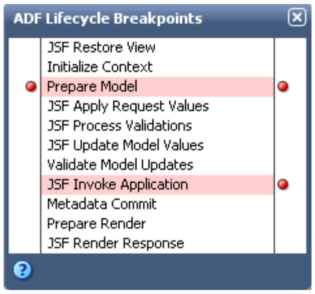
2. In the toolbar of the task flow diagrammer or in the title bar of ADF Structure window, click the **ADF Lifecycle Breakpoints** icon, as shown in Figure 48-31.

Figure 48-31 ADF Lifecycle Breakpoints Icon



3. In the ADF Lifecycle Breakpoints window, click in the left margin next to the ADF lifecycle phase to set a Before breakpoint, and on the right margin to set an After breakpoint. A red dot icon appears to indicate the breakpoint is set, as shown in Figure 48-32. The breakpoint will be set with the default breakpoint options. To remove the breakpoint, click the red dot icon.

Figure 48-32 Setting Breakpoints in the ADF Lifecycle Breakpoints Window



ADF Lifecycle Breakpoints

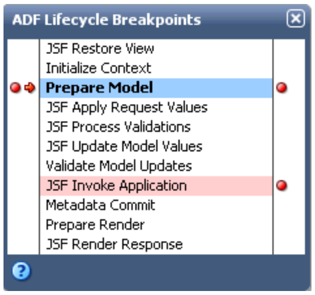
JSF Restore View
Initialize Context
● Prepare Model ●
JSF Apply Request Values
JSF Process Validations
JSF Update Model Values
Validate Model Updates
JSF Invoke Application ●
Metadata Commit
Prepare Render
JSF Render Response

4. If you want to edit breakpoint options, select the breakpoint in the Breakpoints window and click the **Edit** icon.

To debug an application using ADF Lifecycle Phase Breakpoints:

1. Start the debugging process by using any of the following techniques.
   - In the main menu, choose **Run** and then **Debug *<project_name>***.
   - From the Applications window, right-click the project, `adfc-config.xml`, task flow, or page and choose **Debug**.
   - From the task flow diagrammer, right-click an activity and choose **Debug**. Only task flows that do not use page fragments can be run.
2. When the application is paused at an ADF lifecycle phase breakpoint, an **Execution Point** icon (red right arrow) appears next to the breakpoint icon and the ADF lifecycle phase is in bold in the ADF Lifecycle Breakpoints window, as shown in Figure 48-33. You can examine the application using several debugger windows.

Figure 48-33 Execution Point Displayed in the ADF Lifecycle Breakpoints Window



> **Note:**
>
> Be sure that the application has actually hit a breakpoint by checking the Breakpoints window for an breakpoint encounter or checking that there is an **Execution Point** icon (red right arrow) next to the breakpoint. Depending on where you set the breakpoint, an application may appear to be stopped when in fact it is waiting for user input at the page.

3. The ADF Structure window and the ADF Data window appear by default, as well as several debugger windows. You can examine the runtime structure in the ADF Structure window and its corresponding data in the ADF Data window. The current ADF lifecycle phase is displayed at the top of the ADF Structure window. For more information, see How to Use the ADF Structure Window, and How to Use the ADF Data Window.
4. Select a node in the ADF Structure window and view pertinent information in the ADF Data window.
5. Continue debugging the application as required, using the step functions as described in Table 48-3. The key step function is Step Over (F8).

   When the application is paused, you can remove or disable existing breakpoints and set new breakpoints.

How to Use the ADF Structure Window

When the application is paused at a breakpoint, the ADF Structure window displays a tree structure of the ADF runtime objects and their relationships within the application. In particular, it shows the hierarchy of view ports, which represent either the main browser window or contained regions and portlets. When you select different items in the ADF Structure window, the data display in the accompanying ADF Data window changes. For more information about the ADF Data window, see How to Use the ADF Data Window.

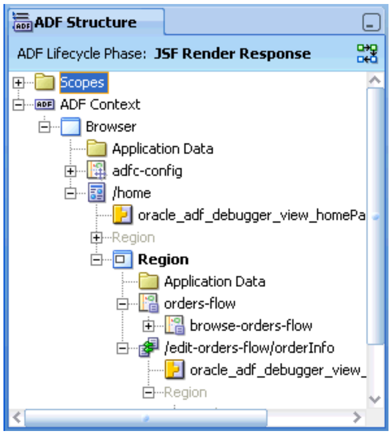
The ADF Structure window and the ADF Data window are shown by default during a debugging session when either of the following is true:

- The project being debugged contains a `WEB-INF/adfc-config.xml` file.
- The project being debugged contains any **ADF Faces** tag libraries.

You can launch the ADF Structure window by choosing **Window > Debugger > ADF Structure** in the main menu. From the ADF Structure window, you can launch the ADF Lifecycle Breakpoints window using the **ADF Lifecycle Breakpoints** icon.

When a breakpoint is encountered, the ADF Structure window displays the ADF lifecycle phase and a tree structure of the runtime objects, as shown in Figure 48-34.

Figure 48-34   ADF Structure Window Showing the Runtime Objects



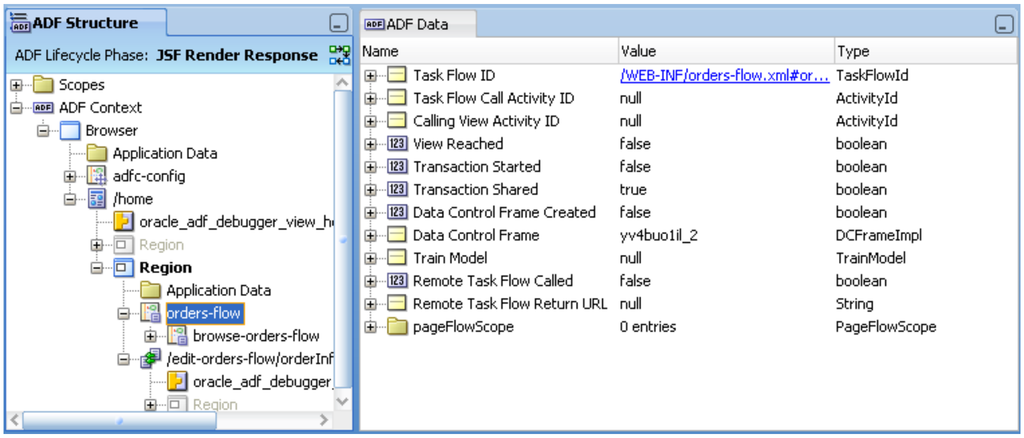
ADF Structure

ADF Lifecycle Phase: **JSF Render Response**

```
Scopes
ADF Context
    Browser
        Application Data
        adfc-config
        /home
            oracle_adf_debugger_view_homePa
            Region
            Region
                Application Data
                orders-flow
                    browse-orders-flow
                /edit-orders-flow/orderInfo
                    oracle_adf_debugger_view_
                Region
```

When you select an item in the ADF Structure window, the data and values associated with that item are displayed in the ADF Data window. Figure 48-35 shows a task flow selected in the ADF Structure window, with its corresponding information displayed in the ADF Data window.

Figure 48-35   ADF Structure Window Selection and ADF Data Window Data



ADF Structure

ADF Lifecycle Phase: **JSF Render Response**

```
Scopes
ADF Context
    Browser
        Application Data
        adfc-config
        /home
            oracle_adf_debugger_view_h
            Region
            Region
                Application Data
                orders-flow
                    browse-orders-flow
                /edit-orders-flow/orderInf
                    oracle_adf_debugger
                    Region
```

ADF Data

Name	Value	Type
Task Flow ID	/WEB-INF/orders-flow.xml#or...	TaskFlowId
Task Flow Call Activity ID	null	ActivityId
Calling View Activity ID	null	ActivityId
View Reached	false	boolean
Transaction Started	false	boolean
Transaction Shared	true	boolean
Data Control Frame Created	false	boolean
Data Control Frame	yv4buo1il_2	DCFrameImpl
Train Model	null	TrainModel
Remote Task Flow Called	false	boolean
Remote Task Flow Return URL	null	String
pageFlowScope	0 entries	PageFlowScope

The roots of the hierarchy are the sibling nodes **Scopes** and **ADF Context**. The current view port where processing has stopped appears in bold. Default selections within the tree will be retained from the previous breakpoint, so you can monitor any changes between breakpoints. The ADF object where the ADF declarative breakpoint was defined will be opened in the corresponding JDeveloper editor, either the task flow diagrammer or the overview editor for page definition files.

The ADF Structure tree will be rebuilt each time the application breaks and at subsequent steps to reflect the changed state of the objects. Although the entire tree hierarchy will be displayed, only items within the current view port and its parent view port(s) will be available for selection and further inspection. All other items in the tree hierarchy not in the current context will be dimmed and disabled. You can still use the hierarchy to identify runtime object relationships within the application, but it will be limited to the current context (and its parent view ports).

Table 48-4 lists the different types of items that can be displayed in the ADF Structure window hierarchy tree.

Table 48-4 ADF Structure Window Items

ADF Structure Tree Item	Description
Scopes	Displayed at the top of the ADF Structure hierarchy above its sibling **ADF Context** node. There is only one **Scopes** node in the ADF Structure hierarchy. You can expand the **Scopes** node to show a list of child scope nodes (such as **viewScope** and **pageFlowScope**). If you select a child scope node, the ADF Data window displays the variables and values for that scope.
ADF context	Displayed as the root node of the ADF Structure hierarchy below its sibling **Scopes** node. There will only be one **ADF Context** within the ADF Structure hierarchy.
View port	View ports are an ADF Controller concept. For this reason, view ports appear within the ADF Structure hierarchy only when the application being debugged utilizes ADF Controller. View ports can represent one of the following: • Browser: Main browser view ports, also known as root view ports, appear as children of the root **ADF Context**. If multiple browser windows are open during the debugging runtime session, multiple browser view ports are presented within the hierarchy. The label of each browser view port displays the text "Browser". The view port also provides a tooltip for the view port ID similar to the following example: "Root View Port: 999999". • Region: Region view ports appear as the children of page or page fragments. They are also known as child view ports. The label of each region view port displays the text "Region". The region also provides a tooltip for the view port ID similar to the following example: "Child View Port: 999999".
ADF task flows	The page flow stack corresponding to each view port appears as a hierarchy of ADF task flows. The initial ADF task flow called for the stack is a direct child of its corresponding view port. The label of each ADF task flow reflects the corresponding ADF task flow display name (if any) or its task flow ID. Region view ports will not display the item in their page flow stack hierarchy for their implied unbounded task flow. The task flow also provides a tooltip displaying the ADF task flow path, and a context menu item to open to the corresponding ADF task flow within the editor workspace. If ADF Controller is not utilized in the application (or if the page is run outside the context of an ADF task flow), ADF task flows will not appear within the hierarchy.
Page	Represents the page (view) currently displayed within a browser view port. Presented along with its associated binding container (if any) as a child. If the application being debugged utilizes ADF Controller, pages will be children of each browser view port. The label of each page reflects its corresponding runtime view ID. The page also provides a tooltip displaying the page path, and a context menu item to open to the corresponding page within the editor workspace. If a visual user interface in not implemented for the application, the page will not appear within the hierarchy.
Page fragment	Represents the page fragment currently displayed within a region view port. Presented along with its associated binding container (if any) as a child. If the application being debugged utilizes ADF Controller, page fragments will be children of each region view port. The label of each page fragment node reflects its corresponding runtime view ID. The page fragment also provides a tooltip displaying the source file page definition path, and a context menu item to open to the corresponding page fragment within the editor workspace.

Table 48-4 (Cont.) ADF Structure Window Items

ADF Structure Tree Item	Description
Binding container	Represents the binding container for the corresponding page or page fragment. The label of each binding container reflects its corresponding file name (page definition file) without the extension. The binding container node will also provide a tooltip displaying the page fragment path. The binding container also appears under current task flows when used to represent task flow activity bindings (for example, method call activity bindings). If ADF Model is not utilized for the application, binding containers will not appear.
Application data	Represents the application data objects (for example, ADF Business Components objects or ADF Business Components business service objects) instantiated within the data control frame for the corresponding view port (or binding container if ADF Controller is not used). Application data objects don't need to be currently instantiated for the Application Data node to appear.

How to Use the ADF Data Window

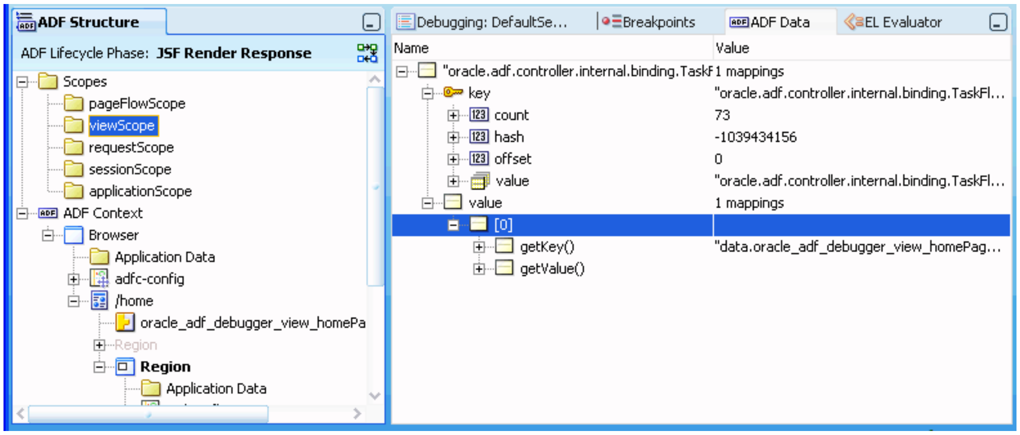
When an application is paused at an ADF declarative breakpoint, the ADF Data window displays relevant data based on the selection in the ADF Structure window. You can launch the ADF Data window by choosing **Window > Debugger > ADF Data** in the main menu. The content of the ADF Data window based on the selection in the ADF Structure window is summarized in Table 48-5.

Table 48-5 ADF Data Window Content for an ADF Structure Window Selection

ADF Structure Window	ADF Data Content
Scopes	Displays memory scope values based on the current context. `pageFlowScope` will also appear within ADF task flow content for the `pageFlowScope` values specific to a selected ADF task flow (not necessarily the current context). `viewScope` will also appear within the view port content for the `viewScope` values specific to a selected view port (not necessarily the current context).
ADF context	Displays the ADF context variables and values hierarchy. ADF context variables and values can be inspected by evaluating the `#(data.adfContext)` EL expression in the EL Evaluator.
View port	Displays view port details, including the `viewScope` contents.
Page flow stack entry	Displays information for the selected page flow stack entry, including current transaction status and ADF Model save point status.
Page/page fragment	Displays the page or page fragment UI component tree hierarchy for the selected page or page fragment if the page or page fragment has been rendered.
Binding container	Displays the binding container runtime values, including parameters, bindings, and executables.
Application data	Displays application data objects (for example, ADF Business Components objects) instantiated within the current **binding context**. If the business service layer is implemented with a technology other than ADF Business Components objects (for example, EJB) the application data objects will be displayed in a more generic form.

The **Scopes** node in the ADF Structure window can be expanded to show a list of child scope nodes. When a child scope node is selected in the ADF Structure window, the ADF Data window displays the current context values for the selected memory scope, as shown in Figure 48-36.

Figure 48-36 Child Scope Selected in the ADF Structure Window



ADF Structure

ADF Lifecycle Phase: **JSF Render Response**

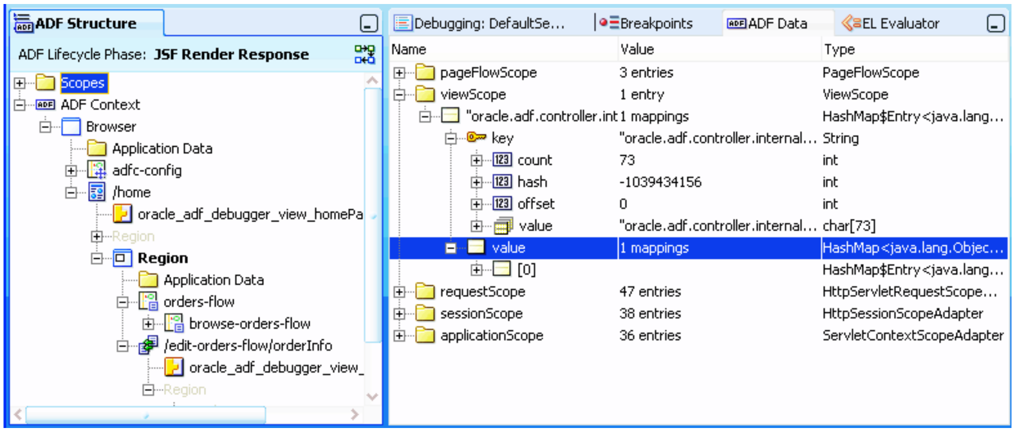
```
Scopes
    pageFlowScope
    viewScope
    requestScope
    sessionScope
    applicationScope
ADF Context
    Browser
        Application Data
        adfc-config
        /home
            oracle_adf_debugger_view_homePa
            Region
            Region
                Application Data
```

Debugging: DefaultSe... | Breakpoints | ADF Data | EL Evaluator

Name	Value
"oracle.adf.controller.internal.binding.TaskF	1 mappings
key	"oracle.adf.controller.internal.binding.TaskFl...
count	73
hash	-1039434156
offset	0
value	"oracle.adf.controller.internal.binding.TaskFl...
value	1 mappings
[0]	
getKey()	"data.oracle_adf_debugger_view_homePag...
getValue()	

If the **Scopes** node itself is selected, then the full list of memory scopes appears also in the ADF Data windows, which can also be expanded for inspection. Figure 48-37 shows the **Scopes** node selected in the ADF Structure window, and the **viewScope** child node being selected with its values displayed in the ADF Data window. You can inspect the values of `requestScope`, `viewScope`, `pageFlowScope`, `applicationScope`, and `sessionScope` by expanding each corresponding node. `pageFlowScope` will also appear within the ADF Task Flow content to reflect the values of the specific ADF task flow currently selected in the ADF Structure window. `viewScope` will also appear within the view port content to reflect the values of the specific view port currently selected in the ADF Structure window.

Figure 48-37 Scopes Node Selected in the ADF Structure Window



ADF Structure

ADF Lifecycle Phase: **JSF Render Response**

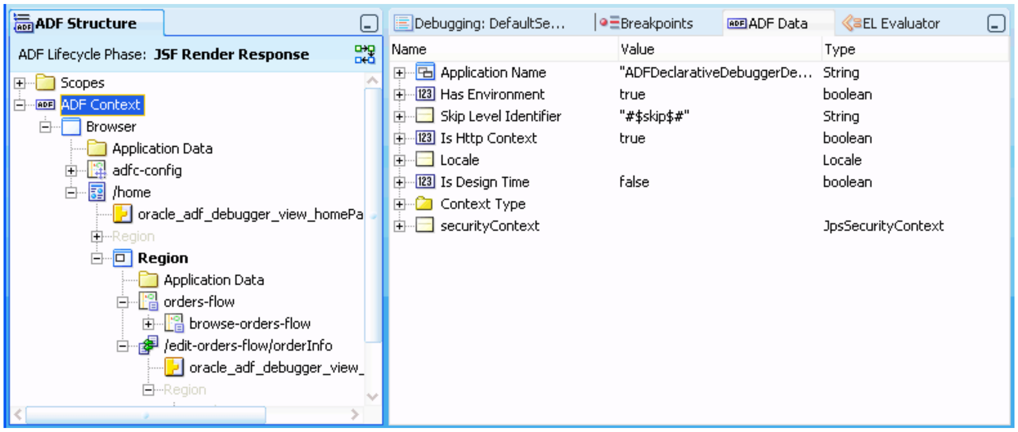
```
Scopes
ADF Context
    Browser
        Application Data
        adfc-config
        /home
            oracle_adf_debugger_view_homePa
            Region
            Region
                Application Data
                orders-flow
                    browse-orders-flow
                /edit-orders-flow/orderInfo
                    oracle_adf_debugger_view_
                    Region
```

Debugging: DefaultSe... | Breakpoints | ADF Data | EL Evaluator

Name	Value	Type
pageFlowScope	3 entries	PageFlowScope
viewScope	1 entry	ViewScope
"oracle.adf.controller.int	1 mappings	HashMap$Entry<java.lang...
key	"oracle.adf.controller.internal...	String
count	73	int
hash	-1039434156	int
offset	0	int
value	"oracle.adf.controller.internal...	char[73]
value	1 mappings	HashMap<java.lang.Objec...
[0]		HashMap$Entry<java.lang...
requestScope	47 entries	HttpServletRequestScope...
sessionScope	38 entries	HttpSessionScopeAdapter
applicationScope	36 entries	ServletContextScopeAdapter

When the ADF context is selected in the ADF Structure window, as shown in Figure 48-38, the current value of the ADF context variables will be displayed in the ADF Data window. You can also inspect ADF context variables and values by evaluating the `#(data.adfContext)` EL expression in the EL Evaluator. For more information, see How to Use the EL Expression Evaluator.

Figure 48-38 ADF Context Selected for the ADF Data Window



ADF Structure

ADF Lifecycle Phase: **JSF Render Response**

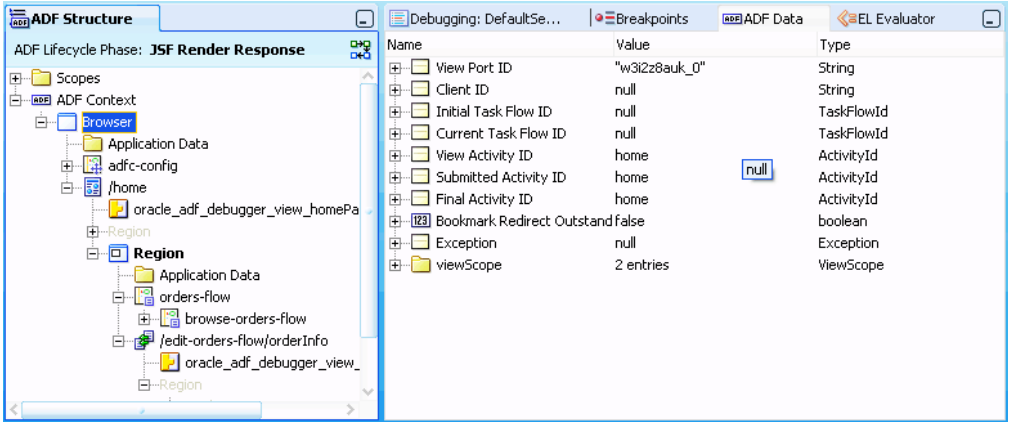
```
Scopes
ADF Context
  Browser
    Application Data
    adfc-config
    /home
      oracle_adf_debugger_view_homePa
      Region
      Region
        Application Data
        orders-flow
          browse-orders-flow
        /edit-orders-flow/orderInfo
          oracle_adf_debugger_view_
          Region
```

Debugging: DefaultSe... | Breakpoints | ADF Data | EL Evaluator

Name	Value	Type
Application Name	"ADFDeclarativeDebuggerDe...	String
Has Environment	true	boolean
Skip Level Identifier	"#$skip$#"	String
Is Http Context	true	boolean
Locale		Locale
Is Design Time	false	boolean
Context Type		
securityContext		JpsSecurityContext

Selecting a view port within the ADF Structure hierarchy will display the view port's current view port details in the ADF Data window, as shown in Figure 48-39. Values displayed for each view port are summarized in Table 48-6.

Figure 48-39 View Port Selected for the ADF Data Window



ADF Structure

ADF Lifecycle Phase: **JSF Render Response**

```
Scopes
ADF Context
  Browser
    Application Data
    adfc-config
    /home
      oracle_adf_debugger_view_homePa
      Region
      Region
        Application Data
        orders-flow
          browse-orders-flow
        /edit-orders-flow/orderInfo
          oracle_adf_debugger_view_
          Region
```

Debugging: DefaultSe... | Breakpoints | ADF Data | EL Evaluator

Name	Value	Type
View Port ID	"w3i2z8auk_0"	String
Client ID	null	String
Initial Task Flow ID	null	TaskFlowId
Current Task Flow ID	null	TaskFlowId
View Activity ID	home	ActivityId
Submitted Activity ID	home	ActivityId
Final Activity ID	home	ActivityId
Bookmark Redirect Outstand	false	boolean
Exception	null	Exception
viewScope	2 entries	ViewScope

null

Table 48-6 ADF Data Window Content for View Port

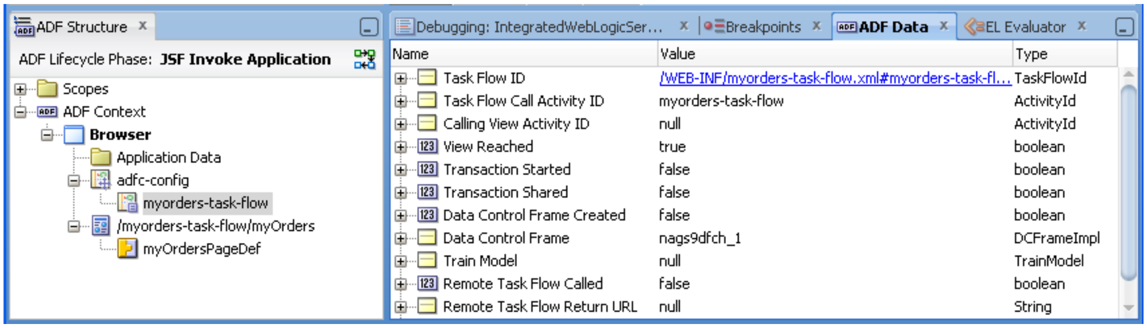
View Port	Description
View port ID	It is displayed
Client ID	It is displayed
Initial task flow ID	Initial ADF task flow on the view ports page flow stack. Not displayed for unbounded task flows. Appears as a link to open the corresponding task flow definition in the editor workspace.

Table 48-6 (Cont.) ADF Data Window Content for View Port

View Port	Description
Current task flow ID	Displayed for bounded task flows and not displayed for unbounded task flows. Current ADF task flow on the view port's page flow stack. Appears as a link to open the corresponding task flow definition in the editor workspace.
View activity ID	Current ADF task flow view activity ID. Applicable only if the current ADF task flow activity is a view activity.
Submitted activity ID	ADF task flow activity submitting the current request.
Final activity ID	ADF task flow activity receiving the current request.
Bookmark redirect outstanding	(Boolean)
Exception	(If any)
View memory scope	View memory scope variables and values for the selected view port.

In the ADF Structure window, each individual ADF task flow within a page flow stack hierarchy is selectable. An ADF task flow selected in the ADF Structure window will display the current task flow information in the ADF Data window, as shown in Figure 48-40. Task flow templates utilized by the selected ADF task flow will be determined by manually navigating to the ADF task flow source file. This is the same way similar functionalities are handled for Java source files. Current information for a selected ADF task flow is summarized in Table 48-7.

Figure 48-40 Task Flow Selected for the ADF Data Window



ADF Lifecycle Phase: **JSF Invoke Application**

- Scopes
- ADF Context
  - **Browser**
    - Application Data
    - adfc-config
      - myorders-task-flow
    - /myorders-task-flow/myOrders
      - myOrdersPageDef

Name	Value	Type
Task Flow ID	/WEB-INF/myorders-task-flow.xml#myorders-task-fl…	TaskFlowId
Task Flow Call Activity ID	myorders-task-flow	ActivityId
Calling View Activity ID	null	ActivityId
View Reached	true	boolean
Transaction Started	false	boolean
Transaction Shared	false	boolean
Data Control Frame Created	false	boolean
Data Control Frame	nags9dfch_1	DCFrameImpl
Train Model	null	TrainModel
Remote Task Flow Called	false	boolean
Remote Task Flow Return URL	null	String

Table 48-7 ADF Data Window Content for Task Flow

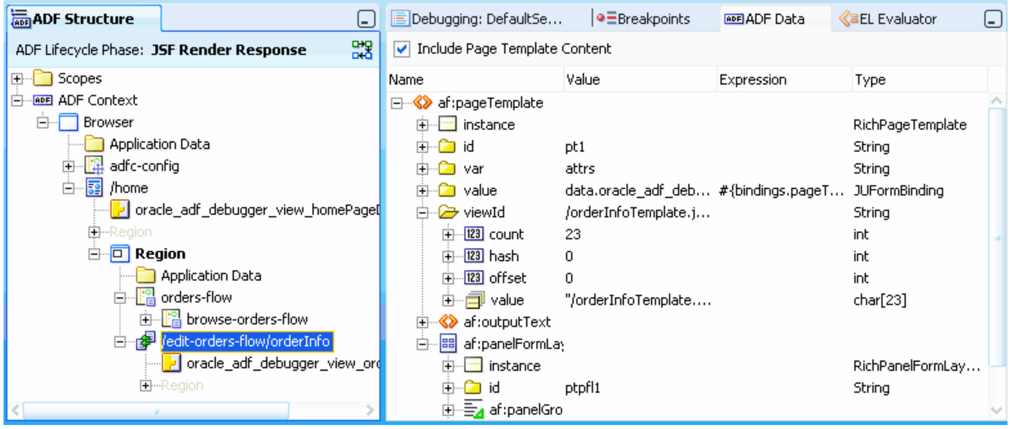
Task Flow	Description
ADF task flow reference	ADF task flow reference
Task flow call activity ID	Task flow activity ID for the calling task flow. Will be `null` for the first ADF task flow within each view port task flow Calls window.
Calling view activity ID	The calling view activity of the current view activity displayed by the ADF task flow, if any.
View reached	(Boolean)

Table 48-7 (Cont.) ADF Data Window Content for Task Flow

Task Flow	Description
Train model	Only applicable if the ADF task flow is created as a train.
Transaction started	(Boolean) Identifies the current status of the ADF task flow transactional state. For example, did the ADF task flow begin a new transaction?
Transaction shared	(Boolean) Identifies the current status of the ADF task flow transactional state. For example, did the ADF task flow join an existing transaction?
Save point	Identifies the current status of the ADF task flow's ADF Model save point creation state. For example, was a model save point created upon ADF task flow entry?.
Remote task flow called	(Boolean)
Remote task flow return URL	Applies only when calling an ADF task flow remotely. Identifies the URL for return once the task flow called remotely completes.
Data control frame created	(Boolean)
Data control frame	Name of data control frame associated with the ADF task flow.
Page flow memory scopes	Appears as an expandable node to allow inspection of the values of the page flow memory scopes for the task flow selected in the ADF Structure window. The page flow memory scopes will also be displayed within the ADF Structure window's **Scopes** node. However, the page flow memory scope for the **Scopes** node will always be based on the application's current context, not the selected task flow.

When you select a page or page fragment node in the ADF Structure hierarchy, the corresponding UI component tree is displayed within the ADF Data window, as shown in Figure 48-41. If a page or page fragment is based on a page template, you can include the content coming from the page template outside any facet reference elements by selecting the **Include Page Template Content** checkbox at the top of the ADF Data window. If the page template content is not included, the page or page fragment UI component tree will appear structurally similar to its source file.

Figure 48-41 Page Selected for the ADF Data Window



☑ Include Page Template Content

Name	Value	Expression	Type
af:pageTemplate			
instance			RichPageTemplate
id	pt1		String
var	attrs		String
value	data.oracle_adf_deb...	#{bindings.pageT...	JUFormBinding
viewId	/orderInfoTemplate.j...		String
count	23		int
hash	0		int
offset	0		int
value	"/orderInfoTemplate....		char[23]
af:outputText			
af:panelFormLa			
instance			RichPanelFormLay...
id	ptpfl1		String
af:panelGro			

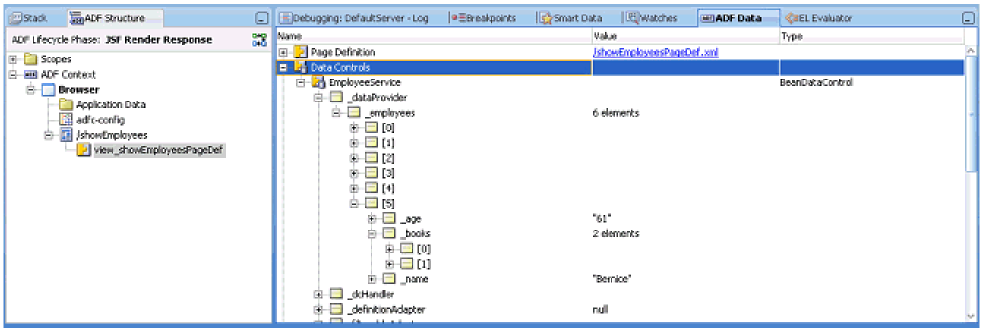
When you select a binding container in the ADF Structure hierarchy, it displays within the ADF Data window the node selection listed in Table 48-8.

Table 48-8 ADF Data Window Content for Binding Container

Binding Container	Description
Page definition link	Navigates to the corresponding page definition source file and opens it within the editor workspace.
Data Controls	Displays the binding container's data controls. Data controls implemented by ADF Business Components objects and non-ADF Business Components objects will be presented slightly different. ADF Business Components-based data controls will appear similar to the actual business service implementation using row collections. Non-ADF Business Components-based data controls will typically appear as raw member variables similar to what is displayed in the ADF Declarative Debugger Data window. The ADF Data window shows only cached information such as member variables and arrays. Standard debugger functionality can also be used to customize each element. Each ADF Business Components data control will display the following information: • Row collections • Query string for each row collection • Query string with variable substitution for each row collection • Application data rows • Current row indicator • Change indicator • Current and original values (if changed within the same request)
Parameters	Current values of all binding container parameters.
Executables	Displays executables showing current row indicators, and current and original values (if changed within the same request). This includes the following types of executables: • Iterator - presents corresponding attribute bindings along with their `Refresh` and `RefreshCondition` properties. • task flow - current value of the task flow ID assigned to the task flow binding and all of its associated parameter values. Task flow IDs will appear as links navigating to open the corresponding task flow definition source file within the editor workspace. Link text consists of `<task flow source document>#<task flow id>`. • Search region - presented similar to iterators, but also displays criteria and criteria with substitution information.
Bindings	Displays value, table, tree, and method bindings. Each binding will display the following information: • Associated executables • Change indicator • Current and original values (if changed within the same request)
Binding container of page template	If the corresponding page or page fragment utilized a page template, the binding container of the page template will appear as a child of page or page fragment binding container content.

When you select a binding container for an application based on non-ADF Business Components objects, the ADF Data window displays the binding container content, as shown in Figure 48-1.

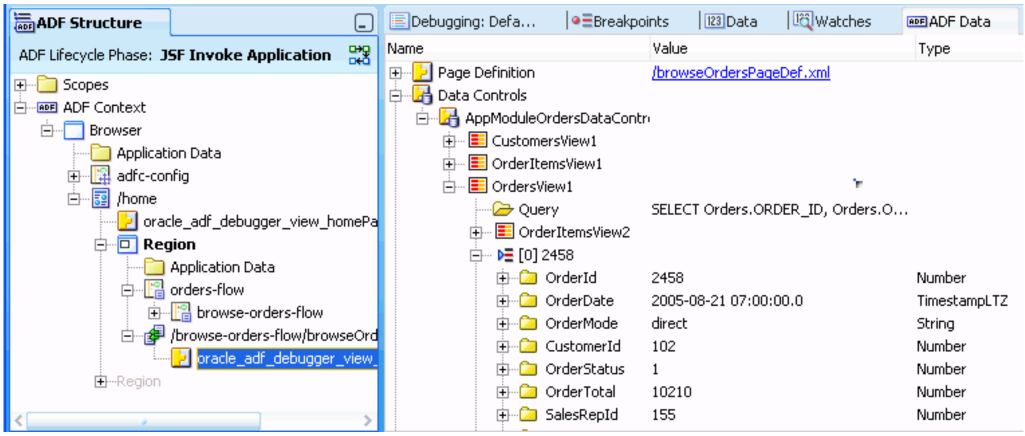
Figure 48-42 Binding Container (Non-Business Components) Selected for the ADF Data Window



Name	Value	Type
Page Definition	/showEmployeesPageDef.xml	
Data Controls		
EmployeeService		BeanDataControl
_dataProvider		
_employees	6 elements	
[0]		
[1]		
[2]		
[3]		
[4]		
[5]		
_age	"61"	
_books	2 elements	
[0]		
[1]		
_name	"Bernice"	
_dcHandler		
_definitionAdapter	null	

When you select a binding container for an application based on ADF Business Components objects, the ADF Data window displays standard row collection icons, as shown in Figure 48-2.

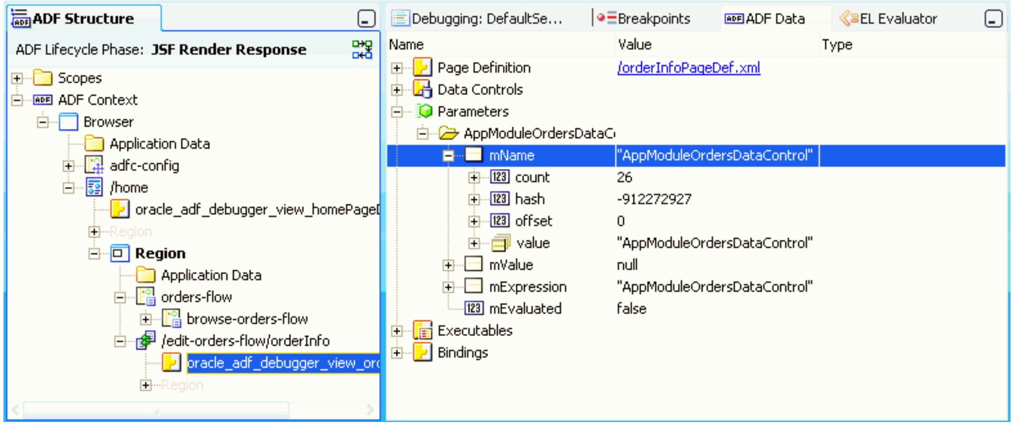
Figure 48-43 Binding Container (Business Components) Selected for the ADF Data Window



Name	Value	Type
Page Definition	/browseOrdersPageDef.xml	
Data Controls		
AppModuleOrdersDataContr		
CustomersView1		
OrderItemsView1		
OrdersView1		
Query	SELECT Orders.ORDER_ID, Orders.O...	
OrderItemsView2		
[0] 2458		
OrderId	2458	Number
OrderDate	2005-08-21 07:00:00.0	TimestampLTZ
OrderMode	direct	String
CustomerId	102	Number
OrderStatus	1	Number
OrderTotal	10210	Number
SalesRepId	155	Number

Expanding the Parameters node in the ADF Data window displays information similar to that shown in Figure 48-44.

Figure 48-44   Parameters Selected for the ADF Data Window



ADF Structure

ADF Lifecycle Phase: **JSF Render Response**

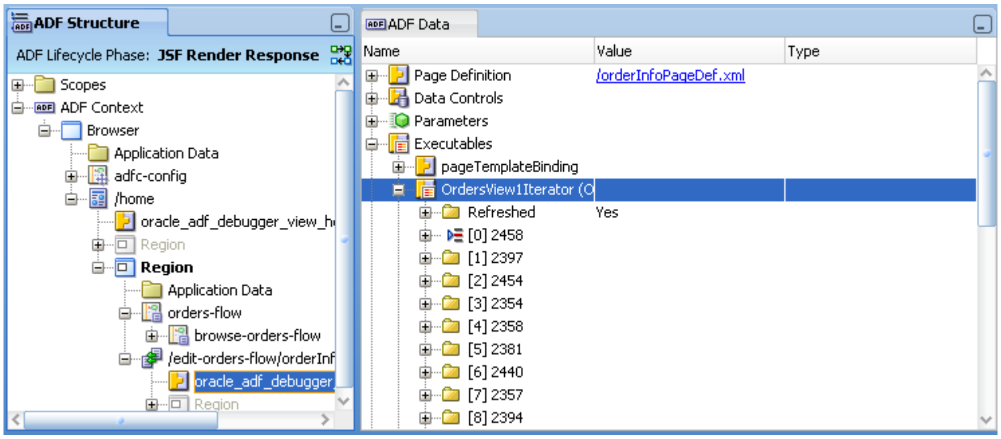
```
Scopes
ADF Context
  Browser
    Application Data
    adfc-config
    /home
      oracle_adf_debugger_view_homePageI
      Region
      Region
        Application Data
        orders-flow
          browse-orders-flow
          /edit-orders-flow/orderInfo
            oracle_adf_debugger_view_orc
            Region
```

Debugging: DefaultSe... | Breakpoints | ADF Data | EL Evaluator

Name	Value	Type
Page Definition	/orderInfoPageDef.xml	
Data Controls		
Parameters		
AppModuleOrdersDataC		
mName	"AppModuleOrdersDataControl"	
count	26	
hash	-912272927	
offset	0	
value	"AppModuleOrdersDataControl"	
mValue	null	
mExpression	"AppModuleOrdersDataControl"	
mEvaluated	false	
Executables		
Bindings		

Expanding the Executables node in the ADF Data window displays information similar to that shown in Figure 48-45.

Figure 48-45   Executables Selected for the ADF Data Window



ADF Structure

ADF Lifecycle Phase: **JSF Render Response**

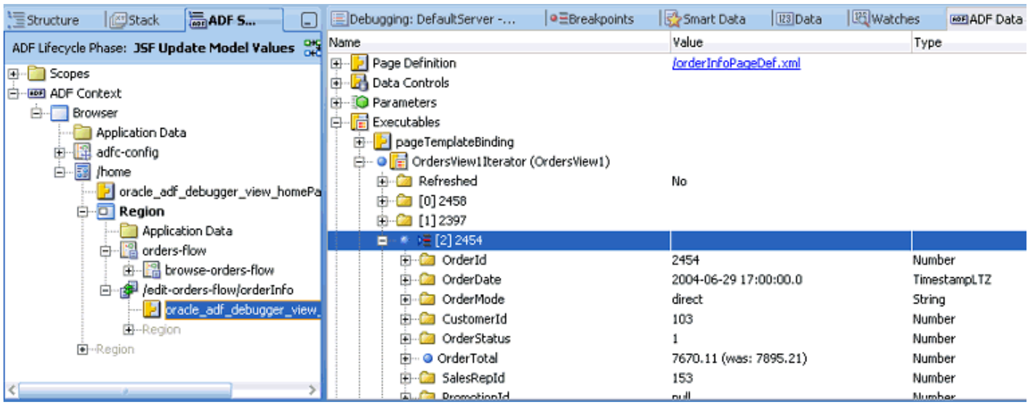
```
Scopes
ADF Context
  Browser
    Application Data
    adfc-config
    /home
      oracle_adf_debugger_view_h
      Region
      Region
        Application Data
        orders-flow
          browse-orders-flow
          /edit-orders-flow/orderInf
            oracle_adf_debugger
            Region
```

ADF Data

Name	Value	Type
Page Definition	/orderInfoPageDef.xml	
Data Controls		
Parameters		
Executables		
pageTemplateBinding		
OrdersView1Iterator (O		
Refreshed	Yes	
[0] 2458		
[1] 2397		
[2] 2454		
[3] 2354		
[4] 2358		
[5] 2381		
[6] 2440		
[7] 2357		
[8] 2394		

If a value has changed, the changed item will be marked with a blue dot to its left and the previous value is displayed in parenthesis. For instance, suppose the `OrderTotal` value has changed from 7895.81 to 7670.11. The ADF Data window places a blue dot next to `OrderTotal` and its parent `OrdersView1Iterator` and displays the current and previous values in the **Value** column, as shown in Figure 48-46.

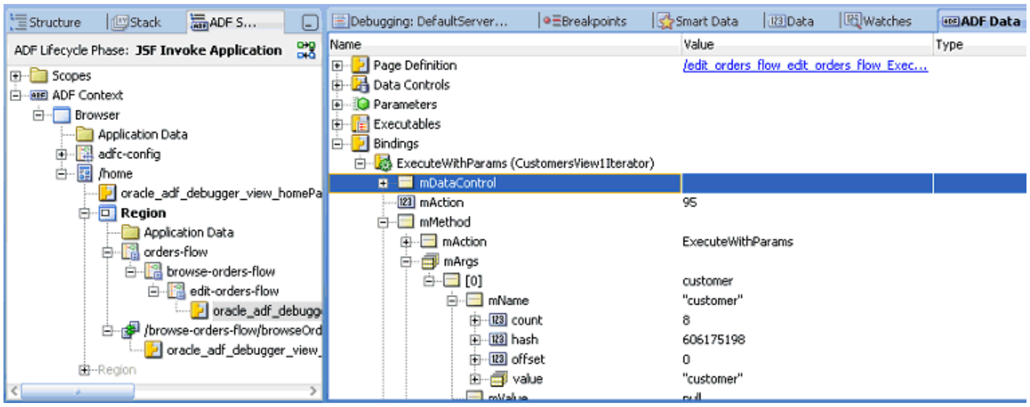
Figure 48-46 A Value Change Is Indicated by a Blue Dot in the ADF Data Window



Name	Value	Type
Page Definition	/orderInfoPageDef.xml	
Data Controls		
Parameters		
Executables		
pageTemplateBinding		
● OrdersView1Iterator (OrdersView1)		
Refreshed	No	
[0] 2458		
[1] 2397		
● [2] 2454		
OrderId	2454	Number
OrderDate	2004-06-29 17:00:00.0	TimestampLTZ
OrderMode	direct	String
CustomerId	103	Number
OrderStatus	1	Number
● OrderTotal	7670.11 (was: 7895.21)	Number
SalesRepId	153	Number
PromotionId	null	Number

Method binding information is displayed in the ADF Data window similar to what is shown in Figure 48-47.

Figure 48-47 Method Bindings Selected for the ADF Data Window

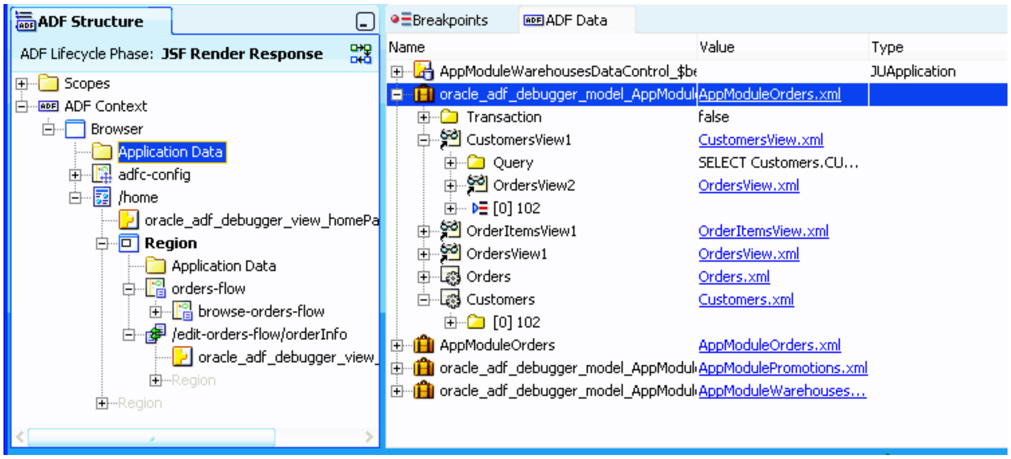


Name	Value	Type
Page Definition	/edit_orders_flow_edit_orders_flow_Exec...	
Data Controls		
Parameters		
Executables		
Bindings		
ExecuteWithParams (CustomersView1Iterator)		
mDataControl		
mAction	95	
mMethod		
mAction	ExecuteWithParams	
mArgs		
[0]	customer	
mName	"customer"	
count	8	
hash	606175198	
offset	0	
value	"customer"	
mValue	null	

When you select an Application Data node from the ADF Structure window, the ADF Data window displays the application objects, such as ADF Business Components objects, instantiated within the current data control frame for the corresponding view port (or **binding context** if ADF Controller is not used).

Business services implemented by ADF Business Components objects display the application data content, as shown in Figure 48-48 and described in Table 48-9.

Figure 48-48 Application Data for ADF Business Components Business Services



ADF Structure

ADF Lifecycle Phase: **JSF Render Response**

- Scopes
- ADF Context
  - Browser
    - Application Data
    - adfc-config
    - /home
      - oracle_adf_debugger_view_homePa
      - **Region**
        - Application Data
        - orders-flow
          - browse-orders-flow
        - /edit-orders-flow/orderInfo
          - oracle_adf_debugger_view_
          - Region
      - Region

Breakpoints | ADF Data

Name	Value	Type
AppModuleWarehousesDataControl_$be		JUApplication
oracle_adf_debugger_model_AppModul	AppModuleOrders.xml	
Transaction	false	
CustomersView1	CustomersView.xml	
Query	SELECT Customers.CU...	
OrdersView2	OrdersView.xml	
[0] 102		
OrderItemsView1	OrderItemsView.xml	
OrdersView1	OrdersView.xml	
Orders	Orders.xml	
Customers	Customers.xml	
[0] 102		
AppModuleOrders	AppModuleOrders.xml	
oracle_adf_debugger_model_AppModul	AppModulePromotions.xml	
oracle_adf_debugger_model_AppModul	AppModuleWarehouses...	

Table 48-9 ADF Data Window Content for Application Data

Binding Container	Description
Application module	The application module(s) of the corresponding view port data control frame will appear within the application data hierarchy as the root node(s). An application module design time icon will be used to identify the node(s). The application module node(s) will provide the following information: • Application module link - link to open the corresponding application module source file within the editor workspace. • Transaction - the application module current transaction status, if applicable. • View objects • Entity objects

Table 48-9 (Cont.) ADF Data Window Content for Application Data

Binding Container	Description
View object	View objects instantiated within the corresponding view port data control frame will appear underneath the corresponding application module root node as subordinate nodes. Design time icons will be used to identify them. Child view objects will appear subordinate to their parent view objects within the hierarchy. Named row sets will appear similar to view objects. Named iterators for a view object will appear similar to child view objects. Each view object node will provide the following information: • View object link - link to open the corresponding view object source file within the editor workspace. • Query - last executed view object SQL statement. Displays bind variables without value substitution. • Query with substitution - last executed view object SQL statement. Displays bind variables with value substitution. • Bind variables - last executed view object SQL statement bind variables and their values. • View object rows - each row displayed will be identified by its concatenated key values. The current row will be identified with a special icon. • Attributes - attributes contained on each row will display their current values along with their originating **entity object**. Transient attributes will also be displayed. • Modifications - changes made within the same request will be identified by a blue dot to left of attribute, row, and view objects node labels. Both the old and new value of the modification will be displayed.
Entity object	Entity objects instantiated within the corresponding view port data control frame will appear underneath the application module root node as subordinate nodes. Design time icons will be used to identify them. Each entity object node will provide the following information: • Entity object link - link to open the corresponding entity object source file within the editor workspace. • Entity object rows - each row displayed will be identified by its concatenated key values. The current entity-state and post-state of the row (e.g., `STATUS_MODIFIED`) will also be presented. • Attributes - attributes contained on each row will display their current values. • Modifications - changes made within the same request will be identified by a blue dot to left of attribute, row, and entity objects node labels. Both the old and new value of the modification will be displayed.

Business services implemented by non-ADF Business Components objects display application data content using raw member variables. The format is similar to the display of non-ADF Business Components content for the binding container, as shown in Figure 48-42.

What Happens When You Set an ADF Declarative Breakpoint

When you set an ADF declarative breakpoint, JDeveloper adds the breakpoint to the appropriate class, method, or other construct in the ADF source Java code that corresponds to the breakpoint. Once the breakpoint is set in the code, the standard Java debugger mechanism pauses application execution when the breakpoint is reached. When the breakpoint is reached, it will be identified by a red dot icon in the Breakpoints window. Depending on the type of declarative breakpoint that was reached, it will also appear as a red dot icon in the task flow activity, in the page definition breakpoint margins, or in the ADF Lifecycle Breakpoints window.

For task flow activity breakpoints, the debugger pauses the application within the JSF Invoke Application phase before the activity where the breakpoint is set. In other words, the activity where the breakpoint is set is not executed.

For task flow view activities, however, the application is paused within the JSF Render Response phase after the view activity is executed, but before the new page is rendered.

For a page definition Before executable breakpoint, the debugger pauses the application when the executable is refreshed. For a page definition Before action binding breakpoint, the debugger pauses the application when the binding is executed. For a page definition Before attribute value binding breakpoint, the debugger pauses the application before the attribute's `setInputValue()` method in the ADF source code is executed.

For a Before lifecycle breakpoint, the debugger pauses the application before it enters the next lifecycle phase. For an After lifecycle breakpoint, the debugger pauses the application after the lifecycle phase and before the next phase.

Setting Breakpoints in Java Code and Groovy Script

In most ADF applications, declarative breakpoints provide more than enough information to troubleshoot the application errors. But there might be situations where you may need to set breakpoints on specific classes or methods for further investigation. Using Java code and Groovy snippet breakpoints in combination with ADF declarative breakpoints will help you to identify these errors.

You can use the ADF Declarative Debugger to set breakpoints on Java classes and methods, as in any standard Java code debugger. You can use Java code breakpoints and Groovy snippet breakpoints in combination with ADF declarative breakpoints. For most ADF applications, ADF declarative breakpoints will provide enough debugging information to troubleshoot the application. For information about using ADF declarative breakpoints, see Setting ADF Declarative Breakpoints. However, you may need to set breakpoints on specific classes or methods for further inspection. Or, you may be debugging a non-ADF application, in which case, you can use Java code breakpoints.

JDeveloper provides a class locator feature that assists you in finding the class you want to break on. If you can obtain Oracle ADF source code, you can enhance your debugging by having access to various ADF classes and methods. For more information about getting ADF source code, see Using ADF Source Code with the Debugger. If you obtained the ADF source, you can further enhance the debugging experience by using the debug library version of the ADF source, as described in How to Use Debug Libraries for Symbolic Debugging.

How to Set Breakpoints in Groovy Script

When you define validation rules or attribute default values using Groovy script, you can set breakpoints on these Groovy snippets.

Before you begin:

It may be helpful to have an understanding of when to use code breakpoints instead of ADF declarative breakpoints. For more information, see Setting Breakpoints in Java Code and Groovy Script.

To set Groovy snippet breakpoints to debug an application:

1. In the Applications window, expand the entity object or view object that contains the Groovy script where you want to set a breakpoint, and then double-click the object’s `.bcs` file.
2. In the source editor, click in the margin next to the appropriate line to set a breakpoint, and then run the debugger.
3. When the application stops on the breakpoint, you can use the Data window to examine the local variables and arguments of the current context.

How to Set Java Breakpoints on Classes and Methods

You can set Java breakpoints on your classes and methods. If you have ADF source code, you can set Java breakpoints in the source as well. If you are debugging an ADF application, you should check to see whether ADF declarative breakpoints can be used instead of Java code breakpoints. For more information, see Setting ADF Declarative Breakpoints.

Before you begin:

It may be helpful to have an understanding of when to use Java code breakpoints instead of ADF declarative breakpoints. For more information, see Setting Breakpoints in Java Code and Groovy Script.

You will need to complete this task:

> Before you attempt to use breakpoints, you should try to run the application and look for missing or incomplete data, actions and methods that are ignored or incorrectly executed, or other unexpected results. If you did not find the problem, create a debugging configuration that will enable the ADF Log and send Oracle ADF messages to the Log window. For more information, see How to Create an Oracle ADF Debugging Configuration.

To set Java breakpoints to debug an application:

1. In the main menu, choose **Navigate** and then **Go To Java Type** (or press Ctrl+Minus) and use the dialog to locate the Oracle ADF class that represents the entry point for the processing failure.

> **Note:**
>
> JDeveloper will locate the class from the user interface project with current focus in the Applications window. If your workspace contains more than one user interface project, be sure that the one with the current focus is the one you want to debug.

2. Open the class file in the source editor and find the Oracle ADF method call that will enable you to step into the statements of the method.
3. Set a breakpoint on the desired method and run the debugger.
4. When the application stops on the breakpoint, use the Data window to examine the local variables and arguments of the current context.

> **Tip:**
>
> If you are using the **Go to source** context menu command in the Data, Watches, or Smart Data window, you can go back to the execution point by using the back button. You can also access the back button through the **Navigate** menu.

Once you have set breakpoints to pause the application at key points, you can proceed to view data in the Data window. To effectively debug your web page's interaction with the ADF Model layer, you need to understand:

- The ADF page lifecycle and the method calls that get invoked
- The local variables and arguments that the ADF Model layer should contain during the course of application processing

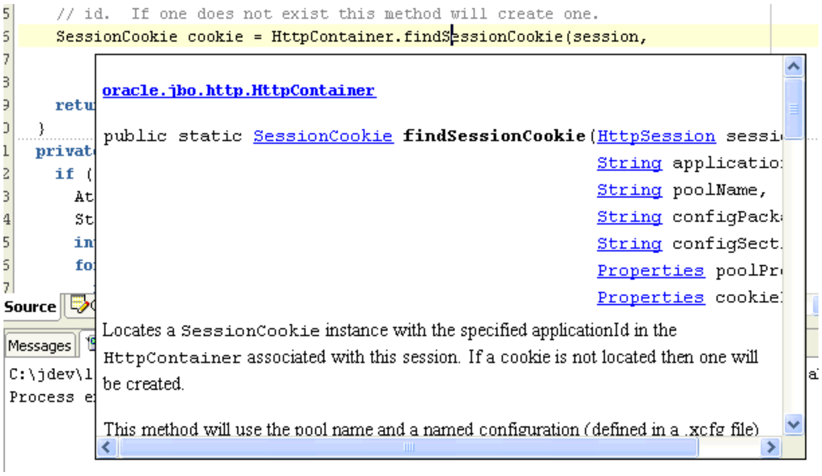
Awareness of Oracle ADF processing will give you the means to selectively set breakpoints, examine the data loaded by the application, and isolate the contributing factors.

> **Note:**
>
> JSF web pages may also use backing beans to manage the interaction between the page's components and the data. Debug backing beans by setting breakpoints for them as you would with any other Java class file.

How to Optimize Use of the Source Editor

Once you have added the ADF source library to your project, you have access to the helpful Quick Javadoc feature (Ctrl+D) that the source editor makes available. Figure 48-49 shows Quick Javadoc for a method like `findSessionCookie()`.

Figure 48-49 Using Quick Javadoc on ADF API in the Source Editor



```
    // id.  If one does not exist this method will create one.
    SessionCookie cookie = HttpContainer.findSessionCookie(session,

    retu
  }
  privat
    if (
      At
      St
      in
      fo
```

oracle.jbo.http.HttpContainer

```
public static SessionCookie findSessionCookie(HttpSession sessi
                                              String applicatio
                                              String poolName,
                                              String configPack
                                              String configSect
                                              Properties poolPr
                                              Properties cookie
```

Locates a SessionCookie instance with the specified applicationId in the HttpContainer associated with this session. If a cookie is not located then one will be created.

This method will use the pool name and a named configuration (defined in a .xcfg file)

```
C:\jdev\l
Process e
```

How to Set Breakpoints and Debug Using ADF Source Code

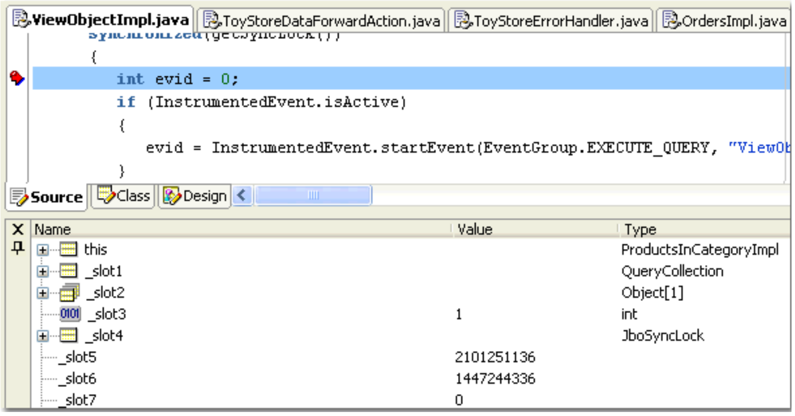
After loading the ADF source code, you can debug any Oracle ADF code for the current project the same way that you do your own Java code. This means that you can press Ctrl+Minus to type in any class name in Oracle ADF, and JDeveloper will open its source file automatically so that you can set breakpoints as desired.

How to Use Debug Libraries for Symbolic Debugging

When debugging Oracle ADF source code, by default you will not see symbol information for parameters or member variables of the currently executing method.

For example, in a debugging session without ADF source code debug libraries, you may see unrecognizable names such as "_slot", as shown in Figure 48-50.

Figure 48-50 Local Symbols Are Hard to Understand Without Debug Libraries



```
    synchronized(getSyncLock())
    {
        int evid = 0;
        if (InstrumentedEvent.isActive)
        {
            evid = InstrumentedEvent.startEvent(EventGroup.EXECUTE_QUERY, "ViewOb
        }
```

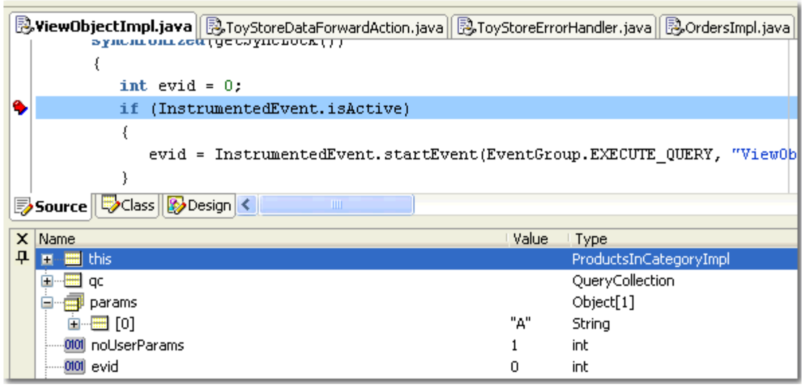
Name	Value	Type
this		ProductsInCategoryImpl
_slot1		QueryCollection
_slot2		Object[1]
_slot3	1	int
_slot4		JboSyncLock
_slot5	2101251136	
_slot6	1447244336	
_slot7	0	

These names are hard to decipher and make debugging more difficult. You can make debugging easier by using the debug versions of the ADF JAR files supplied along with the source while debugging in your development environment.

> **Note:**
>
> The supplied debug libraries should not be used in a test or production environment, since they typically have slightly slower runtime performance than the optimized JAR files shipped with JDeveloper.

The debug library JARs are versions of Oracle ADF JARs that have been compiled with additional debug information. When you use these debug JAR files instead of the default optimized JARs, you will see all of the information in the debugger. For example, the variable evid is now identified by its name in the debugger, as shown in Figure 48-51.

Figure 48-51 Symbol Information Displayed in the Debugger



```
synchronized(getSyncLock())
{
   int evid = 0;
   if (InstrumentedEvent.isActive)
   {
      evid = InstrumentedEvent.startEvent(EventGroup.EXECUTE_QUERY, "ViewOb
   }
```

Name	Value	Type
this		ProductsInCategoryImpl
qc		QueryCollection
params		Object[1]
[0]	"A"	String
noUserParams	1	int
evid	0	int

Before you begin:

It may be helpful to have an understanding of when to use Java code breakpoints instead of ADF declarative breakpoints. For more information, see Setting Breakpoints in Java Code and Groovy Script.

You will need to complete this task:

Before you replace the standard library JAR, make sure that JDeveloper is not running. If it's currently running, exit from the product before proceeding.

To replace the standard library JARs with the debug library JARs:

1. With JDeveloper closed, make a backup subdirectory of all existing optimized JAR files in the ./BC4J/lib directory of your JDeveloper installation. For example, assuming jdev11 is the JDeveloper home directory:

   ```
   C:\jdev11\BC4J\lib> mkdir backup
   C:\jdev11\BC4J\lib> copy *.jar backup
   ```

2. For each ADF library that you want debug symbols for while debugging, copy the _g.jar version of the matching library over the existing, corresponding library in the C:\jdev11\BC4J\lib directory.

   This is safe to do since you made a backup of the optimized JAR files in the backup directory in Step 2.

Since debug libraries typically run a little slower than libraries compiled without debug information, this diagnostic message is to remind you not to use debug libraries for performance timing:

```
*************************************************************************
*** WARNING: Oracle BC4J debug build executing - do not use for timing ***
*************************************************************************
```

3. To change back to the optimized libraries, simply copy the JAR file(s) in question from the `./BC4J/lib/backup` directory back to the `./BC4J/lib` directory.

What You May Need to Know About Setting Java Code Breakpoints

You first need to understand the different kinds of Java code breakpoints and where to create them.

To see the debugger Breakpoints window, choose **Window** and then **Breakpoints** in the main menu (or press Ctrl+Shift+R).

You can create a new Java code breakpoint by choosing **Create Breakpoint** from the context menu in the Breakpoints window. The **Breakpoint Type** dropdown list controls what kind of breakpoint you will create, as shown in Table 48-10.

> **Note:**
>
> You can also use the Create Breakpoint dialog to create an ADF lifecycle phase declarative breakpoint. For information about creating ADF declarative breakpoints, see How to Set and Use ADF Lifecycle Phase Breakpoints.

Table 48-10 Different Types of Java Breakpoints

Breakpoint Type	The Breakpoint Occurs Whenever	Usage
Exception	An exception of this class (or a subclass) is thrown.	An Exception breakpoint is useful when you don't know where the exception occurs, but you know what kind of exception it is (for example, `java.lang.NullPointerException`, `java.lang.ArrayIndexOutOfBoundsException`, `oracle.jbo.JboException`). The checkbox options allow you to control whether to break on caught or uncaught exceptions of this class. The **Browse** button helps you find the fully qualified class name of the exception. The Exception Class combobox remembers the most recently used exception breakpoint classes. Note that this is the default breakpoint type when you create a breakpoint in the breakpoints window.

Table 48-10 (Cont.) Different Types of Java Breakpoints

Breakpoint Type	The Breakpoint Occurs Whenever	Usage
Source	A particular source line in a particular class in a particular package is run.	You rarely create a source breakpoint in the Create Breakpoint dialog, because it's much easier to create it by first using the **Navigate > Go to Java Type** menu (accelerator Ctrl+Minus), then scrolling to the line number you want — or using **Navigate > Go to Line** (accelerator Ctrl+G) — and finally clicking in the breakpoint margin at the left of the line you want to break on. This is equivalent to creating a new source breakpoint, but it means you don't have to type in the package, class, and line number by hand.
Method	A method in a given class is invoked.	The Method breakpoint is useful for setting breakpoints on a particular method you might have seen in the call stack while debugging a problem. If you have the source, you can set a source breakpoint wherever you want in that class, but this kind of breakpoint lets you stop in the debugger even when you don't have source for a class.
Class	Any method in a given class is invoked.	The Class breakpoint can be used when you might know the class involved in the problem, but not the exact method you want to stop on. This kind of breakpoint does not require source. The **Browse** button helps you quickly find the fully qualified class name you want to break on.
Watchpoint	A given field is accessed or modified.	The Watchpoint breakpoint can be used to find a problem if the code inside a class modifies a member field directly from several different places (instead of going through setter or getter methods each time). You can pause the debugger when any field is modified. You can create a breakpoint of this type by using the **Toggle Watchpoint** menu item on the context menu when pointing at a member field in your class's source.

How to Edit Breakpoints for Improved Control

After creating a Java code breakpoint you can edit the breakpoint in the Breakpoints window by right-clicking it and choosing **Edit** in the context menu.

> **Note:**
>
> You can use the Edit Breakpoint dialog to edit an ADF declarative breakpoint. However, you cannot edit some of the other information such as the information in the **Definition** tab. You can launch the Edit Breakpoint dialog by choosing **Edit** from the context menu in the Breakpoints window. For information about creating ADF declarative breakpoints, see Setting ADF Declarative Breakpoints.

Some of the features you can use by editing your breakpoint are:

- Associate a logical "breakpoint group" name to group this breakpoint with others of the same group name. Breakpoint groups make it easy to enable/disable an entire set of breakpoints in one operation.
- Associate a debugger action to a breakpoint when the breakpoint is hit. The default action is to stop the debugger so that you can inspect the application states, but you can add a sound alert, write information to a log file, and enable or disable group of breakpoints.
- Associate a conditional expression with the breakpoint so that the debugger stops only when that condition is met. The expressions can be virtually any boolean expression, including:
  - `expr ==value`
  - `expr.equals("value")`
  - `expr instanceof.fully.qualified.ClassName`

  > **Note:**
  >
  > Use the debugger Watches window to evaluate the expression first to make sure it's valid.

How to Filter Your View of Class Members

You can use the debugger to filter the members that are displayed in the debugger window for any class. In the debugger's Data window, selecting any item and choosing **Preferences** from the context menu brings up a dialog that lets you customize which members appear in the debugger and (more importantly sometimes) which members *don't* appear. You can filter by class type to simplify the amount of scrolling you need to do in the debugger Data window. This is especially useful when you might be interested only in a handful of a class's members.

What You May Need to Know About Oracle ADF Breakpoints

If you loaded Oracle ADF source code, you can use the breakpoints listed in Table 48-11 to debug your application.

By looking at the Stack window when you hit these breakpoints, and stepping through the source, you can get a better idea of what's going on.

Table 48-11 Commonly Used ADF Breakpoints

Breakpoint	Breakpoint Type	Usage
`oracle.jbo.JboException`	Exception	This breakpoint useful for setting a breakpoint on the base class of all ADF Business Components runtime exceptions.
`oracle.jbo.DMLException`	Exception	This is the base class for exceptions originating from the database, like a failed DML operation due to an exception raised by a trigger or by a constraint violation.
`doIt()`	Method	You can also perform the same debugging function by setting an ADF declarative breakpoint on the page definition action binding. See How to Set and Use Page Definition Action Binding Breakpoints. If you prefer to use this Java breakpoint, you can find it in the `JUCtrlActionBinding` class (`oracle.jbo.uicli.binding` package). This is the method that will execute when any ADF action binding is invoked, and you can step into the logic and look at parameters if relevant.
`oracle.jbo.server.ViewObjectImpl.executeQueryForCollection`	Method	This is the method that will be called when a view object executes its SQL query.
`oracle.jbo.server.ViewRowImpl.setAttributeInternal`	Method	This is the method that will be called when any view row attribute is set.
`oracle.jbo.server.EntityImpl.setAttributeInternal`	Method	You can also perform the same debugging function by setting an ADF declarative breakpoint on the page definition attribute value binding. See How to Set and Use Page Definition Value Binding Breakpoints. This is the method that will be called when any entity object attribute is set.

Debugging Groovy Expressions in Business Components

JDeveloper allows you to debug Groovy scripts/expressions. To start the debugger, right-click on the applicationmodule.xml file to start the Oracle ADF Model Tester or right click on the Java file containing the main method to start executing the Java program.

The `.bcs` file contains Groovy expressions/scripts. The `.bcs` file is associated with an entity object or view object. When Groovy expressions are debugged, if there is an exception, the debugger pauses in the `.bcs` file at the line where the exception occurred and displays the errors. If there is no exception, the debugger proceeds to execute the Groovy expressions, and displays the necessary computational data (in the data/smart data/watch etc windows). Just as you can set breakpoints in a Java source code file, JDeveloper allows you to place breakpoints in a `.bcs` file. When the debugger is run and if the debugger encounters a breakpoint in this `.bcs` file, the debugger pauses execution at the breakpoint.

The process of debugging Groovy scripts in a `.bcs` file is similar to that of a Java file. When the debugger is in the `.bcs` file, use the various step operations, namely Step

Into, Step Over, and Step Out to control the execution of the Groovy scripts by the debugger. At any point when the debugger is in the `.bcs` file, either at a breakpoint, or when the debugger has paused execution at a line because you used the step operations, use the debugger log windows to view execution data.

How to Debug Groovy Expressions

JDeveloper allows you to debug Groovy expressions. To debug Groovy expressions, right-click on the `applicationmodule.xml` file and select Debug to start the Oracle ADF Model Tester or right-click on the Java file containing the main method to start executing the Java program.

To start debugging Groovy expressions, do any of the following two steps.

1. Right-click on an application module (`applicationmodule.xml` file) and select **Debug** to start the Oracle ADF Model Tester.
2. Right-click on the Java file containing the main method and select **Debug** from the context menu to start executing the Java program. Alternatively, double click on the Java file to open it; right-click inside the Java file and select **Debug** from the context menu.

How to Control Groovy Debugging

The application allows you to place breakpoints in the `.bcs` file and use the step operations (Step Into, Step Over, Step Out) to control the execution of the debugger.

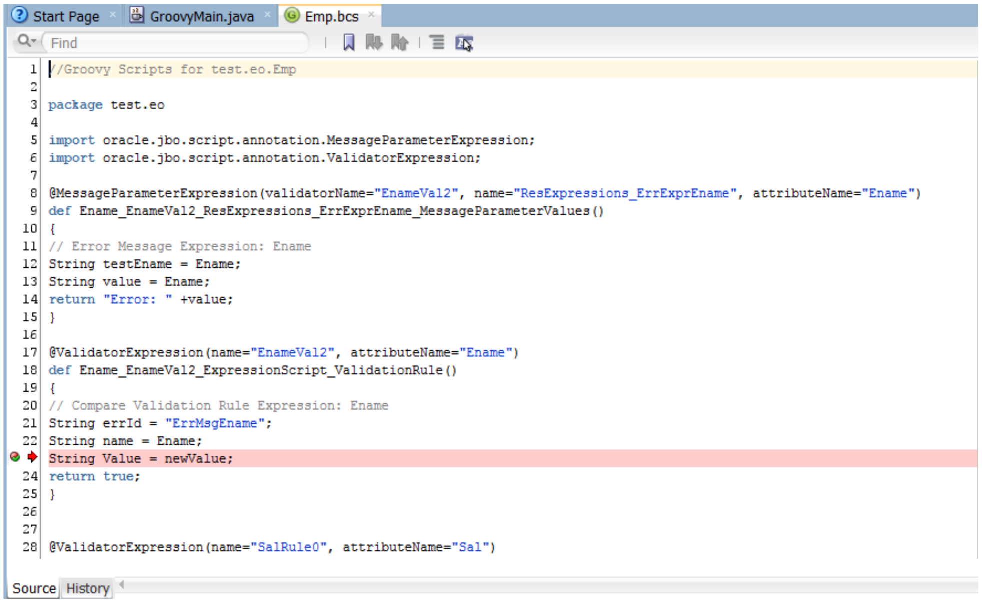
To read more about breakpoints, refer to Managing Breakpoints.

To control Groovy debugging:

1. Double-click on the `.bcs` file to open it.

   Figure 48-52 shows breakpoints placed in the `.bcs` file. The debugger executes the Groovy scripts, and if the debugger encounters a breakpoint in the `.bcs` file, the debugger pauses at this breakpoint. You also use the Step Into, Step Over, Run to Cursor to control Groovy debugging.

Figure 48-52   Breakpoints placed in `.bcs` file



```
//Groovy Scripts for test.eo.Emp

package test.eo

import oracle.jbo.script.annotation.MessageParameterExpression;
import oracle.jbo.script.annotation.ValidatorExpression;

@MessageParameterExpression(validatorName="EnameVal2", name="ResExpressions_ErrExprEname", attributeName="Ename")
def Ename_EnameVal2_ResExpressions_ErrExprEname_MessageParameterValues()
{
// Error Message Expression: Ename
String testEname = Ename;
String value = Ename;
return "Error: " +value;
}

@ValidatorExpression(name="EnameVal2", attributeName="Ename")
def Ename_EnameVal2_ExpressionScript_ValidationRule()
{
// Compare Validation Rule Expression: Ename
String errId = "ErrMsgEname";
String name = Ename;
String Value = newValue;
return true;
}


@ValidatorExpression(name="SalRule0", attributeName="Sal")
```

2. When the debugger has paused at the breakpoint in the `.bcs` file, use the Debug menu, toolbar icons, or keyboard equivalents to use step operations. They are namely Step Into, Step Over, and Step Out.

   Refer to How to Step Into a Method to read more about step operations.
   The application also allows you to run the debugger at the cursor location. Refer How to Run to the Cursor Location to read more.

   The application allows you to pause the debugger while it is running. You can then inspect the state of the program and then continue to run the debugger. Refer How to Pause and Resume the Debugger to read more on how to pause and resume the debugger.

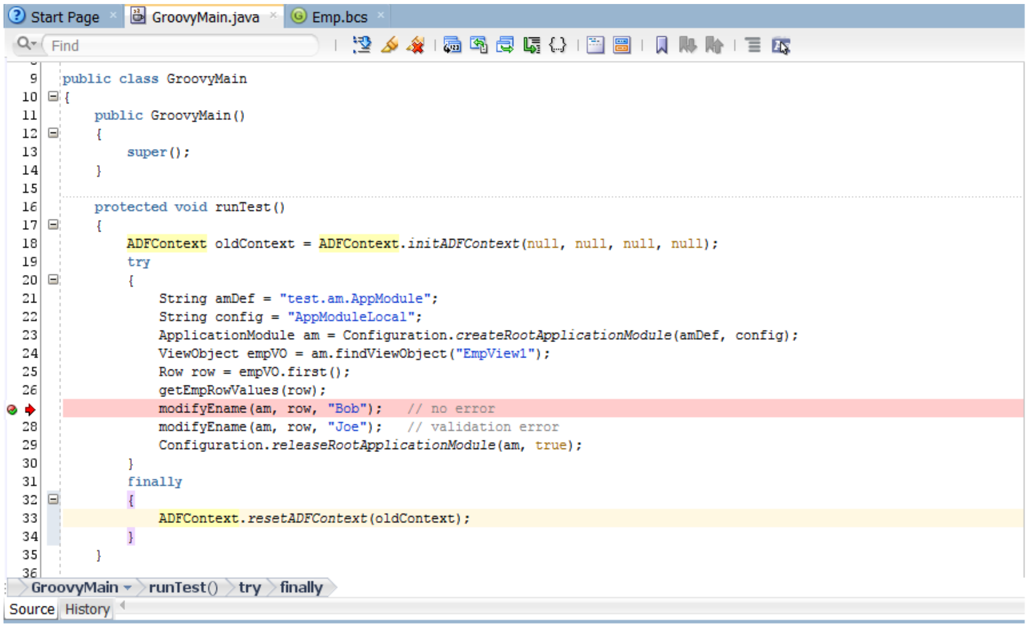
   At any point you can terminate the debugging session. Refer How to Terminate a Debugging Session to read on how to terminate a debugging session.

What Happens When You Debug Groovy Expressions

The application debugs Groovy scripts in a `.bcs` file. Place breakpoints in the `.bcs` file if required to pause the execution of the debugger at these breakpoints. Inspect the information in the data/watch/smart data etc windows whenever the debugger pauses at a breakpoint or when you step operations.

The debugger executes the Groovy scripts and displays information in the data/watch/smart data etc windows. Figure 48-53 displays a breakpoint set at line 27, and the debugger has paused at this line, which is the current execution line (indicated by the red arrow). The current execution line indicates a call to the method `modifyEname()`. This method attempts to modify the Ename attribute of the Emp entity.

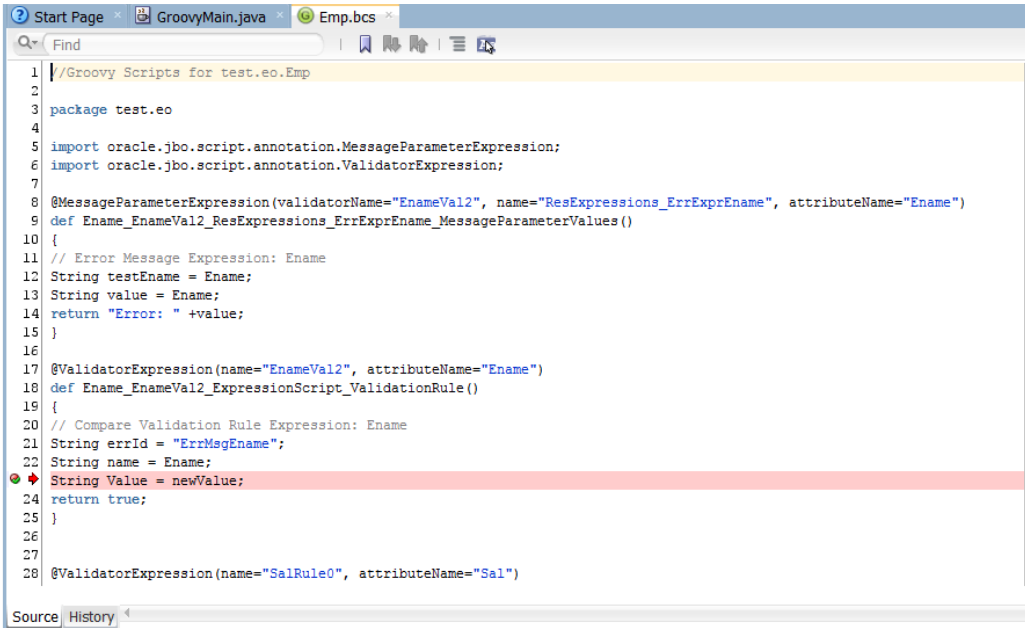
Figure 48-53   Breakpoint placed in Java file at line that references Groovy expressions



```
 9  public class GroovyMain
10  {
11      public GroovyMain()
12      {
13          super();
14      }
15
16      protected void runTest()
17      {
18          ADFContext oldContext = ADFContext.initADFContext(null, null, null, null);
19          try
20          {
21              String amDef = "test.am.AppModule";
22              String config = "AppModuleLocal";
23              ApplicationModule am = Configuration.createRootApplicationModule(amDef, config);
24              ViewObject empVO = am.findViewObject("EmpView1");
25              Row row = empVO.first();
26              getEmpRowValues(row);
                modifyEname(am, row, "Bob");   // no error
28              modifyEname(am, row, "Joe");   // validation error
29              Configuration.releaseRootApplicationModule(am, true);
30          }
31          finally
32          {
33              ADFContext.resetADFContext(oldContext);
34          }
35      }
36
```

A validator is defined for the Ename attribute. The validator is defined using Groovy script/expression. This Groovy script (along with other Groovy scripts defined for other use cases) reside in a `.bcs` file. So when you Step Into this method at line 27, the application accesses this `.bcs` file and executes the validator. Figure 48-54 shows the debugger paused at line 23 in the `.bcs` file at a pre-set breakpoint, whilst executing the validator.

Figure 48-54   Debugger navigates from Java file to .bcs file and pauses at breakpoint in `.bcs` file



```
 1  //Groovy Scripts for test.eo.Emp
 2
 3  package test.eo
 4
 5  import oracle.jbo.script.annotation.MessageParameterExpression;
 6  import oracle.jbo.script.annotation.ValidatorExpression;
 7
 8  @MessageParameterExpression(validatorName="EnameVal2", name="ResExpressions_ErrExprEname", attributeName="Ename")
 9  def Ename_EnameVal2_ResExpressions_ErrExprEname_MessageParameterValues()
10  {
11  // Error Message Expression: Ename
12  String testEname = Ename;
13  String value = Ename;
14  return "Error: " +value;
15  }
16
17  @ValidatorExpression(name="EnameVal2", attributeName="Ename")
18  def Ename_EnameVal2_ExpressionScript_ValidationRule()
19  {
20  // Compare Validation Rule Expression: Ename
21  String errId = "ErrMsgEname";
22  String name = Ename;
    String Value = newValue;
24  return true;
25  }
26
27
28  @ValidatorExpression(name="SalRule0", attributeName="Sal")
```

At this point, resume debugging, or terminate debugging, or use step operations. Refer How to Control Groovy Debugging to read more.

Regression Testing with JUnit

JUnit is an open source regression testing framework for Java which is used to write and run tests that verify Java code. Tests written in JUnit help you write code at an extreme pace and spot defects quickly. In JDeveloper, you can use wizards to create test fixtures, cases, and suites.

Testing your business services is an important part of your application development process. By creating a set of JUnit regression tests that exercise the functionality provided by your application module, you can ensure that new features, bug fixes, or refactorings do not destabilize your application. JDeveloper's integrated support for creating JUnit regression tests makes it easy test your application. Its integrated support for running JUnit tests means that any developer on the team can run the test suite with a single mouse click, greatly increasing the chances that every team member can run the tests to verify their own changes to the system. Furthermore, by using JDeveloper's integrated support for creating and running Apache Ant build scripts, you can easily incorporate running the tests into your automated build process as well. You can create a JUnit test for your application module, run it, and integrate the tests into an Ant build script.

JDeveloper provides the ability to generate JUnit test cases, test fixtures, and test suites. You can create test cases to test individual Java files containing single or multiple Java classes. You can create JUnit test fixtures that can be reused by JUnit test cases. You can group all these test cases into a JUnit test suite, which you can run together as a unit.

You can also use the JUnit BC4J Test Suite wizard to generate a test suite when there is an application module in the project. The wizard generates a test suite, test fixture, and a test case for each view object in the application module.

You can create a separate project to contain your regression tests or to integrate the test files into an existing project. If you are creating an ADF Business Components test, you should create a separate project for testing.

Creating separate projects for testing has the following advantages:

- The ability to compile the base project without having a dependency on JUnit
- The ability to package the base project for deployment without having to exclude the test classes.

If you are creating separate projects for JUnit testing, you should create directory structures that mirror the structure of the packages being tested. You may want to name the test classes using a naming convention that can easily identify the package being tested. For example, if you are testing `myClass.java`, you can name the test class `myClassTest.java`.

Although having separate projects has many advantages, in certain cases it may be easier to include the tests within the project.

You can use the Create Test wizards in the context of the project to create a JUnit test case, test fixture, or test suite. However, if you do not want to include these tests as part of the deployment, you may want to separate the tests out in their own project.

Tip:

If you don't see the Create Test wizards, use JDeveloper's **Help > Check for Updates** feature to install the JUnit Integration extension before continuing.

Each test case class contains a `setUp()` and `tearDown()` method that JUnit invokes to allow initializing resources required by the test case and to later clean them up. These test case methods invoke the corresponding `setUp()` and `tearDown()` methods to prepare and clean up the test fixture for each test case execution. Any time a test in the test case needs access to the application module, it uses the test fixture's `getApplicationModule()` method. The method returns the same application module instance, saved in a member field of the test fixture class, between the initial call to `setUp()` and the final call to `tearDown()` at the end of the test case.

JDeveloper supports JUnit 4, which allows annotations to be used instead of explicitly having to name the methods `setUp()` and `tearDown()`.These annotations — `@Before`, `@After` — allow you to have multiple setup and teardown methods, including inherited ones if required.

The generated `ExampleModuleConnectFixture` is a JUnit test fixture that encapsulates the details of acquiring and releasing an application. It contains a `setUp()` method that uses the `createRootApplicationModule()` method of the `Configuration` class to create an instance of an application module. Its `tearDown()` method calls the matching `releaseRootApplicationModule()` method to release the application module instance.

Your own testing methods can use any of the programmatic APIs available in the `oracle.jbo` package to work with the application module and view object instances in its data model. You can also cast the `ApplicationModule` interface to a custom interface to have your tests invoke your custom service methods as part of their job. During each test, you will call one or more `assertXxx()` methods provided by the JUnit framework to assert what the expected outcome of a particular expression should be. When you run the test suite, if any of the tests in any of the test cases contains assertions that fail, the JUnit Test Runner window displays the failing tests with a red failure icon.

The JUnit test generation wizard generates skeleton test case classes for each view object instance in the data model, each of which contains a single test method named `testAccess()`. This method contains a call to the `assertNotNull()` method to test that the view object instance exists.

```
// In ViewInstanceNameTest.java test case class
  public void testSomeMeaningfulName() {
  // test assertions here
  }
```

Each generated test case can contain one or more test methods that the JUnit framework will execute as part of executing that test case. You can add a test to the test case simply by creating a public void method in the class whose name begins with the prefix `test` or use the annotation `@Test`.

How to Obtain the JUnit Extension

JUnit must be loaded as an extension to JDeveloper before it becomes available and appears in the menu system.

Before you begin:

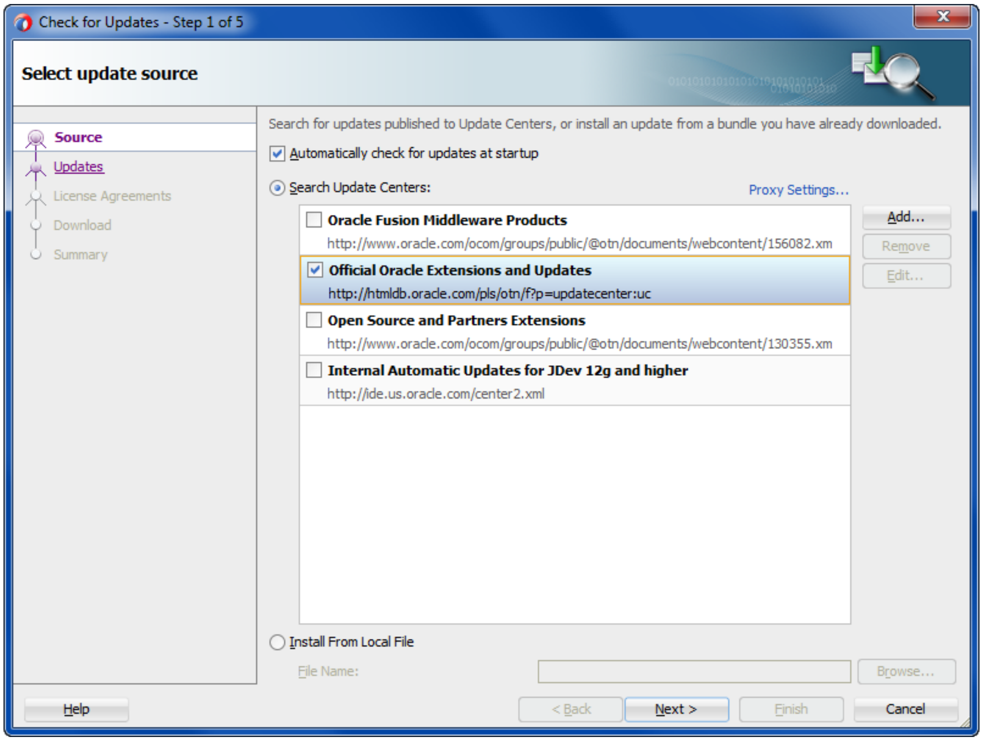
It may be helpful to have an understanding of the using JUnit regression testing to create test cases. For more information, see Regression Testing with JUnit.

To load the JUnit extension:

1. In the main menu, choose **Help** and then **Check For Updates**.
2. In the Source page of the Check for Updates dialog, select **Search Update Centers** and **Official Oracle Extensions and Updates** and click **Next**.

   If you have the JUnit zip file or if the JUnit selection does not appear in the **Available Updates** list, select **Install From Local File** to load the JUnit zip file.
3. In the Updates page, select **JUnit Integration** and click **Next**, as shown in Figure 48-55.

Figure 48-55   Check for Updates Wizard for Adding JUnit Extension



Check for Updates - Step 1 of 5

Select update source

Search for updates published to Update Centers, or install an update from a bundle you have already downloaded.

☑ Automatically check for updates at startup

● Search Update Centers:

☐ Oracle Fusion Middleware Products
http://www.oracle.com/ocom/groups/public/@otn/documents/webcontent/156082.xm

☑ Official Oracle Extensions and Updates
http://htmldb.oracle.com/pls/otn/f?p=updatecenter:uc

☐ Open Source and Partners Extensions
http://www.oracle.com/ocom/groups/public/@otn/documents/webcontent/130355.xm

☐ Internal Automatic Updates for JDev 12g and higher
http://ide.us.oracle.com/center2.xml

○ Install From Local File

File Name:

4. On the License Agreements page, click **I Accept** and click **Finish**.

How to Create a JUnit Test Case

Before you create a JUnit test case, you must have created a project that is to be tested.

Before you begin:

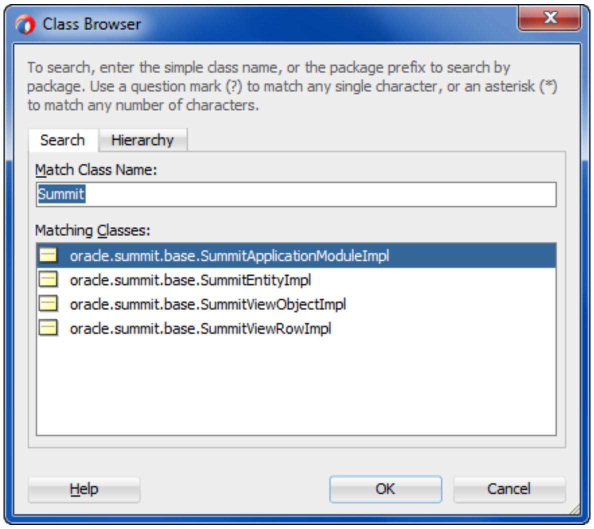
It may be helpful to have an understanding of the using JUnit regression testing to create test cases. For more information, see Regression Testing with JUnit.

To generate a JUnit test case:

1. In the Applications window, right-click the project in which want to create a test case and choose **New** and then **From Gallery**.
2. In the New Gallery, expand **General**, select **Unit Tests** and then **Test Case**, and click **OK**.
3. In the Select the Class to Test page of the Create Test Case dialog, enter the class under test or click **Browse**.
4. In the Class Browser dialog, locate the class you want to test or enter the beginning letters in the **Match Class Name** field. The **Match Class** list will be filtered for easier identification.

   For example, entering `Summit` filters the list down to four items, as shown in Figure 48-56.

Figure 48-56 Class Browser for Selecting Class Files to Test



To search, enter the simple class name, or the package prefix to search by package. Use a question mark (?) to match any single character, or an asterisk (*) to match any number of characters.

Match Class Name: Summit

Matching Classes:

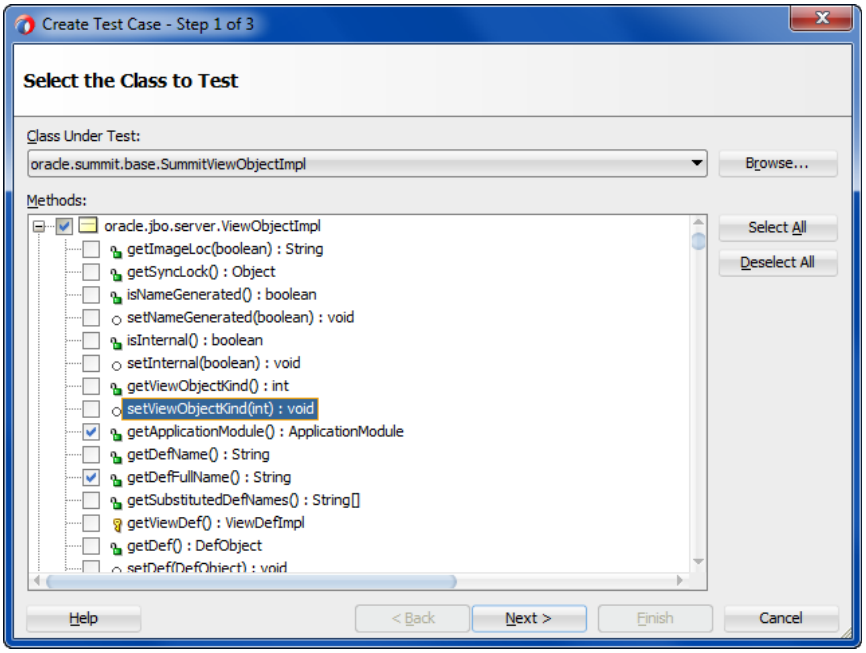
oracle.summit.base.SummitApplicationModuleImpl
oracle.summit.base.SummitEntityImpl
oracle.summit.base.SummitViewObjectImpl
oracle.summit.base.SummitViewRowImpl

   Select the class and click **OK** to close the dialog. Click **Next**.

5. Select the individual methods you want to test, and click **Next**.

   For example, in Figure 48-57, the four methods that are checked are to be tested.

Figure 48-57 Create Test Case Dialog for Selecting Methods to Test



Create Test Case - Step 1 of 3

Select the Class to Test

Class Under Test: oracle.summit.base.SummitViewObjectImpl

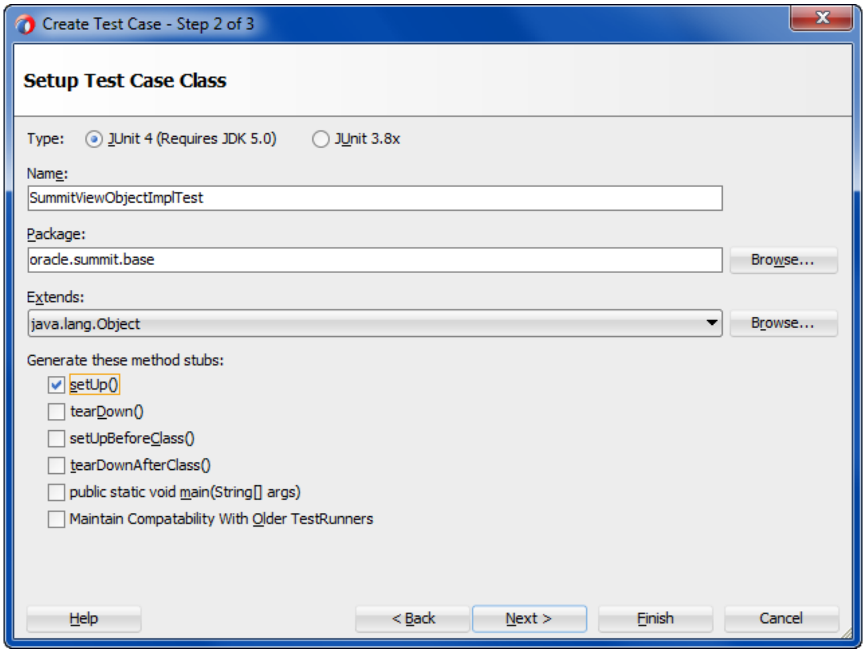
Methods:

- ☑ oracle.jbo.server.ViewObjectImpl
  - □ getImageLoc(boolean) : String
  - □ getSyncLock() : Object
  - □ isNameGenerated() : boolean
  - □ setNameGenerated(boolean) : void
  - □ isInternal() : boolean
  - □ setInternal(boolean) : void
  - □ getViewObjectKind() : int
  - □ setViewObjectKind(int) : void
  - ☑ getApplicationModule() : ApplicationModule
  - □ getDefName() : String
  - ☑ getDefFullName() : String
  - □ getSubstitutedDefNames() : String[]
  - □ getViewDef() : ViewDefImpl
  - □ getDef() : DefObject
  - □ setDef(DefObject) : void

6. In the Setup Test Case Class page, enter the name of the test case, the package, and the class it extends and select the list of built-in functions JUnit will create stubs for. Click **Next**.

   For example, in Figure 48-58, JUnit will create a stub for the `setUp()` method for the `SummitViewObjectImplTest` test case in the `oracle.summit.base` package of the Summit sample application for Oracle ADF.

Figure 48-58 Create Test Case Dialog for Setting Up Classes to Test



Create Test Case - Step 2 of 3

Setup Test Case Class

Type: ● JUnit 4 (Requires JDK 5.0) ○ JUnit 3.8x

Name: SummitViewObjectImplTest

Package: oracle.summit.base

Extends: java.lang.Object

Generate these method stubs:

- ☑ setUp()
- □ tearDown()
- □ setUpBeforeClass()
- □ tearDownAfterClass()
- □ public static void main(String[] args)
- □ Maintain Compatability With Older TestRunners

7. In the Select Test Fixtures page, select any test fixtures you want to add to the test case or click **Browse**.
8. Make sure that all the test fixtures you want to add to the test case are selected in the list and click **Finish**.

How to Create a JUnit Test Fixture

You should create a JUnit test fixture if you require more than one test for a class or method. A JUnit text fixture allows you to avoid duplicating test code that is needed to initialize testing.

Before you begin:

It may be helpful to have an understanding of the using JUnit regression testing to create test cases. For more information, see Regression Testing with JUnit.

To generate a JUnit test fixture:

1. In the Applications window, right-click the project in which want to create a test fixture and choose **New** and then **From Gallery**.
2. In the New Gallery, expand **General**, select **Unit Tests** and then **Test Fixture**, and click **OK**.
3. In the Create Test Fixture dialog, enter the name of the test fixture, the package, and any class it extends.
4. Click **OK**.

How to Create a JUnit Test Suite

Before you create a JUnit test suite, you should have already created JUnit test cases that can be added to the test suite.

Before you begin:

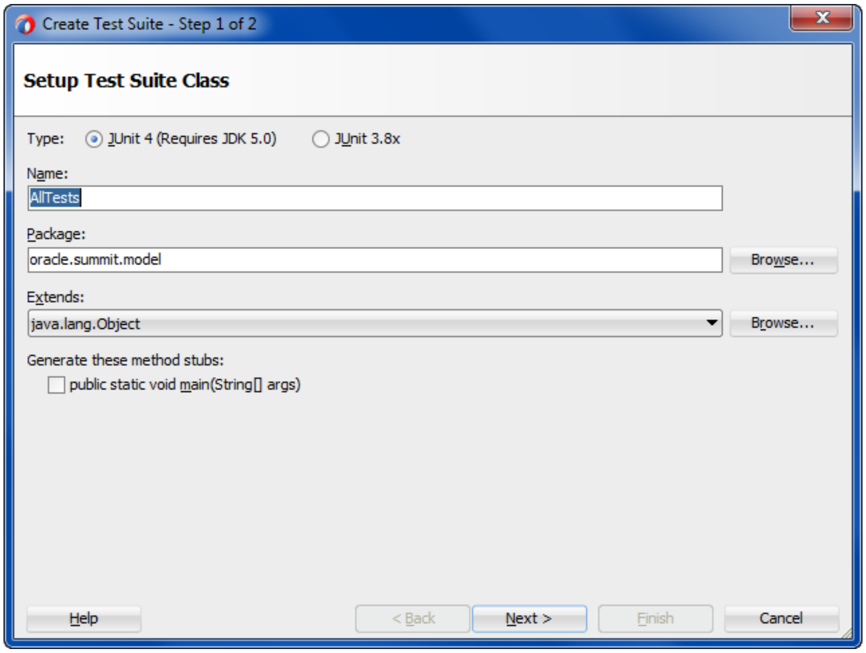
It may be helpful to have an understanding of the using JUnit regression testing to create test cases. For more information, see Regression Testing with JUnit.

To generate a JUnit test suite:

1. In the Applications window, right-click the project in which want to create a test suite and choose **New** and then **From Gallery**.
2. In the New Gallery, expand **General**, select **Unit Test** and then **Test Suite**, and click **OK**.
3. In the Setup Test Suite Class page of the Create Test Suite dialog, enter the name of the test suite, the package, and the class it extends. Click **Next.**

   For example, in Figure 48-59, an `AllTests` test suite is created that extends the `java.lang.Object` class.

Figure 48-59 Create Test Suite Wizard



Create Test Suite - Step 1 of 2

Setup Test Suite Class

Type: ● JUnit 4 (Requires JDK 5.0) ○ JUnit 3.8x

Name: AllTests

Package: oracle.summit.model

Extends: java.lang.Object

Generate these method stubs:

□ public static void main(String[] args)

4. In the Select Test Cases page of the Create Test Suite dialog, check that all the test cases you want included in the test suite have been selected. The test cases you have created will populate the list. Deselect any test cases that you do not want included. Click **Finish.**

How to Create a Business Components Test Suite

The test fixture that is created is a singleton class to reduce the number of connections. If you want to connect or disconnect for each test case, customize the test case using the JUnit 4 annotations `@Before` and `@After`.

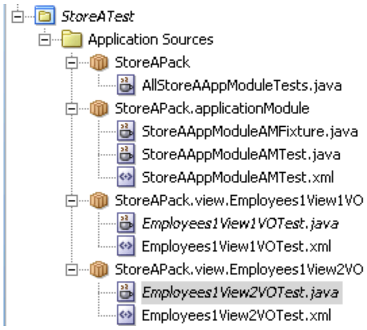
The JUnit BC4J Test Suite wizard will generate tests for each view object in the application module. If the application module does not have exported methods, the wizard will also generate a test for the application module itself. A generated view object class has the format *`view_object`*`VOTest.java` and is placed into a package with the format *`package`*`.view.`*`viewobject`*`VO`, where *`package`* is the application module package. A generated application module test has the format *`application_module`*`AMTest.java` and is placed into a package with the format *`package`*`.applicationModule`. A generated test fixture class has the format *`applicationmodule`*`AMFixture.java` and is placed in the same package as the application module test.

The generated all test suite class has the format `All`*`applicationmodule`*`Test.java` and is placed into the package with the same name as the application module package name.

A test case XML file is also generated for each application module or view object test. The XML file contains test methods defined in the application module or view object test cases. It does not include the test methods from the base classes (if any) because there may be too many duplicates.

For instance, after you created a test suite for an application module named `StoreAAppModule` with view objects `Employees1View1` and `Employees1View2` in the package StoreAPack, the Applications window displays the test hierarchy as shown in Figure 48-20.

Figure 48-60 Business Components Test Suite in the Applications Window


StoreATest
Application Sources
StoreAPack
AllStoreAAppModuleTests.java
StoreAPack.applicationModule
StoreAAppModuleAMFixture.java
StoreAAppModuleAMTest.java
StoreAAppModuleAMTest.xml
StoreAPack.view.Employees1View1VO
Employees1View1VOTest.java
Employees1View1VOTest.xml
StoreAPack.view.Employees1View2VO
Employees1View2VOTest.java
Employees1View2VOTest.xml

Before you begin:

It may be helpful to have an understanding of the using JUnit regression testing to create test cases. For more information, see Regression Testing with JUnit.

You will need to complete this task:

> Create the application modules in the data model project, as described in Implementing Business Services with Application Modules.

To create a business components test suite:

1. In the main menu, choose **File** and then **New** and then **From Gallery**.

   You will create a separate project for the business components tests.
2. In the New Gallery, expand **General**, select **Projects** and then **Java Application Project**, and click **OK**.
3. In the Project Name page of the Create Java Project wizard, enter a name and the directory path for the test project, and click **Next**.
4. In the Project Java Settings page, enter the package name, the directory of the Java source code in your project, and output directory where output class files will be placed, and click **Finish.**
5. In the Applications window, double-click the application module you want to test.
6. In the overview editor, click the **Java** navigation tab.
7. In the Java page of the overview editor, click the **Edit** icon for the **Java Class** section.
8. In the Select Java Options dialog, select **Generate Application Module Class** and click **OK**.
9. In the Java page of the overview editor, click the **Edit** icon for the **Class Interface** section.
10. In the Edit Client Interface dialog, shuttle the methods you want to test to the **Selected** pane, and click **OK**.
11. In the Applications window, right-click the test project you have created and choose **New** and then **From Gallery**.

12. In the New Gallery, expand **General**, select **Unit Tests** and then **Business Components Test Suite**, and click **OK**.
13. In the Configure Tests page of the JUnit BC4J Test Suite wizard, select values for the following and click **Next**:
    - **Business Component Project**: Select the project that has the application module you want to test.
    - **Application Module**: Select the application module you want to test.
    - **Configuration**: Choose a local or shared application module.
    - **Test Base Class-Application Module Extends**: You can specify different base cases. The generated test case classes will extend from that base class where all public abstract methods in the base class will have simple and default implementation method bodies.
    - **Test Base Class-View Object Extends**: You can specify which class the view object extends. The generated test case classes will extend from that base class where all public abstract methods in the base class will have simple and default implementation method bodies.
14. In the Summary page, verify the selections and click **Finish**.

How to a Create Business Components Test Fixture

When you create a business components test suite, a business components test fixture is created with it. You can also create Business Components test fixtures independently.

A generated test fixture class has the format `*applicationmodule*AMFixture.java` and put into a package with the format `*package*.applicationModule`, where *package* is the application module package.

Before you begin:

It may be helpful to have an understanding of the using JUnit regression testing to create test cases. For more information, see Regression Testing with JUnit.

You will need to complete this task:

> Create the application modules in the data model project, as described in Implementing Business Services with Application Modules.

To create a business components test fixture:

1. In the main menu, choose **File** and then **New** and then **From Gallery**.

   You will create a separate project for the business components tests.
2. In the New Gallery, expand **General**, select **Projects** and then **Java Application Project**, and click **OK**.
3. In the Project Name page of the Create Java Project dialog, enter a name and the directory path for the test project, and click **Next**.
4. In the Project Java Settings page, enter the package name and the source and output directories, and click **Finish.**
5. In the Applications window, double-click the application module you want to test.

6. In the overview editor, click the **Java** navigation tab and then click the **Edit** icon for the **Java Class** section.
7. In the Select Java Options dialog, select **Generate Application Module Class**, and click **OK**.
8. In the Java page of the overview editor, click the **Edit** icon for the **Class Interface** section.
9. In the Edit Client Interface dialog, shuttle the methods you want to test to the **Selected** pane, and click **OK**.
10. In the Applications window, right-click the test project you have created and choose **New** and then **From Gallery**.
11. In the New Gallery, expand **General**, select **Unit Tests** and then **Business Components Test Fixture**, and click **OK**.
12. In the Configure Tests page of the JUnit BC4J Test Fixture wizard, select values for the following and click **Next**:
    - **Business Component Project**: Select the project that has the application module you want to test.
    - **Application Module**: Select the application module you want to test.
    - **Configuration**: Choose a local or shared application module.
13. In the Summary page, verify the test fixture class and click **Finish**.

How to Run a JUnit Test Suite as Part of an Ant Build Script

Apache Ant is a popular, cross-platform build utility for which JDeveloper offers design time support. You can incorporate the automatic execution of JUnit tests and test output report generation by using Ant's built-in `junit` and `junitreport` tasks. The following example shows a task called `tests` from an Ant `build.xml` file. It depends on the `build` and `buildTests` targets that Ant ensures have been executed before running the `tests` target.

```
<target name="testCustomizations" depends="compileExtensionClasses">
  <junit printsummary="yes" haltonfailure="yes">
    <classpath refid="customization.classpath">
      <pathelement location="${customization.build.dir}"/>
    </classpath>
    <formatter type="plain"/>
    <test name="oracle.customization.tests.AllTests"/>
  </junit>
</target>
```

The `junit` tag contains a nested `test` tag that identifies the test suite class to execute and specifies a directory in which to report the results. The `junitreport` tag allows you to format the test results into a collection of HTML pages that resemble the format of Javadoc.

To try running the JUnit test from Ant, select the `build.xml` file in the Applications window, and choose **Run Ant Target >** ***tests*** from the context menu.

49
Refactoring a Fusion Web Application

This chapter describes considerations for renaming, moving, and deleting source files, configuration files, objects, attributes, and elements in a Fusion web application. In most cases, JDeveloper can perform the complete refactoring. However, in some cases, you might need to complete some manual steps to refactor.
This chapter includes the following sections:

About Refactoring a Fusion Web Application

JDeveloper offers extensive support for refactoring Fusion web application components, available through the Refactor main menu selections or via context menus for selected ADF components. The refactoring of ADF application components in most cases includes renaming, moving, and deleting these components.

JDeveloper provides refactoring options that allow you to make changes to the name and location of attributes, named elements, files, and **ADF Business Components** objects that your application uses. These refactoring options synchronize your changes with other parts of the application that are dependent on the changes. For example, renaming an ADF Business Components object such as a **view object** using the **Rename** option renames any references to it in other XML source files.

In a Fusion Web Application, you can use the refactoring actions (move, rename, and delete) in JDeveloper to make changes to the names and locations of objects. These actions are available from the main menu under **Refactor**, as well as the context menu for each type of object in the Applications window. The move, rename, and delete operations are available for most (but not all) objects and file types. When an object or file is selected in the Applications window, the applicable refactoring operations are enabled in the Refactor menu.

Additionally, there are limitations to what you can refactor in JDeveloper. In some cases, such as moving a project configuration file, there are manual procedures that you can follow. For more information, see Refactoring Limitations .

Refactoring Use Cases and Examples

During the course of development, you will create objects such as entity objects, as needed, to satisfy the needs of the application. Then you might find that you need to subsequently need to delete an object that is no longer used, rename an object to fit a naming convention, or move a set of objects into a different package to consolidate their location. Using JDeveloper, you can refactor these objects so that they are updated and the references to these objects in other objects are also updated to maintain the integrity of the whole application.

Additional Functionality for Refactoring

You may find it helpful to understand other refactoring features available from JDeveloper. Following are links to other functionality that may be of interest.

- For additional information about using JDeveloper features for refactoring Java code, see the Refactoring Java Projects section of *Developing Applications with Oracle JDeveloper*.

Renaming Files

You can display the Refactor menu when you right-click on the ADF Business Component in the Applications window. If you choose to rename a component, JDeveloper will present a Rename dialog. In this dialog you can specify the new name for the object and choose to preview the changes that will be generated by the refactoring process.

You can rename files, such as configuration files, using the following methods:

- In the Applications window, select the file and choose **File** > **Rename** from the main menu.
- In the Applications window, right-click the file and choose **Refactor > Rename**.
- In the source editor, select a class name, right-click it, and choose **Rename**.

To move a file between directories, use the **Move** menu item.

When you rename or move a file, all references to the file are updated as well. For example when you rename a page definition file, its entry in the `DataBindings.cpx` file is updated accordingly.

You cannot rename the XML file or Java files associated with an ADF Business Components object independently of the object, as these files must be kept in synch. See Refactoring ADF Business Components .

Moving JSF Pages

You can change the package of the JSF page with the Refactor menu. This updates the faces-config.xml file, DataBindings.cpx mappings to the page, and the ADF task flows containing page views.

In addition to the other refactoring operations, you can change the package of a JSF page. In the Applications window, right-click a JSF page and choose **Refactor** > **Move** to move the page to another package. Moving the JSF page to another package updates:

- `faces-config.xml` files that reference the page and its package
- ADF task flows containing views associated with the page
- `DataBindings.cpx` mappings to the page

Refactoring pagedef.xml Bindings Objects

The pagedef.xml file defines bindings and executables that populate the data in UI components at runtime. You can choose any data binding or executable and rename or delete it using the Refactor menu.

The `pagedef.xml` binding objects that you can refactor include bindings and executables. See Working with Page Definition Files.

Before you begin:

It may be helpful to have an understanding of the options you have for refactoring. See About Refactoring a Fusion Web Application .

In JDeveloper, open the application that contains the objects you want to refactor.

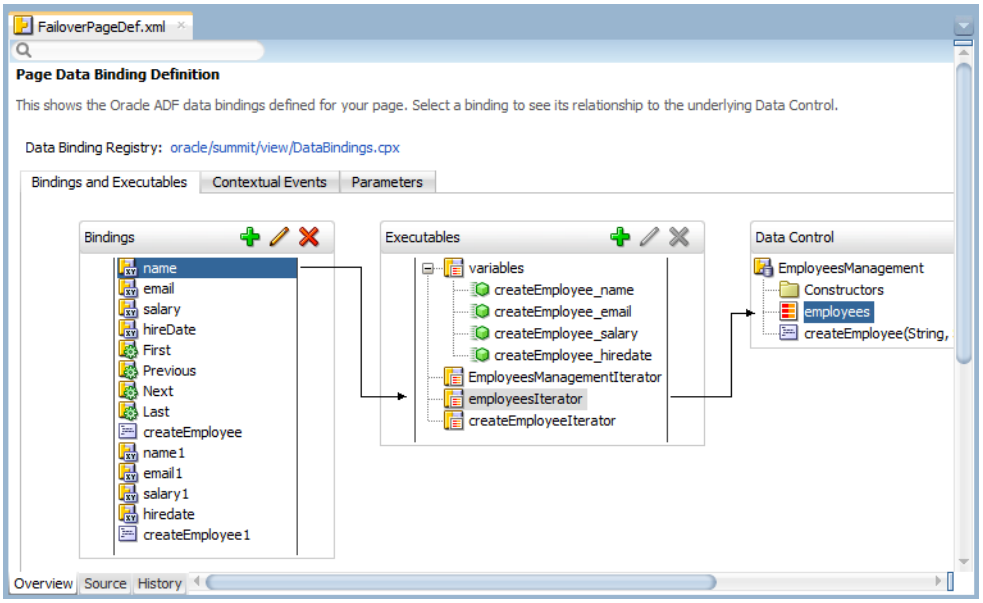
To refactor pagedef.xml binding objects:

1. In the Applications window, select the page node on which you have added a bound object such as an ADF Form or selection list.
2. Right-click the page node and choose **Go to Page Definition**.

   If the page does not already have a page definition, the Create Page Definition dialog appears. Click **OK** to create a page definition for the page.
3. In the overview editor, click the **Bindings and Executables** tab.

   Data bindings such as list bindings and iterator bindings defined for the page display under **Bindings and Executables**, as shown in Figure 49-1.

Figure 49-1 Page Data Binding Definition Overview Tab



Page Data Binding Definition

This shows the Oracle ADF data bindings defined for your page. Select a binding to see its relationship to the underlying Data Control.

Data Binding Registry: oracle/summit/view/DataBindings.cpx

4. In the Structure window, right-click a data binding or executable, choose **Refactor** and a refactoring option, such as **Rename** or **Delete**.

5. To display the usages between bindings, executables, and data controls, right-click a binding or executable and choose **Find Usages**.

Refactoring ADF Business Components

When it comes to refactoring ADF components in JDeveloper, the common refactoring features that are used are rename, move or delete ADF Business Components objects.

ADF Business Components includes objects such as view objects and entity objects. Table 49-1 shows support for refactoring ADF Business Components.

Table 49-1 Refactoring ADF Business Components

Action	Result
Move	Moves the object to a different package or directory and updates all references.
Delete	JDeveloper shows all dependencies on the object and permits a forced delete. The application may not work at this point. You may need to resolve broken references.
Rename	ADF Business Components objects are defined by an XML file. The XML file has a file name identical to the object name. For example, the name of the XML file for a view object named `Persons1View` is `Persons1View.xml`. Renaming results in changing the `Name` attribute, renaming the XML file, and updating all references. For example, the name of an entity (`Customer`) is stored as an attribute in the XML file (`name=Customer`). The XML file has the same name the entity name (`Customer.xml`).

Table 49-1 (Cont.) Refactoring ADF Business Components

Action	Result
Find Usages	JDeveloper shows all dependencies on the object.

> **Note:**
>
> Refactoring does not cross abstraction layers. For example, when a view object is created based on the `Dept` entity object, it is named `DeptView` by default. Renaming the `Dept` entity object updates the entity usage in `DeptView`, but does not change the name of the view object.

Before you begin:

It may be helpful to have an understanding of the options you have for refactoring. For more information, see About Refactoring a Fusion Web Application .

In JDeveloper, open the application that contains the object you want to refactor.

To refactor ADF Business Components objects:

1. In the Applications window, expand the project and then the **Application Sources** node and then the package containing the object you want to refactor.
2. Right-click the object and choose **Refactor** and a refactoring option.
   - To change the name of the object, choose **Rename**.

     The Rename dialog displays the current name of the object. Enter the new object name and click **OK** to proceed with the refactoring or **Cancel** to cancel.

     You can optionally select **Preview** and click **OK** to display the usages in the Log window, which allows you to proceed with the refactoring or cancel. When you click a usage in the Log window, the file that contains the usage is opened in the Source editor and the containing line is highlighted.
   - To change the location of the object, choose **Move**.

     The Move dialog displays the current location of the object. Enter the new package name, or click **Browse** to navigate to it. Click **OK** to proceed with the refactoring or **Cancel** to cancel.
   - To remove the object, choose **Delete**.

     The Confirm Delete dialog displays the name of the object and searches for usages. Click **Show Usages** to display where the object is used. Click **Yes** to proceed with the refactoring or **No** to cancel. Click **Preview** to display the results in the Log window, which allows you to proceed with the refactoring or cancel. When you click a usage in the Log window, the file that contains the usage is opened in the Source editor and the containing line is highlighted.

     By default JDeveloper searches in the current application for usages of the object. You can change the scope of the search by choosing from the **Finding Usages In** dropdown list, or by clicking the **Manage Working Sets** icon.

Refactoring ADF Business Component Object Attributes

You can refactor attributes of ADF Business Components entity objects and view objects.

Table 49-1 shows support for refactoring attributes of ADF Business Component entity objects and view objects.

Table 49-2 Refactoring Attributes

Action	Result
Move	Not supported.
Delete	JDeveloper shows all dependencies on the attribute and permits a forced delete. The application may not work at this point. You may need to resolve broken references.
Rename	Attributes share data elements represented in entity and view objects (see About Entity Objects for more information). References to the attribute are updated when you rename the attribute. Renaming results in changing the `Name` attribute and updating all references. This includes updating the service implementation of view attributes used in the generated service of an **application module**. Renaming an attribute does not change the data it represents, nor does it rename the underlying table column.
Find Usages	JDeveloper shows all dependencies on the attribute.

Before you begin:

It may be helpful to have an understanding of the options you have for refactoring. See About Refactoring a Fusion Web Application .

In JDeveloper, open the application that contains the attribute you want to refactor.

To refactor attributes:

1. In the Applications window, double-click the entity object or view object that contains the attribute you want to refactor.
2. In the overview editor, click the **Attributes** navigation tab.
3. In the **Name** column, right-click the attribute and choose a refactoring option.
   - To change the name of the attribute, choose **Rename**.

     The Rename dialog displays the current name of the attribute. Enter the new attribute name and click **OK** to proceed with the refactoring or **Cancel** to cancel.

     You can optionally select **Preview** and click **OK** to display the usages in the Log window, which allows you to proceed with the refactoring or cancel.
   - To see where the attribute is used, choose **Find Usages**.

     The Log window displays usages of the attribute. When you click a usage in the Log window, the file that contains the usage is opened in the Source editor and the containing line is highlighted.
   - To remove the attribute, choose **Delete**.

In the Confirm Delete dialog, click **Show Usages** to display where the attribute is used. Click **Yes** to proceed with the refactoring or **No** to cancel. Click **Preview** to display the results in the Log window, which allows you to proceed with the refactoring or cancel.

- To change the type of the attribute, choose **Change Type**.

  In the Change Type dialog, choose the new type from the **Type** dropdown list, or click **Browse** to find the type. Click **OK** to proceed with the refactoring or **Cancel** to cancel.

  You can optionally select **Preview** and click **OK** to display the usages in the Log window, which allows you to proceed with the refactoring or cancel.

Refactoring Named Elements

A named element is not an object or attribute. Named elements are elements in the XML schema that can be referenced by a Name attribute. JDeveloper allows you to refactor a named element.

Named elements are any elements in the XML schema that can be referenced by a `Name` attribute. A named element is not an object or an attribute. Table 49-3 shows support for refactoring named elements in an XML schema.

Table 49-3 Refactoring Named Elements

Action	Result
Move	Not supported.
Delete	Not supported.
Rename	One exception to the definition of named elements is the design time element `Attr`, which does have a `Name` attribute. `Attr` is a name-value pair, is not accessible from the code editor, and should not be renamed.
Find Usages	JDeveloper shows all dependencies on the named element.

Before you begin:

It may be helpful to have an understanding of the options you have for refactoring. See About Refactoring a Fusion Web Application.

In JDeveloper, open the application that contains the named element you want to refactor.

To refactor named elements:

1. In the Applications window, double-click the entity object or view object that contains the named element you want to refactor.
2. In the editor window, click the **Source** tab, and search or scroll to find the named element.

   Named elements are indicated by `Name="<element>"` in the source code, for example:

   ```
   <Key Name="PersonsAffContactChk">
   ```

A named element is not an object or attribute.

3. Right-click the named element you want to refactor and choose **Refactor** and a refactoring option.

   - To change the name of the element, choose **Rename**.

     The Rename dialog displays the current name of the element. Enter the new element name and click **OK** to proceed with the refactoring or **Cancel** to cancel.

     You can optionally select **Preview** and click **OK** to display the usages in the Log window, which allows you to proceed with the refactoring or cancel.

   - To remove the element, choose **Delete**.

     In the Confirm Delete dialog, click **Show Usages** to display where the element is used. Click **Yes** to proceed with the refactoring or **No** to cancel. Click **Preview** to display the results in the Log window, which allows you to proceed with the refactoring or cancel.

Refactoring ADF Task Flows

You can refactor existing activities, JSF page flows, and JSF pages into new ADF Controller components such as bounded task flows and task flow templates using JDeveloper.

For more information, see Refactoring to Create New Task Flows and Task Flow Templates.

Refactoring the DataBindings.cpx File

In order to rename or move the DataBindings.cpx file, you need to create the new one and delete the existing one. While doing this you must not forget to update the adfm.xml file which is the registry of registries and the id property must be similar to the DataBindings.cpx file name otherwise you will get an error message.

The `DataBindings.cpx` file defines the Oracle ADF **binding context** for the entire application and provides the metadata from which the Oracle ADF binding objects are created at runtime (see DataBindings.cpx Syntax for more information). This file is a registry used to quickly find all `.cpx`, `.dcx`, `.jpx`, and `.xcfg` files, which are themselves registries of metadata.

If you rename the `DataBindings.cpx` file to a new name, such as `DataBindingsNew.cpx`, the change is added to the `adfm.xml` file.

The following example shows the contents of the `adfm.xml` file after `DataBindings.cpx` is refactored to `DataBindingsNew.cpx`.

```
<?xml version="1.0" encoding="UTF-8" ?>
<MetadataDirectory xmlns="http://xmlns.oracle.com/adfm/metainf"
version="11.1.1.0.0">
<DataBindingRegistry path="adf/sample/view/DataBindingsNew.cpx"/>
</MetadataDirectory>
```

Additionally, to enable the application to access the correct bindings file, the ID value is changed to an ID similar to the one shown in the following example.

```
<Application xmlns="http://xmlns.oracle.com/adfm/application"
version="11.1.1.49.28" id="DataBindingsNew" SeparateXMLFiles="false"
Package="adf.sample.view" ClientType="Generic">
```

Refactoring Limitations

JDeveloper has set of refactoring issues and limitations that are listed here.

Table 49-4 summarizes the limitations of JDeveloper's refactoring support.

Table 49-4 Refactoring Limitations

Area	Limitation
Database	When a database artifact used in an ADF Business Components object is renamed, the object needs to be updated. This type of refactoring is currently not supported.
Service interface	A service interface defines a contract between two separate pieces of software. For example, the ADF Business Components service interface is responsible for exposing business components to the view and model layers. Changing the name of a service interface can cause a conflict. While developing the application, consider removing the service interface, refactoring the object, and regenerating the service interface. Additionally, if your service interface defines a find operation based on a **view criteria** that specifies bind variables, changing the number or order of the bind variables in the underlying view criteria will require that you regenerate the service interface. See Creating SOAP Web Services with Application Modules.
Java literal references	The Java code generated by ADF Business Components has literal references to the XML metadata. These literal references are updated during refactoring operations. Generated (type safe) methods are also updated. Also, the refactor delete operation is available for local Java variables. However, if the application code directly refers to the metadata, these references are not updated.
Domain	If a domain needs to be renamed or moved, you must create the new domain, then change the type of existing domain usages.For example, you might rename a domain called `EmployeeID` to `EmployeeNumber`. In addition, the entity `Emp` has an attribute called `Empno` that is of type `EmployeeID`.After creating the new domain `EmployeeNumber`, go to the attributes page for the entity, right-click **Empno** and choose **Change Type**. This switches `Empno` from `EmployeeID` to `EmployeeNumber`.
Security	Security policies in the policy store may reference the name of an entity object, attribute, page, or task flow. These policy definitions are not updated in response to the refactoring of the object itself.

Table 49-4 (Cont.) Refactoring Limitations

Area	Limitation
Resource Bundles	Entity object definitions can reference a resource in one or more arbitrary resource bundle (`.properties`) files that you create. You can use this file to define labels for the attributes of entity objects. However, if you rename the `.properties` file you created, JDeveloper will not update the entity object definitions to reflect the new file name. As an alternative to resource bundle files that you create, you can specify a project setting to generate a single, default resource bundle file for the data model project. In this case, JDeveloper will not allow you to rename this generated file. However, if you attempt to change the project-level default resource bundle file, JDeveloper will warn you about the change. The data model project will honor the new ADF Business Components project-level setting for any objects that have not yet been linked to the default resource bundle file; all existing Business Components that have already been linked to the original default file will continue to use it instead.
`.jpx` project configuration file	Renaming the ADF Business Components project configuration file (`.jpx`) is not supported. In previous versions of JDeveloper, ADF Business Components project configuration `.jpx` files were created only in the root package of the `src` directory of a project and were named with the same base name as the project. The ADF Business Components objects (entity objects, view objects, and application modules) were all created in the model package (for example, `/model/AppModule.xml`), but the `/Model.jpx` is not.This may cause a reusability problem when attempting to package them in ADF JAR files for use on the class path. There may be name conflicts because several projects are named `Model`.

Moving the ADF Business Components Project Configuration File (.jpx)

If the ADF Business Components Project file (.jpx) is moved to a different package, then the jbo.project property available in bc4j.xcfg file must be changed accordingly. However, it is not always changed automatically.

Although refactoring the ADF Business Components project configuration file (`.jpx`) is not supported, you may need to change its name or location to avoid conflicts when sharing your project contents as an ADF library. The `.jpx` file contains configuration information that JDeveloper uses in the design time to allow you to create the data model project with ADF Business Components. If you need to refactor this file, you must do so manually.

Before you begin:

It may be helpful to have an understanding of the options you have for refactoring. See About Refactoring a Fusion Web Application.

To manually move the ADF Business Components project configuration file (.jpx):

1. Move the `.jpx` file to the new source tree location.

For example, you can move the `Model.jpx` file from `src/Model.jpx` to `src/newpackage/name/here/Model.jpx`.

2. Change the `.jpx` file contents. The `PackageName` attribute of the root element `JboProject` needs to have the correct value.

   For example, you can specify `PackageName="newpackage.name.here"`.

3. Change the `jbo.project` attributes in all `common/bc4j.xcfg` files that contain elements referred to in the `.jpx` file to include the new package name.

   For example:

   ```
   <AppModuleConfig name="ScottDeptAMLocal"
   ApplicationName="newpackage.name.here.ScottDeptAM"
   DeployPlatform="LOCAL" JDBCName="scottdb"
   jbo.project="newpackage.name.here.Model">
   ```

4. Change the contents of the JDeveloper project file (the `.jpr` file):

   - Set the new `.jpx` package location. If you later change the default package in the project properties, you will again raise a `NotFound` error for the `.jpx` file.

     For example:

     ```
     <value n="defaultPackage" v="newpackage.name.here"/>
     ```

   - Fix any `ownerURL` elements in the `ownerMap` that contain references to the old location of the `.jpx` file.

     For example:

     ```
     <url n="ownerURL" path="src/newpackage/name/here/Model.jpx"/>
     ```

50 Reusing Application Components

This chapter describes how to package certain Oracle ADF components into the ADF Library for reuse in Fusion web applications. Reusable ADF components are application modules, business components (entity objects, view objects, associations), data controls, task flows, page templates, and declarative components.
This chapter includes the following sections:

About Reusable Components

In the course of ADF application development, certain components will often be reused. Creating and consuming reusable components should be included in the early design and architectural phases of the application development. It is advantageous to package these reusable components into a library that can be shared between different teams. These shared libraries can be added to a repository. Later if you need a component, you may look into the repository for something to be reused.

In the course of application development, certain components will often be used more than once. Whether the reuse happens within the same application, or across different applications, it is often advantageous to package these reusable components into a library that can be shared between different developers, across different teams, and even across departments within an organization.

In the world of Java object-oriented programming, reusing classes and objects is just standard procedure. With the introduction of the model-view-controller (MVC) architecture, applications can be further modularized into separate model, view, and controller layers. By separating the data (model and **business services layer**) from the presentation (view and controller layers), you ensure that changes to any one layer do not affect the integrity of the other layers. You can change business logic without having to change the UI, or redesign the web pages or front end without having to recode domain logic.

Oracle ADF and JDeveloper support the MVC design pattern. When you create an application in JDeveloper, you can choose many application templates that automatically set up data model and user interface projects. Because the different MVC layers are decoupled from each other, development can proceed on different projects in parallel and with a certain amount of independence.

ADF Library further extends this modularity of design by providing a convenient and practical way to create, deploy, and reuse high-level components. When you first design your application, you design it with component reusability in mind. If you created components that can be reused, you can package them into JAR files and add them to a reusable component repository. If you need a component, you may look into the repository for those components and then add them into your project or application.

For example, you can create an **application module** for a domain and package it to be used as the data model project in several different applications. Or, if your application will be consuming components, you may be able to load a page template component from a repository of ADF Library JARs to create common look and feel pages. Then you can put your page flow together by stringing together several task flow components pulled from the library.

An ADF Library JAR contains ADF components and does not, and cannot, contain other JARs. It should not be confused with the JDeveloper library, Java EE library, or Oracle WebLogic shared library.

Table 50-1 lists the reusable components supported by ADF.

Table 50-1 Oracle ADF Reusable Components

Reusable Component	Description
Data control	Any data control can be packaged into an ADF Library JAR. Some of the data controls supported by Oracle ADF include application modules, Enterprise JavaBeans, web services, URL services, JavaBeans, and placeholder data controls.
Application module	When you are using **ADF Business Components** and you generate an application module, an associated **application module data control** is also generated. When you package an application module data control, you also package up the ADF Business Components associated with that application module. The relevant entity objects, **view objects**, and associations will be a part of the ADF Library JAR and available for reuse.
Business components	Business components are the entity objects, view objects, and associations used in the ADF Business Components data model project. You can package business components by themselves or together with an application module.
Task flows and taskflow templates	Task flows can be packaged into an ADF Library JAR for reuse. If you drop a bounded task flow that uses page fragments, JDeveloper adds a region to the page and binds it to the dropped task flow. ADF bounded task flows built using pages can be dropped onto pages. The drop will create a link to call the bounded task flow. A task flow call activity and control flow will automatically be added to the task flow, with the view activity referencing the page. If there is more than one existing task flow with a view activity referencing the page, it will prompt you to select the one to automatically add a task flow call activity and control flow. If an ADF task flow template was created in the same project as the task flow, the ADF task flow template will be included in the ADF Library JAR and will be reusable.
Page templates	You can package a page template and its artifacts into an ADF Library JAR. If the template uses image files and they are included in a directory within your project, these files will also be available for the template during reuse.
Declarative components	You can create declarative components and package them for reuse. The tag libraries associated with the component will be included and loaded into the consuming project.

You can also package up projects that have several different reusable components if you expect that more than one component will be consumed. For example, you can create a project that has both an application module and a bounded task flow. When this ADF Library JAR file is consumed, the application will have both the application module and the task flow available for use. You can package multiple components into one JAR file, or you can package a single component into a JAR file. Oracle ADF and JDeveloper give you the option and flexibility to create reusable components that best suit you and your organization.

You create a reusable component by using JDeveloper to package and deploy the project that contains the components into a ADF Library JAR file. You use the components by adding that JAR to the consuming project. At design time, the JAR is added to the consuming project's class path and so is available for reuse. At runtime, the reused component runs from the JAR file by reference. For the procedure to add the JAR manually, see How to Add an ADF Library JAR into a Project Manually. For the procedure to add the JAR using the JDeveloper Resource Catalog, see How to Add an ADF Library JAR into a Project using the Resources Window.

Before you proceed to create reusable components, you should review the guidelines for creating reusable components.

Creating Reusable Components

Creating and consuming reusable components should be included in the early design and architectural phases of software projects. You and your development team should consider which components are candidates for reuse, not only in the current applications but also for future applications and including those applications being developed in other departments.

You and your team should decide on the type of repository needed to store the library JARs, where to store them, and how to access them. You should consider how to organize and group the library JARs in a structure that fits your organizational needs. You should also consider creating standardized naming conventions so that both creators and consumers of ADF Library JARs can readily identify the component functionality.

> **Tip:**
>
> If, in the midst of development, you and your team find a module that would be a good candidate for reuse, you can use the extensive refactoring capabilities of JDeveloper to help eliminate possible naming conflicts and adhere to reusable component naming conventions.

Naming Conventions

When you create reusable components, you should try to create unique and relevant names for the application, project, application module, task flow, connection, or any other file or component. Do not accept the JDeveloper wizard default names such as Application, Project, ViewController, AppModule, `task-flow-defintion.xml`, or Connection. You want to try to have unique names to avoid naming conflicts with other projects, components, or connections in the application. Naming conflicts could arise from components created in the consuming application and those loaded from other JAR files. Table 50-2 lists the objects that you may be required to rename.

Table 50-2 Example Unique and Relevant Names for Reusable Components

Type	JDeveloper Default	Example
Application	Application	`SummitADF`
Project	Model ViewController Project	`BackOfficService` `OrderTrackingUI`
Package	Various possibilities. For more information, see Naming Considerations for Packages .	`oracle.summit`
Application module	AppModule	`SummitAppModuleDataControl`
Connection	Connection1	`summit_adf`
Task flow	`task-flow-defintion.xml`	`customer-task-flow-definition.xml`
Page template	`templateDef`.jspx	`SummitTemplate.jspx`
Declarative Component	`componentDef` `componentDef.jspx`	`SummitsuperwidgetDef` `SummitsuperwidgetDef.jspx`
ADF Library JAR file	adflib<string or 3-digit random number>*N* For more information, see The Naming Process for the ADF Library JAR Deployment Profile.	`SummitADF_Model_adflibSummitADF1`

Naming Considerations for Packages

Be aware that some components use the default package name of the project without allowing the name to be explicitly set. In this situation, you must take extra care to avoid package name collisions. You can set the package name in the application creation wizard and you should check the names in the Project Properties dialog afterwards. If you don't set the package name, it will default to a variant of the project name, typically with the first letter being lowercase. For example, a project with the name `Project1` will have a default package name of `project1`. You should manually change the package name to a more unique name before you proceed to build the project.

> **Note:**
>
> The basic package naming requirement is that ADF metadata registries (`.dcx`,`.cpx`, and so on) are generated based on the project's package name, and you should avoid metadata naming conflicts between projects that will be combined at runtime.

When you are creating a reusable component's web resource files (such as JSPs, HTMLs, and task flows), you should create them in their own relative directories. When the JAR is deployed into another application, there will be less chance for conflict between the reusable component's files and the consuming application's files.

> **Note:**
>
> You can override the staleness period setting of 364 days for all static web application resources in ADF Libraries by adding initialization parameters to the web.xml file. For more information, see web.xml.

Naming Considerations for Connections

Often, several modules in an application will connect to the same data source. You should standardize the connection name to the same data source to avoid confusion because there is only one namespace for connections across the application. This would require coordination with other developers, component producers, and component consumers. For example, if customers and suppliers both have a connection to the same database, the connection name should be standardized to an agreed upon name, such as orders_db.

ADF Library JARs (with connections) may be used in different applications with different connection requirements. ADF Library JAR producers should choose connection names that are at least representative of the connection source, if not the actual standardized connection name. Be aware that consumers of the JAR that was created with connections will be required to satisfy the connection requirements when they add the component to the application.

For example, for a database connection, choosing an endpoint host name is usually not appropriate. The most appropriate name is a complete representative for the schema. Acceptable example names for connections are oracle-appsdb and oracle-customersdb. You should realize that if many reusable components use different names for the same logical connection, then the consumer of the component will have to satisfy each one individually with duplicate information. The consumer will have to supply connection details for several different connection names, when in fact they all refer to the same instance.

Naming Considerations for Applications with EJB Projects

If an application has both EJB projects and a web application project with data binding, you should check to see that the EJB component names are not in conflict with any other web application project component names. The EJB project components may have global scope because the project is automatically added to the global class path of all web projects in the application. The web application project may mistakenly access a component with the same name in the EJB project rather than within its own project.

For example, if both an EJB project and a web-based project have a test1.jspx page, when the web-based project is run, it may try to run the EJB project test1.jspx page.

At runtime, JDeveloper detects if there are EJB projects and web application projects with data binding in the same application. If there are both types in the application, when the project is run and the server starts up, a warning message will appear in the Log window.

The Naming Process for the ADF Library JAR Deployment Profile

Before you package the project, you must create a deployment profile with the name and path for the JAR file. You should choose a name that follows you and your development team's naming convention and that is descriptive of the function of the component. You should realize that the consuming project may also include other JAR files from other software authors.

For example, if the component is a task flow for self-service paying, you might name it `mycompany.hcm.pay.selfservice.taskflows.jar`. Other examples are `oracle.apps.hcm.pay.model.jar` and `mycompany.hcm.pay.model.overrides.jar`.

When you create the deployment profile, JDeveloper will present a default name with the following format:

`adflib`*`identifierN`*

Where *`identifier`* is random number up to three-digits and *`N`* is a number that starts with 1 and increments by 1 for each iteration of the profile.

For example, the default name for a profile may be `adflib7491(`random number is `749` and suffix is `1)`. The default name for the another profile may be `adflib51` (random number is `5` and suffix is `1)`.

You can change this default name to a name of your choice.

Keeping the Relevant Project

When you are creating reusable components, you should eliminate any projects that are not relevant to the reusable component. For example, if you want to create a reusable application module, you would need a data model project, but you would not need a user interface project. In this instance, if you had created an application with both a data model and a user interface project, you could delete the user interface project. Of course, you should rename the default name `Model` to something more relevant, such as `BackOfficeService`. Similarly, if you are creating a reusable task flow that is not databound, you can delete any data model project from the application.

Selecting the Relevant Feature

If you know the technology scope of your consuming projects, you can design your component with technologies that will be compatible. For example, if the consuming application uses only standard JSF Faces, then it may not be compatible with a declarative component that is built with ADF Faces.

When you create your application, you can define the features using the Create Application Wizard by selecting from the application template.

After the project has been created, you can define the technology using the Features page of the Project Properties dialog.

Selecting Paths and Folders

If you are using the file system to store your ADF Library JARs, you should select file system locations that can function as repositories. You may want to put groups of JARs into a common directory folder, for example, `C:\ADF\jdev\DevTeamADFRepository`. If your team or organization plans to share ADF Library JARs, you should consider setting up network-accessible repository folders, directories, or services. The Resources window has provisions to connect to different repository sources and make multiple connections. For more information about accessing ADF Library JARs using the Resources window, see Using the Resources Window.

Including Connections Within Reusable Components

If the project you are packaging into an ADF Library JAR includes a connection, that information can be included in the JAR and may be available to the consuming project. Oracle ADF uses connection architecture which defines a connection as two parts, the connection name and the connection details (or endpoint definition). JDeveloper will present the producer of the JAR the option to package the connection name only or to include connection details with the connection name.

If a connection is present in the project, the packaged ADF Library JAR will contain a `jar-connections.xml` file and a `jar-adf-config.xml` file. They will be added to the `META-INF` directory. The `jar-connections.xml` file contains the connection name and other relevant connection information. The `jar-adf-config.xml` file stores the information about the credentials used for the connections. If connection credentials were also specified, then a `jar-credential-jazn-data.xml` will also be included for the credential store. You can select the individual connections or connection types that you want to be packaged with the ADF Library JAR when you create the ADF Library JAR deployment profile. You make connection selections in the Create or Edit ADF Library Deployment Profile Properties dialog Connections page, as described in How to Package a Component into an ADF Library JAR.

When an ADF Library JAR is being added to a project, JDeveloper checks for conflicts between the application's connections and the JAR's connections. A dialog will be presented to allow you to decide whether to proceed with adding the JAR to the project. Connections defined in the ADF Library JAR may be added to the consuming project, depending on the following conditions:

- If the connection defined in the JAR is fully configured and there are no connection name conflicts with the consuming project, a new connection will be added.
- If the connection defined in the JAR is partially configured, a new connection will be added to the consuming project but it must be configured before use. Connection dialogs may appear to allow the user to enter connection information. This partially configured connection may be indicated by an incomplete icon.
- If the connection defined in the JAR has the same name as the application's connection, it will not be added to the project. The application's existing connection will be used.
- If the connection defined in the JAR has the same name as the application's connection but is of different type, the JAR's connection will not be added to the project. For example, if a database connection in the JAR and a URL connection in the application have the same name, the database connection will not be added to the application.

For instructions on how to add an ADF Library to a project that includes connections, see How to Add an ADF Library JAR into a Project using the Resources Window.

> **Note:**
>
> In the process of adding the ADF Library to a project, you will see one message displayed in the Log window for each connection defined in the JAR. The messages indicate the merge action taking place in the project.

Reusable ADF Components Use Cases and Examples

Packaging projects into ADF Library JARs allows components to be reused multiple times in the same or in different applications. Different development teams can create and consume projects and create libraries of reusable modules. ADF Library JARs support the M-V-C design pattern. You can create ADF Libraries and access them using drag and drop from the Resources window.

For instance, you can package up an application module into an ADF Library JAR and place it in a public folder for other team to use. Different implementation teams may access that JAR to create their applications. They may use the application module to create different pages for their use instead on having to create their own application modules. When there is a change in the model, the original developer can repackage the application module and republish the ADF Library JAR. The other implementation teams can update their application using the updated ADF Library.

Additional Functionality for Reusable ADF Components

You may find it helpful to understand other ADF features before you work with reusable ADF components. Following are links to other functionality that may be of interest.

- You can package business components for reuse. For more information on application modules, see Getting Started with ADF Business Components.
- You can package application modules for reuse. For more information on application modules, see Implementing Business Services with Application Modules.
- You can package task flows for reuse. For more information on task flows, see Getting Started with ADF Task Flows .
- You can package page templates for reuse. For more information on task flows, see Getting Started with Your Web Interface .
- You can package page templates for reuse. For more information, see the Using Declarative Components section of the *Developing Web User Interfaces with Oracle ADF Faces*.

Common Functionality of Reusable ADF Components

When you package or consume an ADF Library JAR, you also include and consume the extension libraries that are associated with the project. When you consume an ADF Library, you can use the Resources window.

Using Extension Libraries

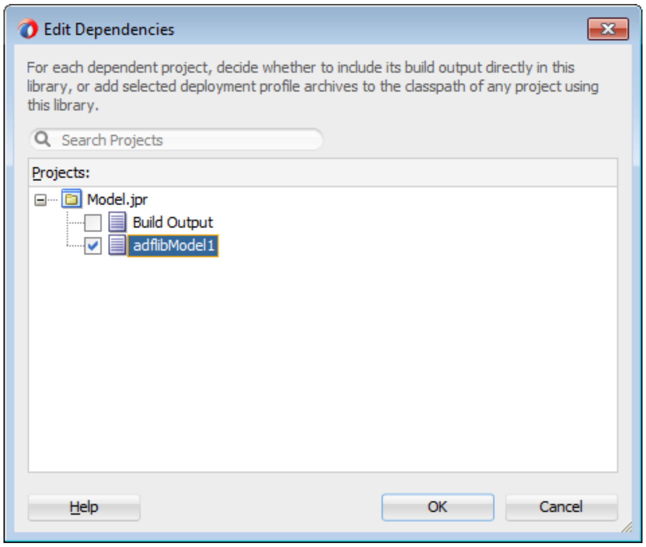
An ADF project usually includes a list of extension libraries that it needs to run. These libraries are loaded in the class path of the project. You can view a project's dependent libraries by selecting the **Libraries and Classpath** node of the Project Properties dialog. Some of the libraries that may appear are JSP Runtime, ADF Page Flow Runtime, Connection Manager, and Oracle JDBC.

When a project is packaged into an ADF Library JAR, its extension libraries are packaged with it. And when an ADF Library JAR is being consumed by another project, JDeveloper automatically resolves any extension library conflicts between them. During the consuming process, JDeveloper checks to see whether the consuming project already has the extension libraries of the ADF Library JAR in its class path and loads only those libraries that it does not have. For example, if JSP Runtime already exists in the consuming project, it will not be loaded again if the ADF Library JAR also includes it. The consuming project's extension libraries will be a union of its own libraries and the libraries in the ADF Library JAR.

If the project you want to package into an ADF Library has a dependent project, you can include the dependent project's extension libraries directly in the JAR or, if the dependent project has a deployment profile, you can add the dependent project's JAR to the ADF Library. For more information about setting up the deployment process, see How to Package a Component into an ADF Library JAR.

For example, suppose that project `View` is being packaged into ADF Library `adflibView1.jar` and that has a dependency on the `Model` project. For project `View`, the `Model` project is a dependent project with the deployment profile option (`adflibmodel1`) selected, as shown in Figure 50-1.

Figure 50-1 Edit Dependencies Dialog with Deployment Profile Option


Edit Dependencies

For each dependent project, decide whether to include its build output directly in this library, or add selected deployment profile archives to the classpath of any project using this library.

Search Projects

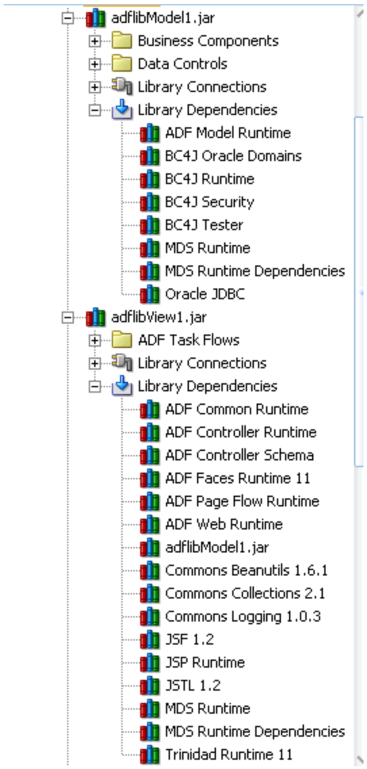
Projects:

- Model.jpr
  - □ Build Output
  - ☑ adflibModel1

When the deployment profile is selected, the dependent project's JAR file will be added to the ADF Library JAR. As a result, the extension libraries of the dependent project will also be made available to any consuming project. In Figure 50-2, the ADF

Library being packaged, `adflibView1.jar`, includes the dependent `adflibModel.jar` as listed under the Library Dependencies node.

Figure 50-2 Resources Window Showing adflibView1.jar and adflibModel.jar Extension Libraries

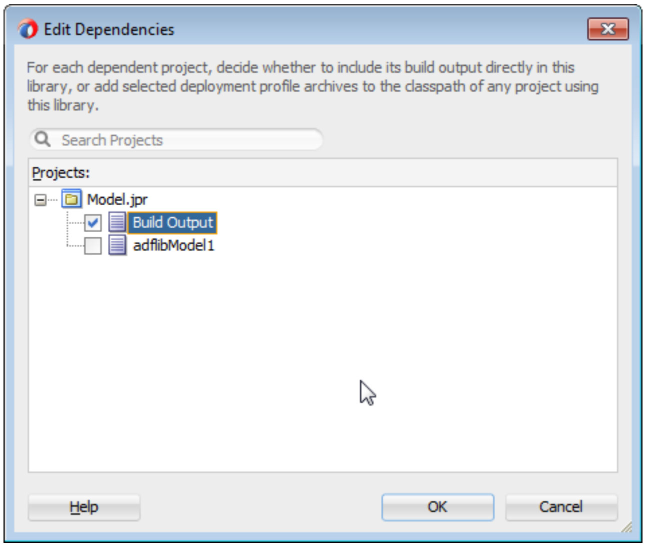


- adflibModel1.jar
  - Business Components
  - Data Controls
  - Library Connections
  - Library Dependencies
    - ADF Model Runtime
    - BC4J Oracle Domains
    - BC4J Runtime
    - BC4J Security
    - BC4J Tester
    - MDS Runtime
    - MDS Runtime Dependencies
    - Oracle JDBC
- adflibView1.jar
  - ADF Task Flows
  - Library Connections
  - Library Dependencies
    - ADF Common Runtime
    - ADF Controller Runtime
    - ADF Controller Schema
    - ADF Faces Runtime 11
    - ADF Page Flow Runtime
    - ADF Web Runtime
    - adflibModel1.jar
    - Commons Beanutils 1.6.1
    - Commons Collections 2.1
    - Commons Logging 1.0.3
    - JSF 1.2
    - JSP Runtime
    - JSTL 1.2
    - MDS Runtime
    - MDS Runtime Dependencies
    - Trinidad Runtime 11

Alternately, you can include the dependent project's artifacts and extension libraries directly into the ADF Library JAR.

For example, project `View` can also be packaged into ADF Library `adflibView2.jar`. It also has a dependency on the `Model` project. But in this second deployment profile, the `Model` project is a dependent project with the **Build Output** option selected (as opposed to the deployment profile (`adflibmodel1`) being selected), as shown in Figure 50-3.

Figure 50-3 Edit Dependencies Dialog Used to Built adflibview2.jar

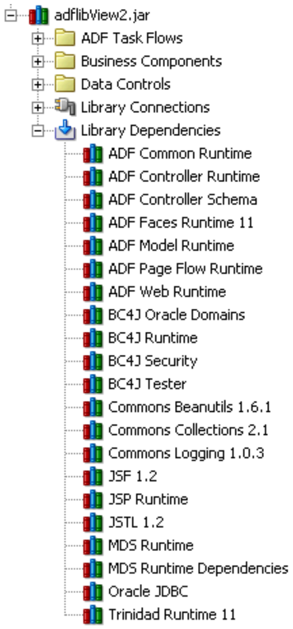


Edit Dependencies

For each dependent project, decide whether to include its build output directly in this library, or add selected deployment profile archives to the classpath of any project using this library.

When **Build Output** is selected, the dependent project's classes and extension libraries will be added directly to the ADF Library JAR. Figure 50-4 shows the ADF Library `adflibView2.jar`, which includes artifacts of the `Model` project and its extension libraries. Note that the extension libraries under the `adflibView2.jar` Library Dependencies node are the same as the combined extension libraries under the `adflibView1.jar` and `adflibModel.jar` shown in Figure 50-2.

Figure 50-4 Resources Window Showing adflibView2.jar Extension Libraries



How you decide to package the dependent project depends on how you intend the ADF Library to be used. Including the dependent project as a JAR may be advantageous when the dependent project is itself a reusable component, or when it is a dependent project for several projects that will be packaged into an ADF Library JAR. In the example, the dependent project `Model` may be a dependent project for several view projects. On the other hand, packaging the dependent project as **Build Output** is straightforward and eliminates the need for multiple JARs.

> **Note:**
>
> If you are creating an ADF Library JAR that is included in a JDeveloper extension library, you should include the additional manifest `JDevLibrary:` *`extension_library_name`* entry in the JAR. When you use **Add to Project** from the Resources window to add a ADF Library JAR that has a `JDevLibrary` manifest entry, the JDeveloper extension library containing this JAR will be added instead of the ADF Library JAR itself.

Using the Resources Window

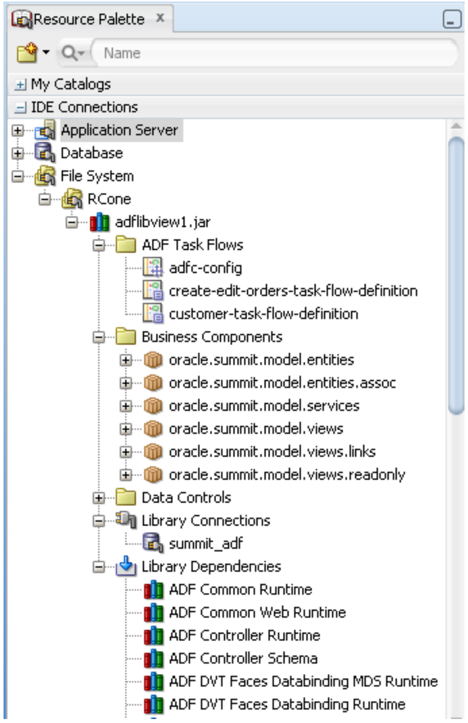
ADF Library JARs can be packaged, deployed, discovered, and consumed like any other Oracle Library component. Creating an ADF Library JAR is the action of packaging all the artifacts (and additional control files) of a project into a JAR. Consuming a reusable component from an ADF Library JAR is the action of loading that ADF Library JAR into the project's set of libraries.

However, the easiest way to manage and use ADF Library JAR components is by using JDeveloper's Resources window. With the Resources window, developers who want to consume reusable components can easily find and discover available components and add them to their projects. The Resources window provides search and browse functions across different data management systems to locate the component. It provides multiple connections to access different sources. It has a structure tree view for displaying different connections and the ADF Library JAR component types. Figure 50-5 shows the Resources window with several file system connections.

The Resources window tree structure displays each JAR as subcategories. Separate nodes are created for each type of reusable component. For example, application modules are under the Data Controls node, and task flows are under the ADF Task Flows node.

The tree structure for the ADF Library JAR lists any connection information under a Library Connections node and lists all the producing project's extension libraries under the Library Dependencies node.

Figure 50-5 Resources Window Showing ADF Library Structure



- My Catalogs
- IDE Connections
  - Application Server
  - Database
  - File System
    - RCone
      - adflibview1.jar
        - ADF Task Flows
          - adfc-config
          - create-edit-orders-task-flow-definition
          - customer-task-flow-definition
        - Business Components
          - oracle.summit.model.entities
          - oracle.summit.model.entities.assoc
          - oracle.summit.model.services
          - oracle.summit.model.views
          - oracle.summit.model.views.links
          - oracle.summit.model.views.readonly
        - Data Controls
        - Library Connections
          - summit_adf
        - Library Dependencies
          - ADF Common Runtime
          - ADF Common Web Runtime
          - ADF Controller Runtime
          - ADF Controller Schema
          - ADF DVT Faces Databinding MDS Runtime
          - ADF DVT Faces Databinding Runtime

Packaging a Reusable ADF Component into an ADF Library

These reusable components can be packaged into the ADF Library for reuse in applications. An ADF Library JAR is like another JAR file, but it contains ADF related components and project dependent libraries. The purpose of the ADF Library JAR file is for reuse.

Once you have decided that a certain component or components can be reused, create an application and a project to develop that component. Follow the guidelines in Creating Reusable Components to name your application, project, package, and other objects and files. A project corresponds to one ADF Library JAR. If you create multiple projects and want to reuse components from each of the projects, you may need to create an ADF Library JAR for each project. In other situations, you may be able to involve multiple components under one project to create a single ADF Library JAR. For example, you may be able to create application modules, business components (entity object, view objects, associations), task flows, and page templates all under one project and create one ADF Library JAR.

Creating an ADF Library JAR involves compiling the project and validating the components, creating a resource service file, control files, an `adflibREADME.txt`, and adding the relevant project files into a JAR. For more information about the ADF Library JAR, see What Happens When You Package a Project to an ADF Library JAR.

If you are packaging a component that itself uses another ADF Library component, the final consuming project must have both ADF Library JARs added to the project. For example, say you created a reusable task flow that contains tables dropped from a data control in another ADF Library JAR. When you add the task flow from an ADF

Library JAR into a consuming project, that project will also require the data control ADF Library JAR.

If you are packaging a component that has dependent JARs, such as third-party JARs, you have two options:

- If the consuming environment has control over the placement of JARs, you can place the ADF Library JAR and its dependent JARs in a JDeveloper Extension Library. The advantage of using a JDeveloper Extension Library is that it does not clutter the consuming project with specific references to the dependent JARs.
- If using JDeveloper Extension Library is not possible, you can place the dependent JARs in the same location as the ADF Library JAR and include a manifest classpath entry for each dependent JAR.

How to Package a Component into an ADF Library JAR

To package up a reusable component, you first create a deployment profile that specifies the archive type, the name of the JAR file, and the directory path where the JAR will be created. Then you deploy the project using the deployment profile.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you package an ADF component. For more information, see Packaging a Reusable ADF Component into an ADF Library.

You may also find it helpful to understand functionality that can be used with packaging an ADF component. For more information, see Additional Functionality for Reusable ADF Components.

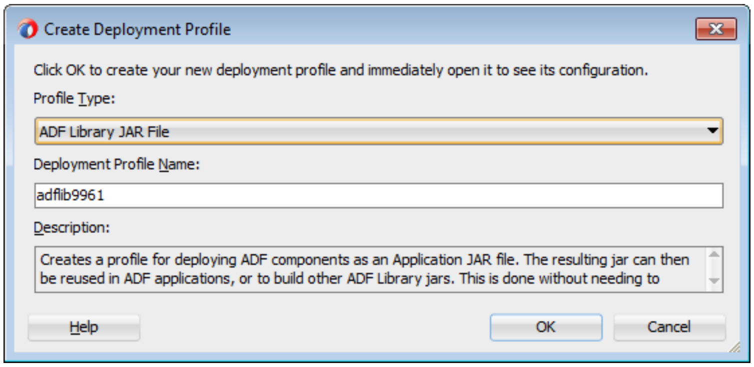
You will need to complete this task:

> Create a project and determine whether it has dependent projects and JAR that need to be packaged. For more information, see Packaging a Reusable ADF Component into an ADF Library.

To package and deploy a project into the ADF Library JAR:

1. In the Applications window, double-click the project that contains the component you want to make reusable.
2. In the Project Properties dialog, select **Deployment** and then click **New**.
3. In the Create Deployment Profile dialog, select **ADF Library JAR file** for **Profile Type**, enter a name or accept the default name for **Deployment Profile Name**, and click **OK**.

Figure 50-6   Create Deployment Profile Dialog with Default Name



Create Deployment Profile

Click OK to create your new deployment profile and immediately open it to see its configuration.

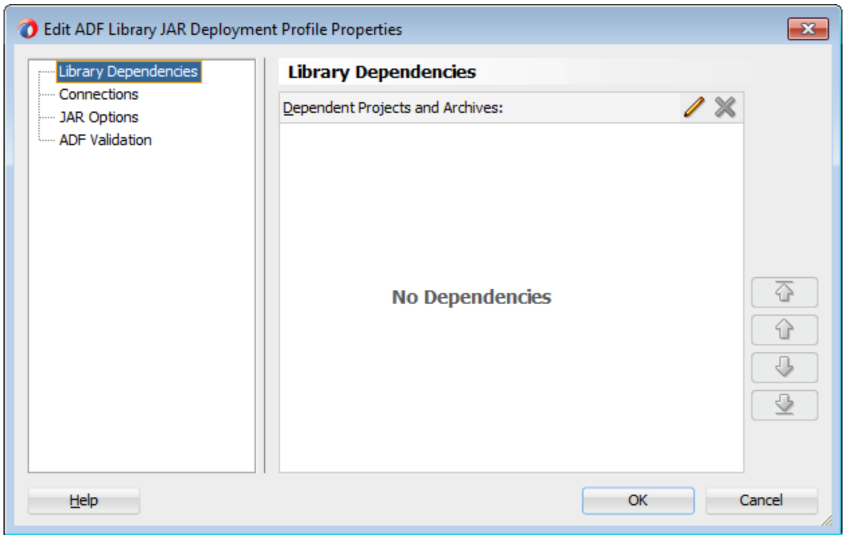
Profile Type: ADF Library JAR File

Deployment Profile Name: adflib9961

Description: Creates a profile for deploying ADF components as an Application JAR file. The resulting jar can then be reused in ADF applications, or to build other ADF Library jars. This is done without needing to

4. In the Project Properties dialog, select the deployment profile and click **Edit**.
5. In the Edit ADF Library JAR Deployment Profile Properties dialog, select the **Library Dependencies** node, as shown in Figure 50-7.

Figure 50-7   ADF Library JAR Deployment Profile Properties Dialog



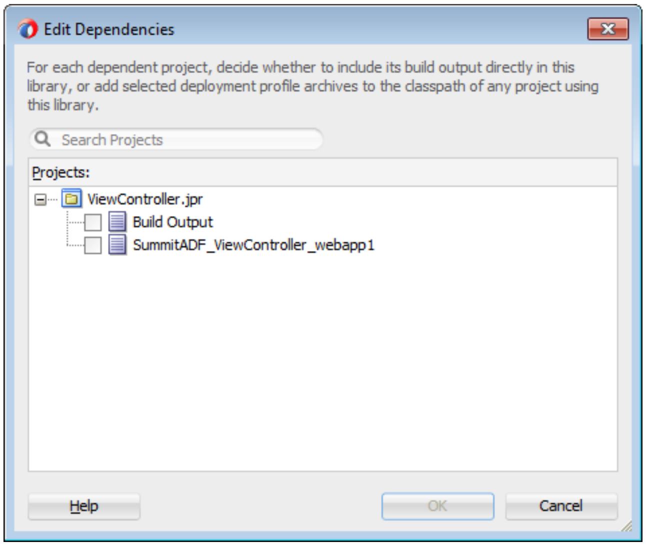
The Library Dependencies pane shows a list of dependent projects for the project being packaged. You can add the dependent project's build output directly into the packaging project's library, or you can add selected deployment profile archives to the class path.

a. To add dependent projects, click the **Edit** icon to bring up the Edit Dependencies dialog, as shown in Figure 50-8.

   If you select **Build Output**, the dependent project's extension libraries will be added directly to the ADF Library JAR. If you select deployment profile, the dependent project's JAR file (which includes its own extension libraries) will be added to the ADF Library JAR. For more information about library dependencies, see Using Extension Libraries.

   In this example, the `Model` project can be set as a dependency only as **Build Output**. However, the `ViewController` project can be set as a dependency either as **Build Output**, or as a deployment profile (`SummitADF_ViewController_webapp`).

Figure 50-8 ADF Library Deployment Edit Dependencies Dialog



Edit Dependencies

For each dependent project, decide whether to include its build output directly in this library, or add selected deployment profile archives to the classpath of any project using this library.

b. For each dependent project, select the **Build Output** node for the project or select the dependent profile and click **OK**.

6. In the Edit ADF Library JAR Deployment Profile Properties dialog, select the **Connections** node, as shown in Figure 50-9.

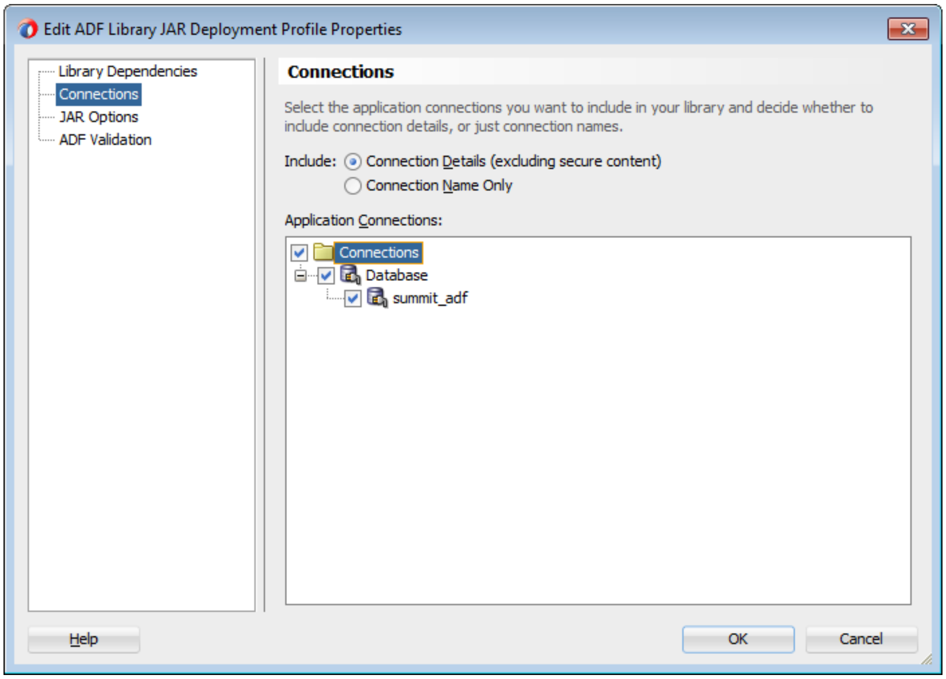
   You can select:

   - **Connection Details (excluding secure content)**: If the project has a connection, select this checkbox if you want to include any available connection details in addition to the connection name. Connection details include hostname, port number, sid, driver type, and user name. For more information, see Including Connections Within Reusable Components.
   - **Connection Name Only**: Select this checkbox if you want to add the connection name without any connection details.

   > **Note:**
   >
   > The connection options available depends on the JDeveloper role. **Connection Details (excluding secure content)** and **Connection Name Only** are the selections available for the Studio Developer role. If your JDeveloper is set to a different role, you may have different options and defaults.

   In the Applications Connections tree structure, select the checkbox for the level of connection you want to include.

Figure 50-9 Connections Page of the Edit ADF Library JAR Deployment Profile Properties Dialog



Connections

Select the application connections you want to include in your library and decide whether to include connection details, or just connection names.

Include: ● Connection Details (excluding secure content)
○ Connection Name Only

Application Connections:

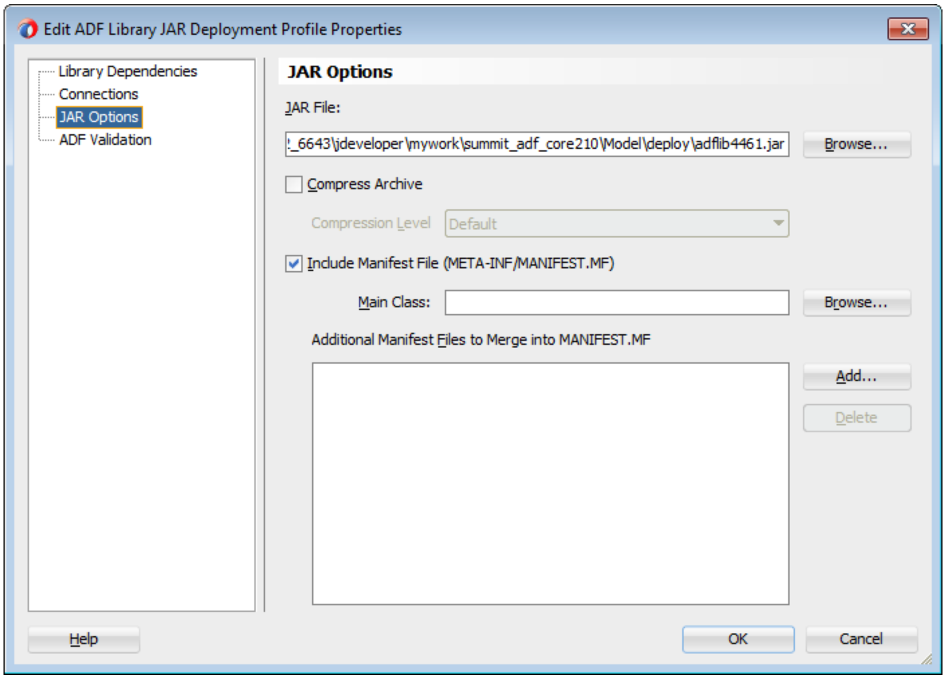
☑ Connections
☑ Database
☑ summit_adf

7. Select the **JAR Options** node, as shown in Figure 50-10, verify the default directory path or enter a new path to store your ADF Library JAR file.

   If the ADF Library JAR is to be included in a JDeveloper extension library, create a text file with a `JDevLibrary:` *`extension_library_name`* entry and place the file in the project root directory. Click **Add** to locate and merge that file into the `Manifest.mf` file.

   When you create your manifest text file, make sure the `JDevLibrary` entry starts in column 1 with a space after the colon, and that there is a blank line at the end of the file.

Figure 50-10 ADF Library JAR Deployment Profile Properties Dialog JAR Option



JAR File: !_6643\jdeveloper\mywork\summit_adf_core210\Model\deploy\adflib4461.jar

□ Compress Archive

Compression Level: Default

☑ Include Manifest File (META-INF/MANIFEST.MF)

Main Class:

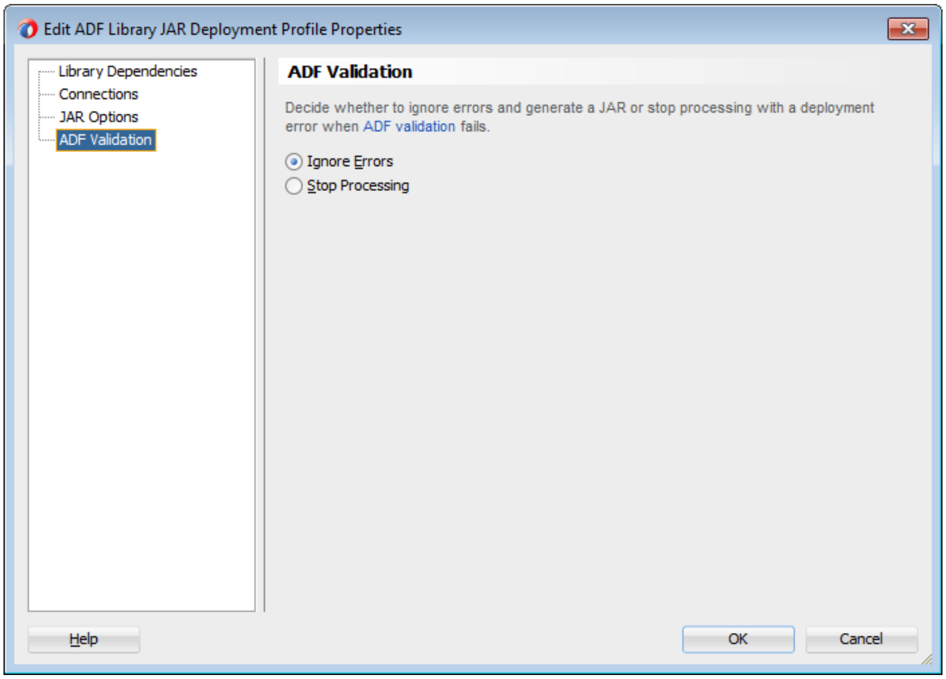
Additional Manifest Files to Merge into MANIFEST.MF

8. Select the **ADF Validation** node.

   You can select:

   - **Ignore Errors**: To create the JAR file even when validation fails. This is the default option.
   - **Stop Processing**: To stop processing when validation fails.

Figure 50-11 ADF Library JAR Deployment Profile Properties Dialog ADF Validation



Decide whether to ignore errors and generate a JAR or stop processing with a deployment error when ADF validation fails.

● Ignore Errors
○ Stop Processing

9. Click **OK** to finish setting up the deployment profile.
10. In the Applications window, right-click the project and choose **Deploy >** ***deployment***, where ***deployment*** is the name of the deployment profile.
11. In the Deploy dialog Deployment Action page, click **Next** and then click **Finish**.

    JDeveloper will create the ADF Library JAR in the directory specified in Step 7. You can check that directory to see whether the JAR was created.

What Happens When You Package a Project to an ADF Library JAR

When you deploy the library JAR, JDeveloper packages up all the necessary artifacts, adds the appropriate control files, generates the JAR file, and places it in the directory specified in the deployment profile. During deployment, you will see compilation messages in the Log window.

When you deploy a project into an ADF Library JAR, JDeveloper performs the following actions:

- Package the HTML root directory artifacts into the JAR. When the JAR is added to the consuming project, JDeveloper will make the reusable component's `public_html` resources available by adding it to the class path.
- Add the `adfm.xml` file to the JAR. If there are multiple `META-INF/adfm.xml` files in the workspace, only the `adfm.xml` in the project being deployed is added. JDeveloper modifies this file to include relevant content from any dependent project's `adfm.xml` file.
- Add a service resources file, `oracle.adf.common.services.ResourceService.sva`, into the `META-INF` directory of the JAR. The addition of this file differentiates an ADF Library JAR file from

standard JAR files. This file defines the service strategies of the JAR and allows the Resources window to properly discover and display the contents of the JAR.

- Add a `Manifest.mf` file to the JAR. The `Manifest.mf` file is used to specify dependencies between JAR files, and to determine whether to copy and include the contents of a JAR file or to reference it. JDeveloper will create a default manifest file. For example:

  ```
  Manifest-Version: 1.0
  ```

- Adds a `jar-connections.xml` file to the JAR for components that require a connection and that use the connections architecture. Note that in the consuming application, connection information in configuration files that are defined in the class path and accessible at runtime may be merged together. If the same connection is named multiple times in the class path, the connection in the main application will be given priority.

Different types of reusable components have different artifact files and different entries in the service resource file.

Application Modules Files Packaged into a JAR

For application modules, JDeveloper adds these control files to the JAR: `oracle.adf.common.services.ResourceService.sva`, `Manifest.mf, adfm.xml`, and the business components `.jpx` file. The service resource file for an application module includes entries for the business components associated with the application module, as well as an entry for the application module data control.

The `jar-connections.xml` file may appear for components that use the connection architecture and that contain connection information regarding the data source.

Data Controls Files Packaged into a JAR

For data controls such as placeholder data controls, JDeveloper includes three control files in the JAR: `oracle.adf.common.services.ResourceService.sva`, `Manifest.mf`, and `DataControl.dcx` file. Data controls are used when the data source is not based on ADF Business Components, and so business components are not included in the JAR file, as is the case in an application module JAR file. The service resource file for a standard data control has an entry for the data control.

The JAR also includes the `Datacontrol.dcx` file from the project to describe the data control type.

Task Flows Files Packaged into a JAR

For task flows, JDeveloper includes three control files in the JAR: `oracle.adf.common.services.ResourceService.sva`, `Manifest.mf`, and `task-flow-registry.xml`. The service resource file for a task flow includes an entry that indicates that one or more task flows are in the JAR.

Page Templates Files Packaged into a JAR

For page templates, JDeveloper includes two control files in the JAR: `oracle.adf.common.services.ResourceService.sva` and `Manifest.mf`.

Declarative Components Files Packaged into a JAR

For declarative components, JDeveloper includes two control files in the JAR: `oracle.adf.common.services.ResourceService.sva` and `Manifest.mf`.

How to Place and Access JDeveloper JAR Files

If you have JAR files that can be reused by other projects, such as third-part JAR files, you can use JDeveloper to place them in an accessible location and create a library file (`.library`) to designate them. The consumer can use JDeveloper to navigate to this location and add the JAR to the project.

An ADF Library JAR contains ADF components and does not and cannot contain other JARs.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you package an ADF component. For more information, see Packaging a Reusable ADF Component into an ADF Library.

You may also find it helpful to understand functionality that can be used with packaging an ADF component. For more information, see Additional Functionality for Reusable ADF Components.

You will need to complete this task:

> Place the JAR files in a accessible location for both the producer and the consumer of these JAR files. For example, the directory may be on a network drive where other shared files are located. For more information, see Packaging a Reusable ADF Component into an ADF Library.

To place and access a JDeveloper LIbrary JAR:

1. From the main menu, choose **Tools > Manage Libraries.**
2. In the Manage Libraries dialog, click **Load Dir**.
3. In the Load Directory dialog, select the directory where the secondary JAR files are located.
4. In the Manage Libraries dialog, click **New**.
5. In the Create Library dialog, enter a library name and click **Add Entry**.

   You have created a library file (with a `.library` extension). You should place library files in source control systems.
6. In the Select Path Entry dialog, select the JARs you want to add and click **OK**.
7. In the Create Library dialog, be sure that the **Deployed by Default** option is set correctly for your JAR, click **OK** and then click **OK** again.
8. In the consuming project.
   a. From the main menu, choose **Tools > Manage Libraries.**
   b. In the Manage Libraries dialog, click **Load Dir**.
   c. In the Load Directory dialog, select the directory where the secondary JAR files are located and click **Select**. This should be the same location specified in Step 3.

d. Right-click the project and select **Project Properties**.

e. In the Project Properties window, select **Libraries and Classpath** and click **Add Library**.

f. In the Add Library dialog, select the library, click **OK** and then click **OK** again.

Adding ADF Library Components into Projects

You can add an ADF Library JAR into your project manually or by using the Resources window. The project will access and consume the components in the JAR as the ADF Library JAR file will be added to the class path of the project.

After ADF Library JARs are created, they must be distributed to the developers who will use these JARs. Distributing the ADF Library JARs may include putting the JARs into a network file system to be searched, browsed, and discovered. It may include using other forms of data store or services to access and retrieve these JARs. Since ADF Library JARs are simply binary files, they can distributed like any other file such as ftp and email.

Once you have access to the ADF Library JARs, you can use JDeveloper to access them and add them to your consuming projects. Using the JDeveloper Resources window is the easiest and most efficient way. You can also use JDeveloper to manually add the JARs into the project by entering them into the class path.

When a project is packaged into an ADF Library JAR, it captures the list of dependent JARs the project needs for deployment and runtime. This list is based on the project's dependent profiles and the information in the project's Libraries and Classpath with **Deployed By Default** selected. When the ADF Library JAR is added to the consuming project, this list of dependent JARs is placed in a library called *ADF Library Dependencies*. This is a locked library in the consuming project and its content will not be shown in the Data Controls panel, Components window, or other places in JDeveloper. The library is maintained and updated as different ADF Library JARs are added and deleted from the project. If the dependencies have changed (for example, the producer project was rebuilt), then you can refresh the dependencies in the consuming project using a JDeveloper menu command.

Once reusable components have been added, how they are used depends on the type of component.

How to Add an ADF Library JAR into a Project using the Resources Window

You can use the Resources window to search, discover, and add the ADF Library JAR to your project.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you package an ADF component. For more information, see Adding ADF Library Components into Projects.

You may also find it helpful to understand functionality that can be used with packaging an ADF component. For more information, see Additional Functionality for Reusable ADF Components.

You will need to complete this task:

Create an ADF Library JAR in a repository folder. If you do not already have a Resources window connection to this repository, you must know the location of this folder. For more information, see Adding ADF Library Components into Projects.

To add a component to the project using the Resources Window:

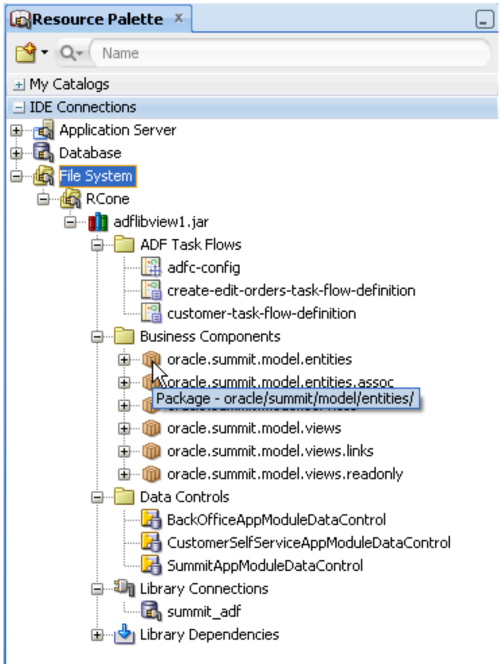
1. From the main menu, choose **Window > Resources**.
2. In the Resources window, click the **New** icon, and then choose **IDE Connections > File System**.
3. In the Create File System Connection dialog, enter a name and the path of the folder that contains the JAR.

   For file path guidelines, see Selecting Paths and Folders.
4. Click **OK**.

   The new ADF Library JAR appears under the connection name in the Resources window.
5. To examine each item in the JAR structure tree, use tooltips. The tooltip shows pertinent information such as the source of the selected item.

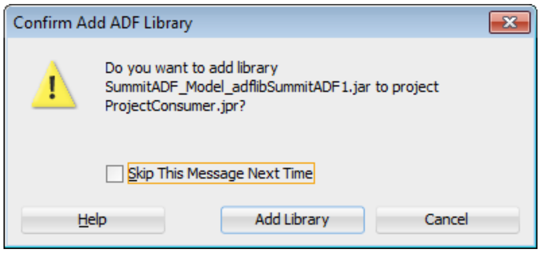
   Figure 50-12 shows a Resources window with a tooltip message that shows package information for a business component.

Figure 50-12 Tooltip Message for a Connection in the Resources Window


Resource Palette
Name
My Catalogs
IDE Connections
Application Server
Database
File System
RCone
adflibview1.jar
ADF Task Flows
adfc-config
create-edit-orders-task-flow-definition
customer-task-flow-definition
Business Components
oracle.summit.model.entities
oracle.summit.model.entities.assoc
Package - oracle/summit/model/entities/
oracle.summit.model.views
oracle.summit.model.views.links
oracle.summit.model.views.readonly
Data Controls
BackOfficeAppModuleDataControl
CustomerSelfServiceAppModuleDataControl
SummitAppModuleDataControl
Library Connections
summit_adf
Library Dependencies

6. To add the ADF Library JAR or one of its items to the project, right-click the item and choose **Add to Project**. In the confirmation dialog that appears, click **Add Library**, as shown in Figure 50-13.

Figure 50-13 Confirm Add ADF Library Dialog



Confirm Add ADF Library

Do you want to add library SummitADF_Model_adflibSummitADF1.jar to project ProjectConsumer.jpr?

If you had previously added that library JAR, you will get a Confirm Refresh ADF Library dialog asking you whether you want to refresh the library.

7. If the ADF Library JAR has changes to its dependencies, you can refresh the project with the new changes. From the Applications window, right-click the project and select **Refresh ADF Library Dependencies in *project*.jpr**.

For application modules and data controls, you have the option to drag and drop the application module or data control from the Resources window into the Data Controls panel.

> **Note:**
>
> JDeveloper will load whichever ADF Library JAR extension libraries are not already in the consuming project. Extension libraries from the ADF Library's dependent JARs will also be checked and loaded if not already part of the consuming project. For more information, see Using Extension Libraries

How to Add an ADF Library JAR into a Project Manually

You can add an ADF Library JAR in the same way as you would other library JARs.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you package an ADF component. For more information, see Adding ADF Library Components into Projects.

You may also find it helpful to understand functionality that can be used with packaging an ADF component. For more information, see Additional Functionality for Reusable ADF Components.

You will need to complete this task:

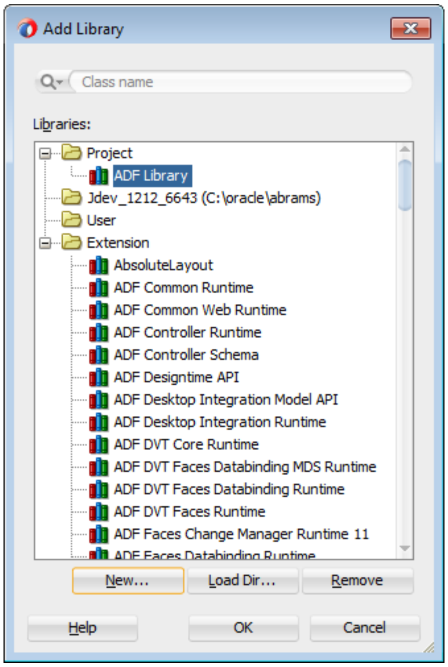
Create an ADF Library JAR. For more information, see Adding ADF Library Components into Projects.

To add a component to a project manually:

1. In the Applications window, double-click the project to which the component is to be added.
2. In the Project Properties dialog, select the **Libraries and Classpath** node and then click **Add Library**.

3. In the Add Library dialog, click **New**.
4. In the Create Library dialog, in the **Location** dropdown list, select **Project** and enter a name for the ADF Library. The preferable name is "ADF Library". Select **Deploy by Default**, and click **Add Entry**.
5. In the Select Path Entry dialog, enter or browse to the ADF Library JAR file and click **Open**.
6. The Create Library dialog reappears with the path of the JAR file filled in under the **Class Path** node. Click **OK**.
7. The Add Library dialog reappears with the ADF Library entry filled in under the **Project** node. Click **OK**.

Figure 50-14 Add Library Dialog



8. The Project Properties dialog reappears with the JAR file added to the list of libraries. Click **OK**.

What Happens When You Add an ADF Library JAR to a Project

When you add an ADF Library JAR to a project, either by using the Resources window or by manually adding the JAR, an ADF Library definition is created in the project. The ADF Library JAR file will be added to the class path of the project. The project will access and consume the components in the JAR by reference.

By default, the **ADF Library Deployed by Default** option in the Create Library dialog is set to true. If this option is set, when the application or module is further archived or built into a WAR file, the contents of the ADF Library JAR will be copied to that archive or WAR file. If the Deploy by Default option is not set, then the JARs in the ADF Library must be loaded in some other way, such as by deploying them in a shared library.

Figure 50-15 shows the empty Data Controls panel for a consuming project before the ADF Library was added.

Figure 50-15 Data Controls Panel of the Consuming Project

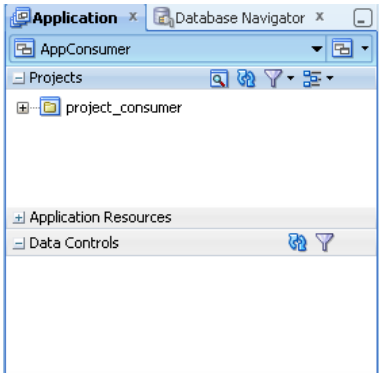


Figure 50-16 shows the `adflibview1` ADF Library being added to the consuming project.

Figure 50-16 Adding the ADF Library JAR to the Project

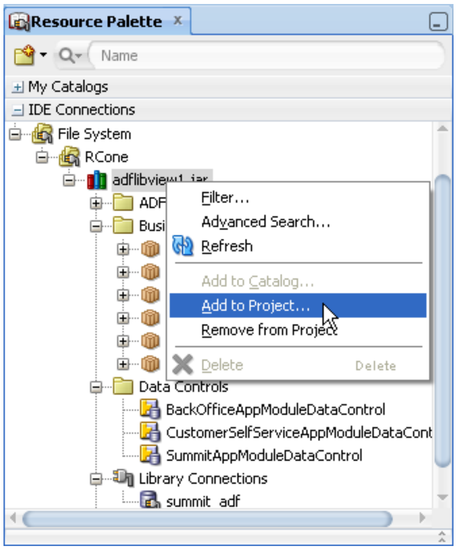
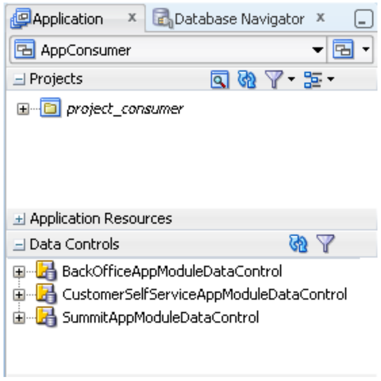


Figure 50-17 shows several data controls from the ADF Library added to the Data Controls panel in the consuming project.

Figure 50-17 Consuming Project Data Controls Panel with Added Application Modules



After adding the ADF Library JAR, you may notice some changes to some of the JDeveloper windows. These changes are different depending on the type of components being added. Table 50-3 lists the effects on several JDeveloper windows.

Table 50-3 JDeveloper Window After Adding an ADF Library

Added Component	Data Controls Panel	Creation Wizards	Components window
Data controls	Data control appears.		
Application module	Application module appears.		
Business components		Entity objects available in view object creation wizard. View objects in JAR also available for use.	
Task flows			Task flows appear in Components window.
Page template		Page template available during JSF creation wizard.	
Declarative components			Tag library appears in Components window. Declarative component appears in the list.

What You May Need to Know About Using ADF Library Components

Although the procedure to add an ADF Library JAR to a project is standardized, the component type determines where it appears in JDeveloper and how it can be reused.

Using Data Controls

When you add a data control to a project, the data control appears in the Data Controls panel. If you are using the Resources window, you have the option of dragging and dropping the data control from the Resources window onto the Data Controls panel, and then dragging and dropping from the Data Controls panel onto the page.

Using Application Modules

When you add an application module to a project, the application module appears in the Data Controls panel. If you are using the Resources window, you have the option of dragging and dropping the application module item from the Resources window onto the Data Controls panel, and then dragging and dropping from the Data Controls panel onto the page.

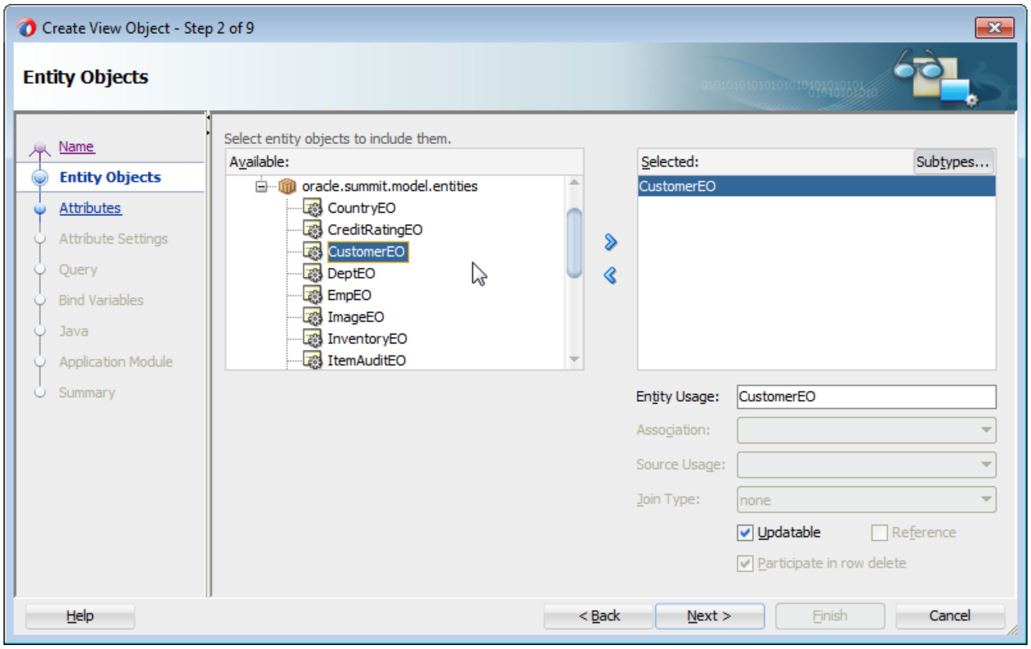
Application modules are associated with business components. When the reusable application module was packaged, the JAR includes the business components used to create the application module. These components will be available for reuse.

Using Business Components

Business components can also be packaged and reused. The entity objects, view objects, and associations can be packaged together and added to a consuming project. By default, packaged application modules will include the business components in the JAR, but business components can be reused by themselves without the accompanying application module.

One way to reuse business components is to create new view objects using the entity objects from an ADF Library JAR. When you add view objects using the wizard, the entity objects will become available within the wizard to support view object generation. For instructions on creating view objects, see How to Create an Entity-Based View Object. When the wizard presents a screen for entity objects used to create view objects, the entity objects from the ADF Library will be available in the shuttle window, as shown in Figure 50-18.

Figure 50-18 Creating View Object Using Entity Objects from ADF Library



Select entity objects to include them.

Selected: CustomerEO

Entity Usage: CustomerEO

Association:

Source Usage:

Join Type: none

☑ Updatable □ Reference

☑ Participate in row delete

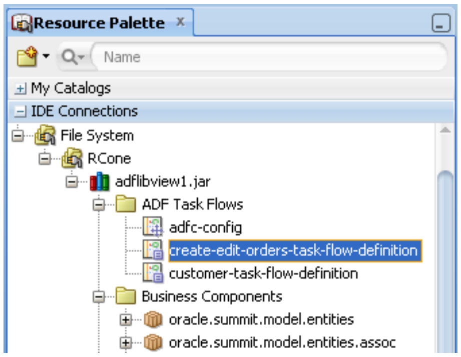
If the consuming project has the **Initialize Project for Business Component** option selected (in the Project Properties dialog Business Components page) before the ADF Library JAR is added, the business components within the JAR will automatically be made available to the project.

If you add the ADF Library JAR first, and then select **Initialize Project for Business Component**, JDeveloper will automatically load the business components.

Using Task Flows

Task flows added to a project will appear in the Components window when you are adding components to a JSF page. If you are using the Resources window, you can also drag and drop the task flow directly from the Resources window onto another task flow or page, as shown in Figure 50-19.

Figure 50-19 Using the Resources Window to Drag and Drop Task Flows



Resource Palette
Name
My Catalogs
IDE Connections
File System
RCone
adflibview1.jar
ADF Task Flows
adfc-config
create-edit-orders-task-flow-definition
customer-task-flow-definition
Business Components
oracle.summit.model.entities
oracle.summit.model.entities.assoc

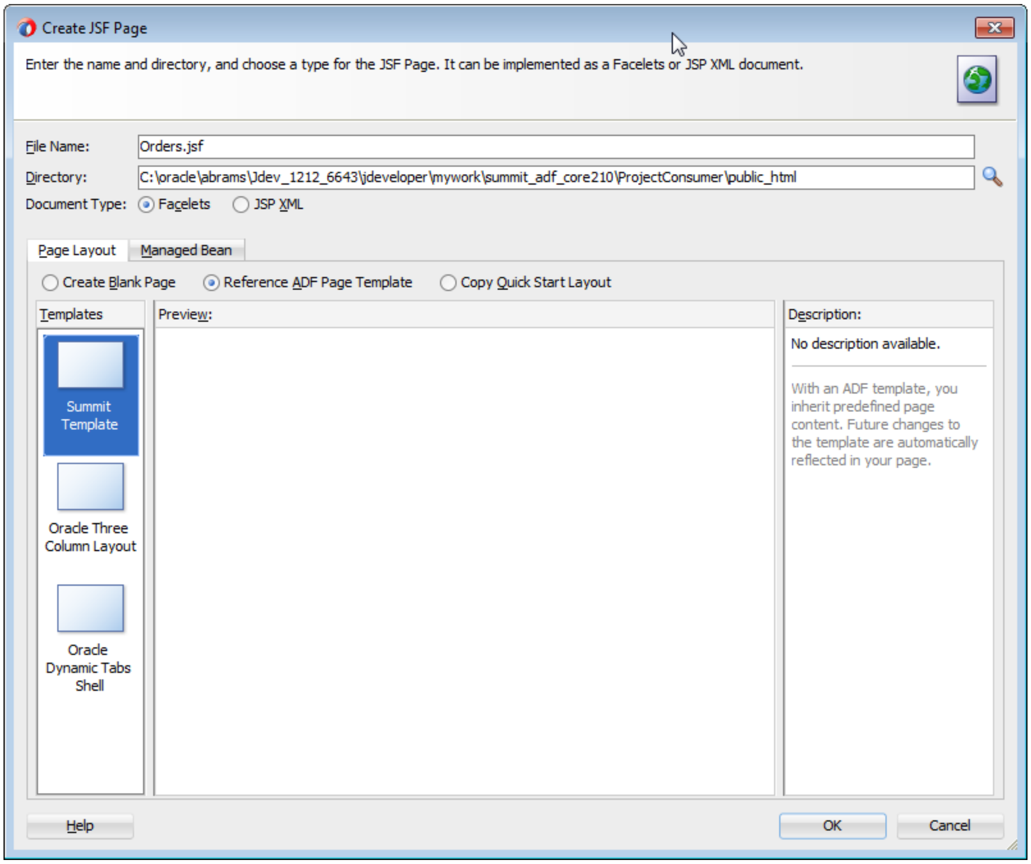
For more information about creating task flows, see Getting Started with ADF Task Flows .

Using Page Templates

When you add a page template to a project, the template will not be exposed in the Applications window. You will not have direct access to individual supporting files, such as image files. However, the template retains its access to its supporting files inside the JAR and is fully reusable within the project. When you apply the template, it will retain all the images that were loaded with the template.

The page template is exposed and accessible when you create a new JSF page using the wizard. When the wizard presents you with the option to use a page template, the ADF Library template will appear in the dropdown list. For example, if you loaded a page template called `SummitTemplate` from an ADF Library, when you use the wizard to create a JSF page, `SummitTemplate` will appear in the wizard, as shown in Figure 50-20. For information on how to use page templates and create a JSF page, see Using Page Templates and the *Developing Web User Interfaces with Oracle ADF Faces*.

Figure 50-20 Using a Page Template from ADF Library



Create JSF Page

Enter the name and directory, and choose a type for the JSF Page. It can be implemented as a Facelets or JSP XML document.

File Name: Orders.jsf

Directory: C:\oracle\abrams\Jdev_1212_6643\jdeveloper\mywork\summit_adf_core210\ProjectConsumer\public_html

Document Type: ● Facelets ○ JSP XML

○ Create Blank Page ● Reference ADF Page Template ○ Copy Quick Start Layout

Description:

No description available.

With an ADF template, you inherit predefined page content. Future changes to the template are automatically reflected in your page.

Using Declarative Components

When you add a declarative component to a project, the JSP tag libraries that contain the component will be added to the project. The tag libraries will appear in the

Components window, and the declarative components will be available for selection. For information about creating and using declarative components, see the Using Declarative Components section of the *Developing Web User Interfaces with Oracle ADF Faces* .

What You May Need to Know About Differentiating ADF Library Components

If you mix components created during application development with components imported from the ADF Library, you may be able to differentiate between them by using the tooltips feature of JDeveloper.

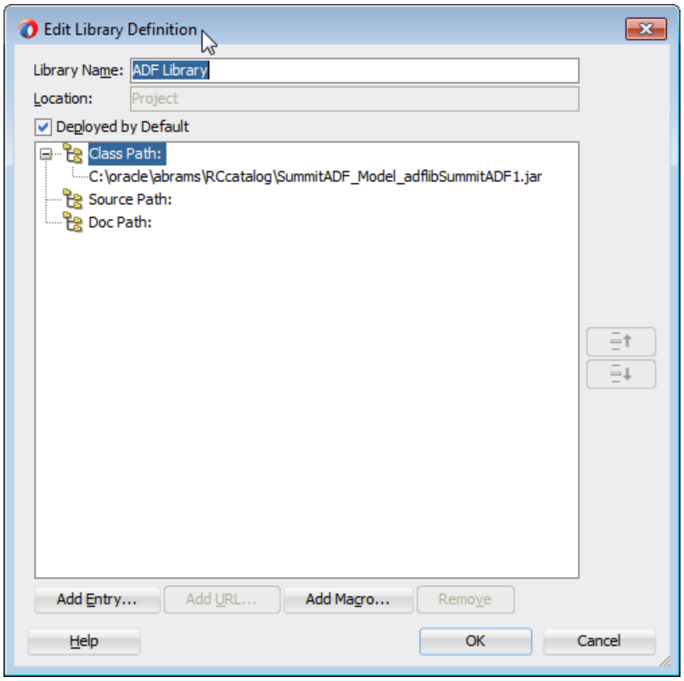
Move the cursor over an application module or data control and you will see the full path of the source. If you see the ADF Library JAR file in the path, that means the component source is the ADF Library.

> **Note:**
>
> The tooltip for items in the Data Controls panel varies depending on whether or not the node has ever been opened in this session. To ensure that you are seeing the most complete tooltip, you should expand the item.

What Happens at Runtime: How ADF Libraries are Added to a Project

After an ADF Library JAR has been added to a project and to the class path, it behaves like any other library file. During runtime, any component that uses the component in the ADF Library JAR will reference that object. The process is transparent and there is no need to distinguish between components that were developed for the project and those that are in ADF Library JARs. Figure 50-21 shows the ADF Library and the path to the JAR as defined for a project.

Figure 50-21 Edit Library Definition Dialog Showing the ADF Library in the Class Path



Edit Library Definition

Library Name: ADF Library

Location: Project

☑ Deployed by Default

Class Path:
C:\oracle\abrams\RCcatalog\SummitADF_Model_adflibSummitADF1.jar
Source Path:
Doc Path:

> **Note:**
>
> Once an ADF Library JAR is created, any Java source used to create the component is not available in the JAR. If you want your consumers to have access to the source, such as for debugging, they must be delivered separately (usually in ZIP files). The consumer can then designate those ZIP files as the related component source for the **Source Path** entry in their ADF Library **Libraries and Classpath** entry in their own projects.

Removing an ADF Library JAR from a Project

You can remove an ADF Library JAR from your project manually or by using the Resources window.

You can use the Resources window to remove an ADF Library JAR from a project, or you can manually remove the JAR using the Project Properties dialog. You can remove an ADF Library JAR only if the components in the project do not have any dependencies on the components in the ADF Library JAR.

When you remove a JAR, it will no longer be in the project class path and its components will no longer be available for use.

How to Remove an ADF Library JAR from a Project Using the Resources Window

The Resources window allows you to remove previously added ADF Library JARs from a project using a simple command.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you package an ADF component. For more information, see Removing an ADF Library JAR from a Project.

You may also find it helpful to understand functionality that can be used with packaging an ADF component. For more information, see Additional Functionality for Reusable ADF Components.

You will need to complete this task:

> Add the ADF Library JAR using the Resources window. For more information, see Adding ADF Library Components into Projects.

To remove an ADF Library JAR from the project using the Resources Window:

1. From the main menu, choose **Window** > **Resources**.
2. In the Applications window, select the project that has the ADF Library JAR you want to remove.
3. In the Resources window, locate the ADF Library JAR you want to remove from the current project.
4. Right-click the JAR and choose **Remove from Project**.

How to Remove an ADF Library JAR from a Project Manually

When you remove an ADF Library JAR manually, be sure to remove the correct library. Be aware of other libraries that are critical to the operation of the project.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you package an ADF component. For more information, see Removing an ADF Library JAR from a Project.

You may also find it helpful to understand functionality that can be used with packaging an ADF component. For more information, see Additional Functionality for Reusable ADF Components.

You will need to complete this task:

> Add the ADF Library JAR manually. For more information, see Adding ADF Library Components into Projects.

To remove an ADF Library JAR from the project manually:

1. In the Applications window, double-click the project.
2. In the Project Properties dialog, select the **Libraries and Classpath** node.
3. In the **Classpath Entries** list, select **ADF Library** and click **Edit**.

4. In the Edit Library Definition dialog, select the ADF Library JAR you want to remove under the **Class Path** node, and click **Remove**.
5. Click **OK** to accept the deletion, and click **OK** again to exit the dialog.

51
Customizing Applications with MDS

This chapter describes how to develop Fusion web applications that can be customized and subsequently deployed by a customer, using the customization features provided by Oracle Metadata Services (MDS) framework. It also covers how to implement customizations on such applications.
This chapter includes the following sections:

For information on how to deploy customized applications, see Deploying the Application.

About Customization and MDS

Oracle Metadata Services (MDS) is the framework for handling ADF application customizations. Customizations are stored in a metadata repository and merged with base metadata at runtime.

Using the customization features provided by MDS, you can create applications that fall into the following customization patterns:

- Seeded customization

  Seeded customization of an application is the process of taking a generalized application and making modifications to suit the needs of a particular group, such as a specific industry or site. Seeded customizations exist as part of the deployed application, and endure for the life of a given deployment. This chapter describes how to create a customizable application, and then customize the application using JDeveloper.

- User customization (change persistence)

  User customization allows an end user to change the content of the application at runtime to suit individual preferences (for example, which columns are visible in a table), and have those changes "remembered" the next time the user opens the application. For information about user customization, see Allowing User Customizations at Runtime .

For more information about the MDS architecture and metadata repositories (database- and file-based) and archives (EAR, MAR), see Managing the MDS Repository in *Administering Oracle Fusion Middleware*.

Customization and Layers: Use Cases and Examples

A customized application contains a base application and one or more layers containing customizations. MDS stores the customizations in a metadata repository and retrieves them at runtime to merge the customizations with the base metadata to reveal the customized application. Since the customizations are saved separately from the base, the customizations are upgrade safe; a new patch to base can be applied without breaking customizations. When a customized application is launched, the customization content is applied over the base application.

For example, say you have a generalized payroll application with a validation rule that limits the salary field to 4000. Then you create a customization of that validation rule that limits the salary field to 3300. At runtime, the customization is applied to the base application and the validation rule for the salary field limits it to 3300.

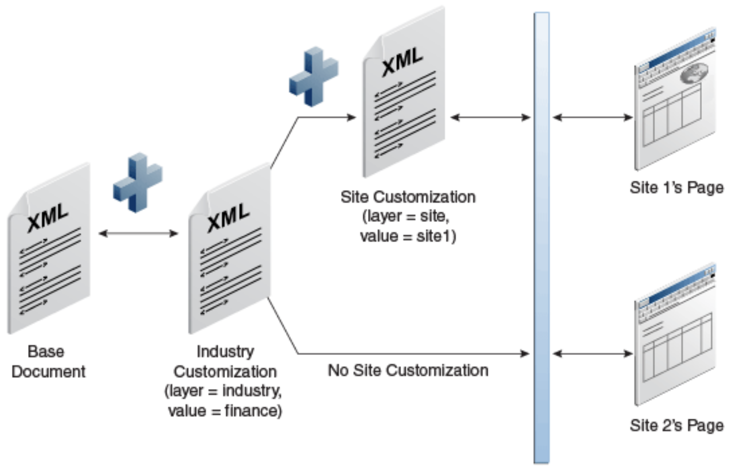
A customizable application can have multiple **customization layer**s. Examples of customization layers are `industry` and `site`. Each layer can have multiple **customization layer value**s, but typically only one such layer value from each layer is applied at runtime. For example, the `industry` layer for a customizable application can contain values for `healthcare` and `financial` industries; but in the deployed customized application, only one of the values from this layer is used at a time.

Layer values from multiple customization layers can be applied, in a specified order of precedence, on top of the base metadata. For example, a customized application can contain customizations in the `financial` layer value of the `industry` layer and the `Financial Company #1` layer value of the `site` layer. Each customization layer corresponds to a customization document that contains a set of instructions that change the underlying metadata.

The **customization context** of a customized application is defined by the set of customization layer values applied to it.

Figure 51-1 illustrates how layers are applied in a customized application.

Figure 51-1 Example of Layered Customization



To support this, you use JDeveloper to create customization classes, define layers and values, and specify the order of precedence. These processes are described in Developing a Customizable Application.

Static and Dynamic Customization Content

Customizations can be categorized as either static or dynamic. Static customizations have only one layer value in effect for all executions of the application, while dynamic customizations can have values that vary based on the execution context of the application. If a customization can vary for different users executing the application, then it is dynamic. If a customization has the same value for all users executing the application then it is static.

When you implement customizations in **ADF Business Components** objects, the customizations remain the same for entire runtime of the application. This is because these objects are loaded only once for an application and reused for the duration of the application. For example, you can have a customized validation rule in the Healthcare Company #1 value of the site layer that limits salaries for that site to 3300. This is static customization content.

However, you can also implement customizations at the controller or view level that allow the layer value to be determined at runtime, based on user roles (responsibilities) or other application-specific criteria. For example, you can design an application so that users from different organizations see different sets of fields on a given screen. This is dynamic customization content.

The determination of whether a customization is static or dynamic is made in the customization class. In the customization class, if the `getCacheHint()` method returns `ALL_USERS`, then the customization layer is static. For more information about `CacheHint` values, see What You May Need to Know About Customization Classes.

All objects could have a static customization layer, depending on how the customization classes are implemented. But for ADF Business Components objects, customizations can only be static.

Additional Functionality for Customization

You may find it helpful to understand other features before you start working with customization. Following are links to other functionality that may be of interest.

- For information about the MDS architecture and metadata repositories (database- and file-based) and archives (EAR, MAR), see Managing the MDS Repository in *Administering Oracle Fusion Middleware*.
- You can use the ADF Faces change persistence framework to create JSF pages that users can customize at runtime. See Allowing User Customizations at Runtime .

Developing a Customizable Application

If you want to customize an ADF application, you must follow certain procedures after creating the base application.

To create a customizable application, create the base application and perform the following procedures:

To prepare an application for customization:

1. Create the customization classes that will be used, as described in How to Create Customization Classes.
2. Enable seeded customization in the application, as described in How to Enable Seeded Customizations for User Interface Projects.
3. Specify the customization classes in the `adf-config.xml` file, as described in How to Configure the adf-config.xml File .
4. You can optionally restrict runtime customizations on the application, as described in How to Edit Extended Metadata Properties.
5. After you have prepared the application for customization, you must prepare JDeveloper so you can use it to implement the customizations, as described in How to Configure Customization Layers.

How to Create Customization Classes

A customization class is the interface that MDS uses to define which customization applies to the base definition metadata. Each customization class defines a customization layer (for example, `industry` or `site`) and can contain multiple layer values. The customization classes that are used in the application must be available to JDeveloper when customizing the application, and included in the deployed application.

Customization Classes

A customization class evaluates the current context and returns a String result. This String result is used to locate the customization layer.

The customization class provides the following information:

- A name, that represents the name of the layer.
- An array of values, that represent the customization layer values. Typically, each layer returns a single value. If multiple values are returned, the customizations available in the MDS repository for those values are applied in the order in which they appear in the array. A null value or an array containing an empty string ("") indicates there is no customization layer to apply. For more information, see Implementing the getValue() Method in Your Customization Class.
- An IDPrefix, for objects created in the layer. When new objects are created in a customization layer they need a unique ID. The IDPrefix is added to the autogenerated identifier for the object to create an ID for the newly added object. Each layer needs a unique IDPrefix so that objects created at different customization layers will have unique IDs.
- A cache hint, for the layer defined by the customization class. The cache hint defines whether a layer is static or dynamic. If the `getCacheHint()` method returns `ALL_USERS`, then the customization layer is static. For more information about dynamic customizations, see Static and Dynamic Customization Content. For more information about `CacheHint` values, see What You May Need to Know About Customization Classes.

Customizations can be used to tailor an application to suit a specific industry domain (verticalization). Each such domain denotes a customization layer and is depicted using a customization class.

To implement seeded customizations using your customization classes:

- The cust-config section (under mds-config) in the adf-config.xml must contain a reference to the customization classes (as described in How to Configure the adf-config.xml File).

  The customization configuration (cust-config) section provides the customization classes and their precedence for a customized application. See How to Configure the adf-config.xml File .

- The customization classes must be included in the application to support seeded customizations.

  For more information, see Making Customization Classes Available to JDeveloper at Design Time. At runtime, your customization classes must be available in the EAR-level application class loader.

- The layer values must be listed in the CustomizationLayerValues.xml file.

  This file is located in the *jdev_install*\jdev directory. The names of the layers in this file must be consistent with the customization classes. (For more information, see How to Configure Customization Layers.) JDeveloper uses this file to retrieve layer values at design time.

When JDeveloper is launched in the **Customization Developer** role, the Customization Context window displays the available customization layers and layer values. You can select the layer and value to which you want to apply customizations in the Customization Context window. For more information about working in the **Customization Developer** role, see Introducing the Customization Developer Role. The layer you choose to customize is called the **tip layer**. For more information, see Introducing the Tip Layer.

The following example shows a sample customization class. Note that all customization classes should have a single, no-argument constructor.

```
package mycompany;

import java.io.IOException;
import java.io.InputStream;
import java.util.Properties;
import oracle.mds.core.MetadataObject;
import oracle.mds.core.RestrictedSession;
import oracle.mds.cust.CacheHint;
import oracle.mds.cust.CustomizationClass;

public class IndustryCC extends CustomizationClass {
  private static final String DEFAULT_LAYER_NAME = "industry";
  private String mLayerName = DEFAULT_LAYER_NAME;

  public IndustryCC() {
  }
  public CacheHint getCacheHint() {
    return CacheHint.ALL_USERS;
  }
  public String getName() {
    return mLayerName;
  }

  public String generateIDPrefix(RestrictedSession sess, MetadataObject mo) {
      return "I";
  }
  public String[] getValue(RestrictedSession sess, MetadataObject mo) {
    // This needs to return the appropriate value at runtime.
    return new String[] {"financial"};
  }
}
```

This example shows the four methods that define how the customization class will work: `getCacheHint()`, `getName()`, `generateIDPrefix()`, and `getValue()`.

The return value for `getCacheHint()` indicates to MDS how widely visible a metadata object with this customization is, and therefore its likely duration in memory. It also defines whether or not a layer is static or dynamic. In this example, the `getCacheHint()` method returns `ALL_USERS`, which means the customization layer is static. For more information about `CacheHint` values, see What You May Need to Know About Customization Classes.

The `getName()` method returns the name of the customization layer. A SiteCC customization class, for example, might return "site." In this example, the `getName()` method returns "industry."

The `generateIDPrefix()` method creates an IDPrefix. The IDPrefix is a unique, abbreviated string that identifies the name and value of the layer. It is used as the prefix of the ID for objects created in the customization layer. The default implementation (if no IDPrefix is specified) returns name of the layer concatenated with customization value. For performance reasons, the IDPrefix should be kept short (4 characters or less). Therefore, the default implementation should be overridden. The related `getIDPrefix()` method returns the IDPrefix of the customization layer. In this example, the `getIDPrefix()` method would return "I" (for "industry").

The `getValue()` method returns the customization value (or values) for the customization class. In this example, the `getValue()` method returns a single value, "financial," that defines the customization context when combined with the layer name. There are additional techniques for using the `getValue()` method described in Implementing the getValue() Method in Your Customization Class.

Note:

The possible layer values corresponding to a layer name are retrieved by JDeveloper in the **Customization Developer** role from the `CustomizationLayerValues.xml` file. The precedence of layers is defined by the order of the customization classes specified in the `adf-config.xml` file. The names of the layers must be consistent in these files and in the customization classes.

Implementing the getValue() Method in Your Customization Class

The `getValue()` method is used to retrieve the layer value(s) of the customization class based on the application context and the metadata. For example, calling `getValue()` on a `SiteCC` customization class might return an array with single entry "`headquarters`." Typically, the `getValue()` method returns an array with a single value, as described in Customization Classes.

You can also return multiple values from the `getValue()` method, as shown in the following example.

```
public String[] getValue(RestrictedSession sess, MetadataObject mo) {
    return new String[]{"North America", "US", "CA"}
}
```

When multiple values are returned, customizations applicable to all values are applied. Customizations are applied in the order in which they appear in the array. In this example, North America customizations are applied over the base application, then US customizations are applied, and finally CA.

Note:

Returning multiple values for a customization layer is an advanced concept that is typically unnecessary.

The getValue() method can return a layer value based on the current execution context for the current user, pulled either from static or thread local state maintained by the client, or from properties set by the client on the MDS session and based on the metadata object name. The following example shows this type of implementation.

```
public String[] getValue(RestrictedSession sess, MetadataObject mo) {
    if (mo.getName().equals("/sample/abc.jspx"))
    {
        return new String[]{"Headquarters"};
    }
     else
    {
        return new String[]{"RemoteSite"};
    }
}
```

In this example, the getValue() method uses the getName() method on the metadata object to determine if the name of the metadata document is "/sample/abc.jspx". If so, getValue() returns Headquarters to apply headquarters customizations. If not, it returns RemoteSite to apply customizations for remote sites.

Note:

Coding the getValue() method to return a value based on the metadata object is an advanced concept that is typically unnecessary. Customization context for dynamic layers is typically determined through facets of the application context, such as user name or responsibility.

An additional technique that can be useful during the development cycle is to use an external properties file to specify layer values. The following example references a properties file (customization.properties) that stores the layer values.

```
public String[] getValue(RestrictedSession sess, MetadataObject mo) {
  Properties properties = new Properties();
    String configuredValue = null;
    Class clazz = IndustryCC.class;
    InputStream is = clazz.getResourceAsStream("/customization.properties");
    if (is != null){
      try {
        properties.load(is);
        String propValue = properties.getProperty(mLayerName);
        if (propValue != null){
          configuredValue = propValue;
        }
      } catch (IOException e) {
          e.printStackTrace();
      }
      finally {
        try {
          is.close();
        } catch (IOException e) {
        e.printStackTrace();
        }
      }
    }
  return new String[] {configuredValue};
}
```

When an application using this technique is run in JDeveloper, you can change the layer value in the properties file and refresh the browser to see new customizations being applied. This customization class and properties file combination allows you to maintain the layer value in a separate file so you don't need to modify and recompile Java code to change it for a particular deployment. If you use a `customization.properties` file, it must be packaged with the customization classes so that they are loaded from the same class loader.

The following example shows a sample `customization.properties` file. When the `IndustryCC` class is loaded with this properties file, the layer value `healthcare` is applied.

```
#Configured values for the default layer values
industry=healthcare
site=headquarters
```

Creating a Customization Class

When creating customization classes, put them in a separate project. This allows you to deploy them to a JAR and to import the JAR into your customizable application. This approach increases modularity, making it easier to include the customization classes in multiple applications across your company, and easier to patch them centrally. For more information about this approach, see How to Consume Customization Classes.

Alternatively, you can put the customization classes in the data model project if you are creating customization classes that will be used in a single application only, or if you just want to run the application locally from JDeveloper or to deploy remotely with MDS configuration.

Make sure that there is only one copy of the customization classes in the application and that they are packaged in a JAR file so that they are loaded at the EAR-level application class loader. By default, adding project dependencies will add the customization classes to the WAR profile, which will not work correctly after the application is packaged and deployed. For more information about packaging your application for deployment, see How to Create Deployment Profiles.

Use the following procedure to create a customization class.

Before you begin:

It may be helpful to have an understanding of the features of customization classes. For more information, see Customization Classes.

You may also find it helpful to understand additional customization functionality that can be added to your applications. For more information, see Additional Functionality for Customization.

You will need to launch JDeveloper using the **Studio Developer** role, and open (or create) the application that will hold the customization classes.

To create a customization class:

1. From the main menu, choose **File** and then **From Gallery**.
2. In the New Gallery, expand **General**, select **Projects** and then **Java Application Project**, and click **OK**.
3. In the Create Java Project dialog, specify the appropriate settings for your project, and click **Finish**.

   > **Note:**
   >
   > You should use a single project to hold all of your customization classes.

   For more information about the options in the Create Java Project dialog, see the online help.
4. In the Applications window, right-click your project and choose **Project Properties**.
5. In the Project Properties dialog, select **Libraries and Classpath**, and then click **Add Library**.
6. In the Add Library dialog, select **MDS Runtime**, and click **OK**, and then click **OK** to close the Project Properties dialog.
7. In the Applications window, right-click the project and choose **New** and then **Java Class**.
8. In the Create Java Class dialog, enter a name for the class and the appropriate package.

   > **Note:**
   >
   > Customization classes are typically named for the layer name they return. For example, a customization class that returns the layer name `industry` would be named `IndustryCC`.
9. In the **Extends** field, enter `oracle.mds.cust.CustomizationClass`.

   For more information about this class, refer to the Javadoc.
10. Make sure that **Implement Abstract Methods** is selected and click **OK**.
11. Replace the code in the generated file with code like that described in Customization Classes.

12. Save your changes and rebuild the project.

    This creates the customization class. The sample code uses the package name `mycompany` and the class name `IndustryCC`. You will need to change these as appropriate for your application.

What You May Need to Know About Customization Classes

As described in Customization Classes, the customization class defines a `CacheHint` which specifies the visibility of metadata objects in a customization layer, and therefore its likely duration in memory. This information is used by MDS to decide whether or not to cache a customization, and where to cache it. Any customization layers constructed using this customization class have this cache hint.

The following constants are supported values of `CacheHint`:

- `ALL_USERS` — The customization is applied globally (unconditionally) for a given deployment. This constant is used for static customization layers.
- `MULTI_USER` — The customization is applied to multiple users.
- `REQUEST` — The customization is applied for the duration of the request only.
- `USER` — The customization is applied for a single user to documents which are accessed throughout the user's application session. (In web applications the application session is typically a servlet session.)

> **Note:**
>
> Customization classes are likely to be executed frequently, once for each document being accessed to get the layer name and layer value, so take care to ensure their efficiency.

How to Consume Customization Classes

After you have created your customization classes, you can use them at design time in the **Customization Developer** role, and at runtime in the application. To be consumed in an application or in JDeveloper, the classes must be packaged appropriately.

Making Customization Classes Available to JDeveloper at Design Time

After you create the customization classes, you must make them available to JDeveloper so that you can use them when implementing customizations while working in the **Customization Developer** role.

Because the customization classes are reusable components, you can create a separate project to contain them, and package them into their own JAR file.
You can then import the JAR into the consuming application, which makes the customization classes available to JDeveloper. For more information about creating the customization classes in a separate project, see Creating a Customization Class. For more information about packaging the customization classes into a JAR file, see Adding Customization Classes into a JAR.

> **Note:**
>
> This procedure is not required if you created your customization classes in the data model project of the consuming application.

Use the following procedure to make the customization classes visible to the application, and then add the customization classes to the `cust-config` section of the `adf-config.xml` file, as described in How to Configure the adf-config.xml File.

Before you begin:

It may be helpful to have an understanding of how customization classes are packaged. For more information, see Customization Classes.

You may also find it helpful to understand additional customization functionality that can be added to your applications. For more information, see Additional Functionality for Customization.

You will also need to complete these tasks:

- Create your customization classes in an external project, as described in Creating a Customization Class.
- Create a JAR using the procedure described in Adding Customization Classes into a JAR.
- Launch JDeveloper using the **Studio Developer** role, and open the application that you want to customize.

To use customization classes from an external project:

1. In the Applications window, click the **Application Menu** icon and choose **Application Properties**.
2. In the Application Properties dialog, select **Libraries and Classpath**, and click **Add JAR/Directory**.
3. In the Add Archive or Directory dialog, select the JAR file you created that contains the customization classes, and click **Open**.
4. Click **OK**.

   Now the customization classes are available to JDeveloper for customization, and for running your project locally in JDeveloper. They will also be packaged to the EAR class path when you package the application.

Making Customization Classes Available to the Application at Runtime

When you package and deploy your customized application, the customization classes must be available at the application level on the application's class path.

When you define the deployment profiles for your application, you need to add the customization classes JAR file to the EAR assembly, and to avoid duplication, you need to make sure that the WAR profile does not include the customization classes JAR file. For more information, see Creating an Application Level EAR Deployment Profile.

How to Enable Seeded Customizations for User Interface Projects

Like all customizable components, the XML elements of a customizable metadata object must be uniquely identifiable by MDS, and therefore must have a unique, non-null identifier. The component's identifier is used to refer to the element in the customization instructions in the customization layer. The ID property is the identifier for each type of component in an ADF Faces `.jspx` or `.jsff` file.

To allow for customizations on your JSF and JSP pages, you must enable seeded customizations in the application's user interface project, which drives some defaults for your pages. This is not necessary for your model and controller projects.

> **Note:**
>
> MDS requires that pages be XML-based to be customized. Therefore, customizations are not allowed on `.jsp` files; use `.jspx` files instead.
>
> Additionally, facelets files must have a `.jsf` extension to be customizable. MDS uses this extension to recognize it as a Facelets file.

Before you begin:

It may be helpful to have an understanding of how seeded customization works. For more information, see Customization and Layers: Use Cases and Examples.

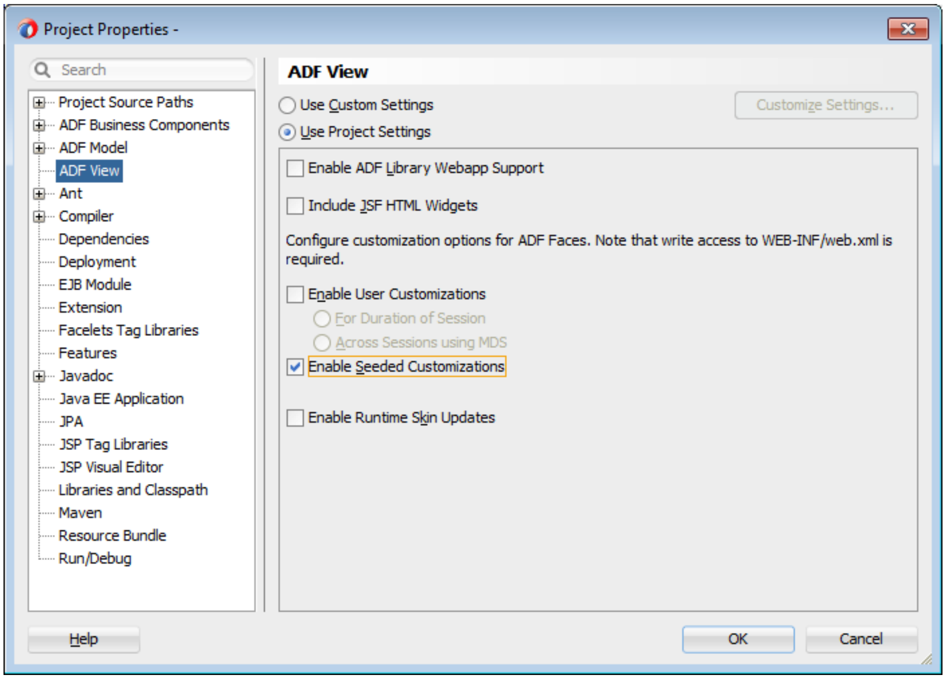
You may also find it helpful to understand additional customization functionality that can be added to your applications. For more information, see Additional Functionality for Customization.

You will also need to launch JDeveloper using the **Studio Developer** role, and open the application that you want to make customizable.

To enable seeded customizations in your user interface project:

1. In the Applications window, right-click the user interface project and choose **Project Properties**.
2. In the Project Properties dialog, select **ADF View** and then select the **Enable Seeded Customizations** checkbox, as shown in Figure 51-2.
3. Click **OK**.
4. Save the changes to your project.

Figure 51-2 Project Properties - Enable Seeded Customizations



ADF View

○ Use Custom Settings

● Use Project Settings

□ Enable ADF Library Webapp Support

□ Include JSF HTML Widgets

Configure customization options for ADF Faces. Note that write access to WEB-INF/web.xml is required.

□ Enable User Customizations

○ For Duration of Session

○ Across Sessions using MDS

☑ Enable Seeded Customizations

□ Enable Runtime Skin Updates

How to Enable Seeded Customizations in Existing Pages

If you have pages in your project that were created in an earlier version of JDeveloper, you must make sure that these preexisting pages are also enabled for seeded customizations. This is only necessary if you migrated the application from an earlier version of JDeveloper and did not generate IDs during migration.

Before you begin:

It may be helpful to have an understanding of how seeded customization works. For more information, see Customization and Layers: Use Cases and Examples.

You may also find it helpful to understand additional customization functionality that can be added to your applications. For more information, see Additional Functionality for Customization.

You will also need to launch JDeveloper using the **Studio Developer** role, and open the application that you want to make customizable.

To enable seeded customizations in an existing page:

1. Create an audit profile to implement ID tokens for all XML objects in your page:
   a. From the main menu, choose **Tools > Preferences**.
   b. In the Preferences dialog, select **Audit > Profiles**.
   c. On the Profiles page, click **Rules** and deselect all rules.
   d. Expand **Application Development > ADF Faces > Component ID Rules**.
   e. Select the rule **Check for ID When ADF Faces is Present**.
   f. From the **Default Fix** dropdown list, select **Generate a unique ID**.

g. Click **Save As**, then enter an identifiable name for the profile (such as Generate Unique IDs). and click **Save**.

h. Click **OK** to close the Preferences dialog.

2. In the Applications window, select the page for which you want to enable seeded customizations. Alternatively, you can select a project to run the audit on all the files it contains.
3. From the main menu, choose **Build > Audit**.
4. In the Audit dialog, select the profile you created to generate IDs and click **Run**.
5. Use the Log window to review issues and apply fixes.
6. When the audit is complete, save your changes.

How to Enable Customizations in Resource Bundles

If you plan to create new resource keys when implementing customizations, you can specify the affected resource bundles using the Application Properties dialog.

Before you begin:

It may be helpful to have an understanding of how seeded customization works. For more information, see Customization and Layers: Use Cases and Examples.

You may also find it helpful to understand additional customization functionality that can be added to your applications. For more information, see Additional Functionality for Customization.

You will also need to launch JDeveloper using the **Studio Developer** role, and open the application that you want to make customizable.

To enable customizations in resource bundles:

1. From the Applications window context menu, choose **Application Properties**.
2. In the Application Properties dialog, click **Resource Bundles**, and then click **Add**.
3. In the Select Resource Bundle dialog, navigate to and select the resource bundles for which you want to enable customization.
4. Click **Open**.
5. In the Application Properties dialog, select the checkbox in the **Overridden** column of the **Bundle** table.
6. Click **OK**.

How to Configure the adf-config.xml File

The application's `adf-config.xml` file must have an appropriate `cust-config` element in the `mds-config` section. The `cust-config` element allows clients to define an ordered and named list of customization classes. You use the overview editor for the `adf-config.xml` file to add customization classes (see Figure 51-3).

Before you begin:

It may be helpful to have an understanding of how customization classes are created and packaged. For more information, see Customization Classes.

You may also find it helpful to understand additional customization functionality that can be added to your applications. For more information, see Additional Functionality for Customization.

You will also need to complete these tasks:

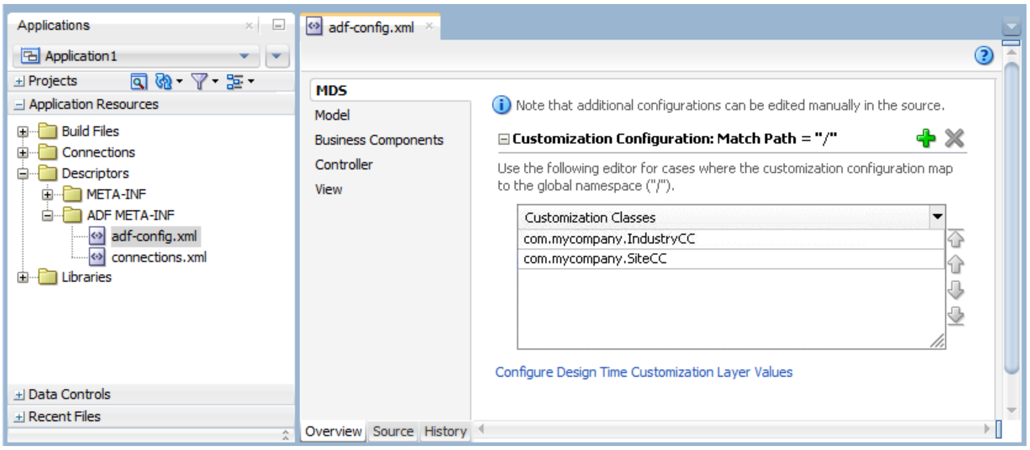
- Create your customization classes, as described in Creating a Customization Class.
- Create a JAR using the procedure described in Adding Customization Classes into a JAR.
- Make your classes available to JDeveloper, as described in Making Customization Classes Available to JDeveloper at Design Time.
- Launch JDeveloper using the **Studio Developer** role, and open the application that you want to customize.

To identify customization classes in the adf-config.xml file:

1. In the Application Resources panel, expand the **Descriptors** and **ADF META-INF** nodes, and then double-click **adf-config.xml**.
2. In the overview editor, click the **MDS Configuration** navigation tab and then click the **Add** icon.
3. In the Edit Customization Class dialog, search for or navigate to the customization classes you have already created.
4. Select the appropriate classes and click **OK**.
5. After you have added all of the customization classes, you can use the arrow icons to put them in the appropriate order.

Figure 51-3 shows the overview editor for the `adf-config.xml` file with two customization classes added.

Figure 51-3 adf-config.xml Overview Editor



Note that additional configurations can be edited manually in the source.

Customization Configuration: Match Path = "/"

Use the following editor for cases where the customization configuration map to the global namespace ("/").

Customization Classes
com.mycompany.IndustryCC
com.mycompany.SiteCC

Configure Design Time Customization Layer Values

The order of the customization-class elements defines the precedence of customization layers. For example, in the code shown in the following example, the `IndustryCC` class is listed before the `SiteCC` class. This means that customizations at the industry layer are applied to the base application, and then customizations at the site layer are applied.

```
<adf-config xmlns="http://xmlns.oracle.com/adf/config">
  <adf-mds-config xmlns="http://xmlns.oracle.com/adf/mds/config">
    <mds-config xmlns="http://xmlns.oracle.com/mds/config" version="11.1.1.000">
      <cust-config>
        <match path="/">
          <customization-class name="com.mycompany.IndustryCC"/>
          <customization-class name="com.mycompany.SiteCC"/>
        </match>
      </cust-config>
    </mds-config>
  </adf-mds-config>
</adf-config>
```

What Happens When You Create a Customizable Application

When you create a customizable application, you have a base application that includes the pieces necessary for you or someone else to use as the basis for a customized application.

To perform the customization, you must open the application in JDeveloper using the **Customization Developer** role, as described in Customizing an Application.

What You May Need to Know About Customizable Objects and Applications

Oracle ADF components (such as controller, model, and business components objects) must have a unique identifier so that they can be customized. ADF components generated by JDeveloper are created with identifiers by default, with the exception of JSP and JSF pages in your user interface projects. To cause JDeveloper to generate identifiers for components on pages in your user interface projects, you must explicitly specify this at the project level (as explained in How to Enable Seeded Customizations for User Interface Projects).

Before you implement customizations in an application, make sure that all objects that you intend to customize have the necessary identifiers. In many cases, you can run an audit rule to catch and fix any omissions (as explained in How to Enable Seeded Customizations in Existing Pages).

Also, take care to ensure the efficiency of your customization classes, because they can be executed frequently, once for each document being accessed to get the layer name and layer value.

Customizing an Application

Customizations are made with Oracle JDeveloper’s Customization Developer role. Customizations are made for each defined layer and only existing metadata is customizable.

Using the **Customization Developer** role, you can create customizations in a customizable application.

Introducing the Customization Developer Role

The **Customization Developer** role is used to customize the metadata in a project. Customization features are available only in this role. When you are in the **Customization Developer** role, you can do the following:

- Create and update customizations
- Select and edit the tip layer of a customized application
- Remove existing customizations

When you are in the **Customization Developer** role, the source editor is read-only and the following JDeveloper features are disabled:

- Workspace migration
- Creation, deletion, and modification of application and IDE connections. You must configure connections in Default role before opening an application in Customization Developer role.

When working with an application in the **Customization Developer** role, new objects cannot be created, and noncustomizable objects cannot be modified. However, the following content types are customizable:

- Portal modules
- ADF modules, including ADF Faces, ADF Model, ADF Business Components, and ADF Controller

> **Note:**
>
> ADF Business Components objects are customizable only if a static customization class is selected in the Customization Context window. Otherwise, business components objects are read-only.

When working in the **Customization Developer** role, you cannot edit noncustomizable files, such as Java classes, resource bundles, security policies, deployment descriptors, and configuration files. Noncustomizable files are indicated by a lock icon when you are working in the **Customization Developer** role.

> **Note:**
>
> Facelets files must have a `.jsf` extension to be customizable. MDS uses this extension to recognize it as a Facelets file.

You are also restricted from modifying project settings and customizing certain ADF Business Components features, including service interfaces and business event definitions. Additionally, you cannot refactor, or make changes to customizable files that would, in turn, necessitate changes in noncustomizable files.

How to Switch to the Customization Developer Role in JDeveloper

The customization features of JDeveloper are available to you in the **Customization Developer** role. To work in this role, you can either choose it when you start JDeveloper or, if JDeveloper is already running, you can use the **Switch Roles** menu to switch to the **Customization Developer** role.

Before you begin:

It may be helpful to have an understanding of the features of the **Customization Developer** role. For more information, see Introducing the Customization Developer Role.

You may also find it helpful to understand additional customization functionality that can be added to your applications. For more information, see Additional Functionality for Customization.

You will also need to launch JDeveloper.

To switch to the Customization Developer role in JDeveloper:

- From the main menu, choose **Tools > Switch Roles > Customization Developer**.

> **Note:**
>
> You can optionally toggle the **Tools > Switch Roles > Always Prompt for Role Selection at Startup** menu item, to specify whether or not you want to choose the role when JDeveloper is launched. If deselected, JDeveloper launches in the role it was in when you last closed it.

Introducing the Tip Layer

When working in the **Customization Developer** role, the layer and layer value combination that is selected in the Customization Context window is called the **tip layer**. The changes you make while in the **Customization Developer** role are applied to this layer.

> **Note:**
>
> When working in the **Customization Developer** role, if the Customization Context window is not displayed, you can access it from the **Window** menu.

The metadata displayed in the JDeveloper editors is a combination of the base metadata and the customization layers up to and including the tip layer, according to the precedence set in `adf-config.xml`, with the values specified in the Customization Context window for each layer.

When working in the **Customization Developer** role, you can also see the noncustomized state of the application. When you select **View without**

Customizations in the Customization Context window, there is no current tip layer. Therefore, what you see is the noncustomized state. While you are in this view, all customizable files show the lock icon (in the Applications window), indicating that these files are read-only.

When you make customizations in a tip layer, these customizations are indicated by an orange icon in the Properties window. A green icon indicates non-tip layer customizations. When you see an orange icon beside a property, you have the option of deleting that customization by choosing **Remove Customization** from the dropdown menu for that property.

How to Configure Customization Layers

To customize an application, you must specify the customization layers and their values in the `CustomizationLayerValues.xml` file so that they are recognized by JDeveloper.

When you open a customizable application in the **Customization Developer** role, JDeveloper reads the `adf-config.xml` file to determine the customization classes to use and their order of precedence. JDeveloper also reads the `CustomizationLayerValues.xml` file to determine the layer values to make available in the Customization Context window. If there are layer values defined in the `CustomizationLayerValues.xml` file that are not defined in the customization classes listed in the `adf-config.xml` file, they are not displayed in the Customization Context window.

Therefore, you can have a comprehensive list of layer values for all of your customization projects in the `CustomizationLayerValues.xml` file, and only those appropriate for the current application are available in the Customization Context window. Conversely, you could have a comprehensive list of customization classes for an application in the `adf-config.xml` file, and only the subset of layer values that you will work on in your `CustomizationLayerValues.xml` file.

> **Note:**
>
> At design time, JDeveloper retrieves customization layer values from the `CustomizationLayerValues.xml` file. But at runtime, the layer values are retrieved from the customization class.

The names of the layers and layer values that you enter in the `CustomizationLayerValues.xml` file must be consistent with those specified in your customization classes. The following example shows the contents of a sample `CustomizationLayerValues.xml` file.

```
<cust-layers  xmlns="http://xmlns.oracle.com/mds/dt">
  <cust-layer name="industry" id-prefix="i">
    <cust-layer-value value="financial" display-name="Financial" id-prefix="f"/>
    <cust-layer-value value="healthcare" display-name="Healthcare" id-
prefix="h"/>
  </cust-layer>
  <cust-layer name="site" id-prefix="s">
    <cust-layer-value value="headquarters" display-name="HQ" id-prefix="hq"/>
    <cust-layer-value value="remoteoffices" display-name="Remote" id-
prefix="rm"/>
```

```
  </cust-layer>
</cust-layers>
```

For each layer and layer value, you can add an id-prefix token. This helps to ensure the uniqueness of the id, so that customizations are applied accurately. When you add a new element (such as a command button) to a page during customization, JDeveloper adds the id-prefix of the layer and layer value (determined by the selected tip layer) to the autogenerated identifier for the element to create an id for the newly added element in the customization metadata file. As the example above shows, the site layer has an id-prefix of "s" and the headquarters layer value has an id-prefix of "hq". So, when you select site/headquarters as the tip layer and add a command button to a page, the command button will have an id of "shqcb1" in the metadata customization file.

For each layer value, you can also add a display-name token to provide a human-readable name for the layer value. When you are working in the **Customization Developer** role, the value of the display-name token is shown in the Customization Context window for that layer value.

For each layer, you can optionally provide a value-set-size token that defines the size of the value set for the customization layer. This can be useful, for example, when using a design-time, application-specific CustomizationLayerValues.xml file. By setting value-set-size to "no_values" you can exclude runtime-only layers at design time.

```
<cust-layer name="runtime_only_layer" value-set-size="no_values"/>
```

You can define the customization layer values either globally for JDeveloper or in an application-specific file. If you use an application-specific file, it takes precedence over the global file. For more information on configuring layer values globally for JDeveloper, see Configuring Layer Values Globally. For more information on configuring application-specific layer values, see Configuring Workspace-Level Layer Values from the Studio Developer Role.

Configuring Layer Values Globally

The following procedure describes how to configure the CustomizationLayerValues.xml file globally for JDeveloper.

Before you begin:

It may be helpful to have an understanding of layers and layer values. For more information, see How to Configure Customization Layers.

You may also find it helpful to understand additional customization functionality that can be added to your applications. For more information, see Additional Functionality for Customization.

You will also need to complete these tasks:

- Create your customization classes, as described in Creating a Customization Class.
- Make your classes available to JDeveloper, as described in Making Customization Classes Available to JDeveloper at Design Time.

To configure design time customization layer values globally for JDeveloper:

1. Locate and open the CustomizationLayerValues.xml file.

You can find this file in the jdev subdirectory of your JDeveloper installation directory (*jdev_install*\jdev\CustomizationLayerValues.xml).

2. For each layer, enter a cust-layer element, as described in How to Configure Customization Layers.
3. For each layer value, enter a cust-layer-value element, as described in How to Configure Customization Layers.
4. Save and close the CustomizationLayerValues.xml file.
5. After you have made changes to the global CustomizationLayerValues.xml file, you must restart JDeveloper.

Configuring Workspace-Level Layer Values from the Studio Developer Role

When configuring layer values for an application, you can use either the **Studio Developer** role or the **Customization Developer** role. Note that when you configure an application-specific CustomizationLayerValues.xml file, you can create and modify layer values, but you cannot create additional customization layers. It is not necessary to restart JDeveloper to pick up changes made to the application-specific layer values.

When you create an application-specific CustomizationLayerValues.xml file, JDeveloper stores it in an application-level directory (for example, *workspace-directory*\.mds\dt\customizationLayerValues\CustomizationLayerValues.xml). You can access this file in the Application Resources panel of the Applications window, under the **MDS DT** node.

The following procedure describes how to configure the CustomizationLayerValues.xml file for a specific application from the **Studio Developer** role. This procedure can also be used from the **Customization Developer** role.

Before you begin:

It may be helpful to have an understanding of layers and layer values. For more information, see How to Configure Customization Layers.

You may also find it helpful to understand additional customization functionality that can be added to your applications. For more information, see Additional Functionality for Customization.

You will also need to complete these tasks:

- Create your customization classes, as described in Creating a Customization Class.
- Make your classes available to JDeveloper, as described in Making Customization Classes Available to JDeveloper at Design Time.
- Launch JDeveloper using the **Studio Developer** role, and open the application that you want to customize.

To configure design time customization layer values at the workspace level from the Studio Developer role:

1. In the Application Resources panel, expand the **Descriptors** and **ADF META-INF** nodes, and then double-click **adf-config.xml**.
2. In the overview editor, click the **MDS Configuration** navigation tab.

3. On the MDS Configuration page, below the table of customization classes, click the **Configure Design Time Customization Layer Values** link to open the workspace-level `CustomizationLayerValues.xml` file.

   When you click the link, the file opens in the source editor. If the override file doesn't already exist, JDeveloper displays a confirmation dialog. Click **Yes** to create and open a copy of the global file.

4. In the file, specify layer values as necessary, as described in How to Configure Customization Layers.
5. Save your changes.

Configuring Workspace-Level Layer Values from the Customization Developer Role

You can also configure layer values for an application from the **Customization Developer** role. Note that when you configure an application-specific `CustomizationLayerValues.xml` file, you can create and modify layer values, but you cannot create additional customization layers. It is not necessary to restart JDeveloper to pick up changes made to the application-specific layer values.

When you create an application-specific `CustomizationLayerValues.xml` file, JDeveloper stores it in an application-level directory (for example, `*workspace-directory*\.mds\dt\customizationLayerValues\CustomizationLayerValues.xml`). You can access this file in the Application Resources panel of the Applications window, under the **MDS DT** node.

The following procedure describes how to configure the `CustomizationLayerValues.xml` file for a specific application from the **Customization Developer** role.

Before you begin:

It may be helpful to have an understanding of layers and layer values. For more information, see How to Configure Customization Layers.

You may also find it helpful to understand additional customization functionality that can be added to your applications. For more information, see Additional Functionality for Customization.

You will also need to complete these tasks:

- Create your customization classes, as described in Creating a Customization Class.
- Make your classes available to JDeveloper, as described in Making Customization Classes Available to JDeveloper at Design Time.
- Launch JDeveloper using the **Customization Developer** role, and open the application that you want to customize.

To configure design time customization layer values at the workspace level from the Customization Developer role:

1. In the Customization Context window, click the **Configure application layer values** link.

When you click the link, the `CustomizationLayerValues.xml` file opens in the source editor. If the override file doesn't already exist, JDeveloper displays a confirmation dialog. Click **Yes** to create and open a copy of the global file.

2. Specify layer values as necessary, as described in How to Configure Customization Layers.
3. Save your changes.

   After you make changes to the application-specific `CustomizationLayerValues.xml` file while you are in the **Customization Developer** role, any tip layer you have selected in the Customization Context window is deselected. You can then select the desired tip layer.

How to Customize Metadata in JDeveloper

You use the same development procedures and techniques to customize metadata that you use when developing the base application. To implement customizations, however, you must be working in the **Customization Developer** role and specify the customization context by selecting a tip layer and layer value before editing the metadata. For an application to be customizable, customizations must be enabled in your project. For more information, see Developing a Customizable Application.

Before you begin:

It may be helpful to have an understanding of how customization works. For more information, see Customization and Layers: Use Cases and Examples, and Customizing an Application.

You may also find it helpful to understand additional customization functionality that can be added to your applications. For more information, see Additional Functionality for Customization.

You will also need to complete these tasks:

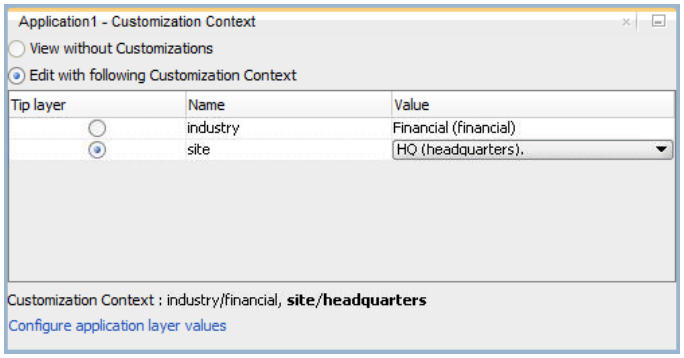
- Create your customization classes, as described in Creating a Customization Class.
- Create a JAR using the procedure described in Adding Customization Classes into a JAR.
- Make your classes available to JDeveloper, as described in Making Customization Classes Available to JDeveloper at Design Time.
- Update the `adf-config.xml` file, as described in How to Configure the adf-config.xml File .
- Enable seeded customizations in your application, as described in How to Enable Seeded Customizations for User Interface Projects, and How to Enable Seeded Customizations in Existing Pages.
- Configure the applications customization layers, as described in How to Configure Customization Layers.
- Launch JDeveloper using the **Customization Developer** role, and open the application that you want to customize.

To customize metadata in JDeveloper:

1. In the Customization Context window, select the layer and value for which you want to implement customizations.

The **Customization Context** (displayed at the bottom of the Customization Context window) changes to reflect your selection, as shown in Figure 51-4.

Figure 51-4 Customization Context Window with site/headquarters Selected as the Tip Layer



Application1 - Customization Context

○ View without Customizations

● Edit with following Customization Context

Tip layer	Name	Value
○	industry	Financial (financial)
●	site	HQ (headquarters).

Customization Context : industry/financial, **site/headquarters**

Configure application layer values

> **Note:**
>
> The selection you make in the Customization Context window indicates the context for the customizations that you will implement in JDeveloper. This selection does not directly impact the runtime for the application. At runtime, the customization context is returned from your customization classes. For more information, see How to Create Customization Classes.

2. Edit the metadata as you typically would during development. For example, double-click an entity object. Then edit the object using the overview editor.

   While you use the same techniques for editing metadata during customization that you would during development, certain restrictions apply. For example, some string properties, such as button labels, cannot be edited directly in the Properties window: they must be edited using the Select Text Resource dialog or the Expression Builder. For more information about restrictions to editing during customization, see Introducing the Customization Developer Role. For information about using the Expression Builder, see Opening the Expression Builder from the Properties Window.

   Even though you use the overview editor to implement customizations, you do not make changes to the base metadata file. Your changes are stored separately in a customization metadata file.

   > **Note:**
   >
   > To see the uncustomized base metadata, you can select **View without Customizations** in the Customization Context window.

3. You can optionally choose **Remove Customization** from the dropdown menu for a property (in the Properties window) to clear the existing customization.

> **Note:**
>
> In the Properties window, tip layer customizations are indicated by an orange icon, while properties that are not customized in the current tip layer are indicated by a green icon. A customization can only be removed in the context in which it was added. So you can remove only those customizations that have an orange indicator in the Properties window.

4. From the main menu, choose **File > Save** to save your changes.

After you have completed your customizations, you can run and test the customized application.

What Happens When You Customize an Application

When you implement customizations in an application, JDeveloper creates a metadata file for the customizations and a subpackage to store them in.

The metadata file contains the customizations for the customized object, which are applied over the base metadata at runtime. The new metadata file is named the same as the base file for the object with an additional `.xml` extension. For example, if you implement customizations for the `browseOrders.jsff` page, the customization metadata file is named `browseOrders.jsff.xml`. Or if you implement customizations on the `OrderItems` entity object, the base metadata file is named `OrderItems.xml` and the customization metadata file is named `OrderItems.xml.xml`.

The customization metadata files are stored in a subpackage hierarchy that is created at the same level as the object you customize. The first-level package is named `mdssys`, and it contains a package named `cust`. The `cust` package contains a package for each customization layer for which you have implemented customizations.

For example, say you have a base application that has a package called `oracle.summit.model.entities` containing your entity objects, and you have a customization layer named `site` with two layer values: `headquarters` and `remoteoffices`. Then you implement customizations for the `OrderItems` entity object at the `headquarters` layer value. When you implement these customizations, JDeveloper creates the subpackage hierarchy `oracle.summit.model.entities.mdssys.cust.site.headquarters` and stores the customization metadata files there.

Similarly, for pages in your user interface project, JDeveloper creates a directory structure to store the customization metadata files. For example, if you customize the `BrowseOrders.jsff` page in the Web Content node of your user interface project, JDeveloper creates the directory structure `mdssys/cust/site/headquarters` under Web Content and stores the customization metadata file there.

How to Customize ADF Library Artifacts in JDeveloper

In the **Customization Developer** role, you can use JDeveloper to customize artifacts in an ADF library. This need can arise when, for example, one development team produces task flows as part of a framework service and makes them available to other teams as an ADF library. Then another development team uses one of the task flows in a consuming application, and needs to fine-tune it to fit the requirements of the application.

Customizing an ADF Library Artifact

You can add an ADF library to your project in the **Customization Developer** role just as you would add it in the **Studio Developer** role. However, in the **Customization Developer** role, content from an ADF library appears as editable to allow you to implement customizations, whereas in the **Studio Developer** role it is read-only. For more information about working with ADF libraries, see Reusing Application Components .

Before you begin:

It may be helpful to have an understanding of how customization works. For more information, see Customization and Layers: Use Cases and Examples, and Customizing an Application.

You may also find it helpful to understand additional customization functionality that can be added to your applications. For more information, see Additional Functionality for Customization.

You will also need to complete these tasks:

- Create your customization classes, as described in Creating a Customization Class.
- Create a JAR using the procedure described in Adding Customization Classes into a JAR.
- Make your classes available to JDeveloper, as described in Making Customization Classes Available to JDeveloper at Design Time.
- Update the `adf-config.xml` file, as described in How to Configure the adf-config.xml File .
- Enable seeded customizations in your application, as described in How to Enable Seeded Customizations for User Interface Projects, and How to Enable Seeded Customizations in Existing Pages.
- Configure the applications customization layers, as described in How to Configure Customization Layers.
- Launch JDeveloper using the **Customization Developer** role, and open the application that you want to customize.

To customize an ADF library artifact:

1. In the Applications window, click the **Applications Window Options** icon and choose **Show Libraries**.

   This displays the libraries in the Applications window, so that you can explore them and access their artifacts.

2. Add the desired library to your project if it is not already shown among the libraries in the Applications window.

   For information about how to do this, see Adding ADF Library Components into Projects.

3. Customize the artifacts just as you would customize other content in your project.

For example, you can drag and drop taskflows from a library to `.jspx` or `.jsff` pages in a consuming project, drag and drop taskflows from a library to a page or fragment in another library, drag and drop library content or taskflows from the Resource catalog, drag and drop data controls from Data Controls panel to `.jspx` or `.jsff` pages in a library, edit business components, and drag and drop a data control from a library to the Data Controls panel and then drop to a page in another palette. All of these actions result in customizations of the library.

Specifying a Location for ADF Library Customizations

The location where ADF library customizations are stored is *`project-dir`*`\libraryCustomizations` by default. If your workspace contains multiple projects, you should change this to a workspace-level location for each project (for example, *`workspace-dir`*`\.mds\ADFLibraryCustomizations`).

You can change the location of ADF library customizations on the **Project Source Paths > ADF Library Customizations** page of the Project Properties dialog. If you change this location after you have implemented customizations on an ADF library, you must move the customization metadata files to the new location. To do this, use the file system to move the customization metadata (XML) files from the old directory to the new one.

If you have more than one project in the workspace with existing ADF library customizations that need to be moved to the common location, move the customizations to the new location one project at a time. For each project, change the location of ADF library customizations in the Project Properties dialog, and then move the customization metadata files from the old location to the new location. To mitigate conflicts where an ADF library artifact has customizations in more than one project, you have following options:

- From both projects, open the ADF Library artifact for which there are conflicting customizations, and decide upon which customizations you want. Preserve the customizations you want to keep and delete the others.
- If both customizations are important, open the ADF library artifact from the first project and implement the customizations that were previously done in second project. Then save the customizations in the first project and remove the customizations from the second project.

Do not open ADF library customization metadata files from multiple projects in a text editor and merge their contents. This can corrupt the customization metadata file.

Additionally, the location where ADF library customizations are stored is automatically included when you create a MAR deployment profile. If you change this location after you have created a MAR profile, you must also change the corresponding entry in the contributors list on the User Metadata file group page of the Edit MAR Deployment Profile Properties dialog before packaging. Alternatively, you can re-create the MAR profile to pick up this change. For more information about creating a MAR deployment profile, see How to Create Deployment Profiles.

How to View ADF Library Runtime Customizations from Exported JARs

When working in the JDeveloper Customization Developer role, you can view runtime customizations implemented on ADF library artifacts contained in an exported JAR.

Before you begin:

It may be helpful to have an understanding of how customization works. See Customization and Layers: Use Cases and Examples, and Customizing an Application.

You may also find it helpful to understand additional customization functionality that can be added to your applications. For more information, see Additional Functionality for Customization.

You will also need to complete these tasks:

- Create your customizable application, as described in Developing a Customizable Application.
- Deploy the application and implement runtime customizations on it, then export the runtime customizations to a JAR file.
- Launch JDeveloper using the **Customization Developer** role, and open the customizable application.

To make runtime customizations viewable from JDeveloper:

1. From the Applications window context menu, choose **Application Properties**.
2. In the Application Properties dialog, select **Customization Libraries**.
3. Click **Browse**, and navigate to the location of the exported JAR file.
4. Select the JAR file and click **Select**.
5. Click **OK**.

Now the JAR is available so that JDeveloper can look up customizations on ADF library artifacts. When you open an object that contains ADF library artifacts, JDeveloper looks for customizations in this JAR file and displays them if appropriate. JDeveloper decides what to display for a given artifact in a given customization context as follows:

- If the artifact has *neither* runtime customizations *nor* seeded customizations associated with it, the artifact from the ADF library is displayed.
- If the artifact has only *either* runtime customizations *or* seeded customizations associated with it (but not both), the customized artifact is displayed.
- If the artifact has *both* runtime customizations *and* seeded customizations associated with it, the seeded customization takes precedence and is displayed.

Additionally, when you run the application locally from JDeveloper, the runtime customizations are displayed. However, the runtime customizations are not included in any packaging of the application for deployment.

What Happens When You Customize ADF Library Artifacts

During the development of enterprise applications, there might be artifacts (such as task flows) that can be reused in multiple applications. To facilitate reuse of these common artifacts, they can be packaged into an ADF library and distributed. This allows you to add the ADF library to the list of libraries that the consuming application depends on. Then when the application is packaged, the customizations from all such ADF libraries are included in the MAR, which is later deployed to the MDS repository. Note, however, that ADF library customizations are added to the MAR deployment profile when you create the profile, which is not automatically updated when customizations are implemented after creating the profile. To pick up these subsequent customizations, use the **Update Profile** button on the MAR Options page of the Edit MAR Deployment Profile Properties dialog.

> **Note:**
>
> The ADF library provider should take care to ensure that no name conflicts arise due to customizations in the library. In the event that name conflicts arise between customizations packaged in an ADF library and the customizations from the consuming project, the customizations from the ADF library are ignored.

When you implement customizations on objects from an ADF library, the customization metadata is stored by default in a subdirectory of the project called `libraryCustomizations`. And although you create ADF library customizations at the project level, they are merged together during packaging to be available at the application level at runtime. Essentially, ADF libraries are JARs that are added at the project level, which map to library customizations being created at the project level. However, although projects map to web applications at runtime, the MAR (which contains the library customizations) is at the EAR level, so the library customizations are seen from all web applications.

Therefore, you can customize an ADF library artifact in only one place in an application for a given customization context (customization layer and layer value). Customizing the same library content in different projects for the same customization context would result in duplication in MAR packaging. To avoid duplicates that would cause packaging to fail, implement customizations for a given library in only one project in your application.

> **Note:**
>
> Using the same location for ADF library customizations for all of the projects in your application can help to avoid duplicate customization. For more information, see Specifying a Location for ADF Library Customizations.

For example, say the ADF library you are using contains a page fragment `text.jsff`. In the consuming application, customize this library page in only one project. By doing so, customizations are available for all projects in the application that consume this library at runtime.

You are also restricted from customizing an object from an ADF library when your project already contains an object with the same name. In case of duplication, you must fix the projects by deleting one of the duplicate documents or deleting one and manually merging the differences into the other.

Similarly, if the ADF library contains seeded customizations for an artifact within a given customization context (customization layer and layer value), you cannot implement customizations for that artifact within the same customization context. In this situation, the ADF library artifact is read-only. You can, however, implement customizations for the artifact within other customization contexts.

For example, say the ADF library you are using contains seeded customizations for the `Headquarters` layer value in the `Site` layer. When you select this as your tip layer in the Customization Context window, the customized objects in that ADF library are read-only. However, if you select `Site/Remote Site 1` as your tip layer, then the objects are customizable.

> **Note:**
>
> When the consuming application implements customizations on content from an ADF library, the customizations are written to the local project directories, but they are not automatically injected with the `web-app-root` during packaging. For more information, see Creating a MAR Deployment Profile.

What Happens at Runtime: How Customizations are Applied

At runtime, the application applies the customization metadata files over the base application in the order of precedence defined in the `cust-config` section of the `adf-config.xml` file.

The layer value is retrieved from the customization class at runtime and evaluated in the context the application is running, and the appropriate customizations for that layer value are applied.

What You May Need to Know About Customized Applications

When you are customizing an application, you might be using integrated source control or customizing resource strings. When you use these features, there is additional information you need to know.

Customization and Integrated Source Control

When working in the **Customization Developer** role, your source control integration complements the process of customization. If JDeveloper is configured to automatically check out and add new files to source control and you attempt to customize a base document that is available from a source control system, JDeveloper behaves in the following way:

- If the corresponding customization file is not already available, then a new customization file is created in source control and the customizations are written to it.
- If the corresponding customization file exists, it is checked out and customizations are written to it.
- If the corresponding customization file exists and it is already checked out or not yet in version control, customizations are written to it without any further version control operation.

Since the base document is not modified in the **Customization Developer** role, the base document is not checked out.

If JDeveloper is not configured to automatically check out or add new files to source control, you must manually make the customization files editable and check in newly created customization files to source control. For more information about using source control in JDeveloper, see section Using a Source Control System.

Editing Resource Bundles in Customized Applications

During the course of customizing your application, you might want to customize the content to use different resource bundle keys or define and use new resource keys.

You can open a customizable application in the **Customization Developer** role and use the Properties window to customize the usages of resource bundle strings. You can change a document to use another already existing resource key in a resource bundle, or create a new resource. For more information about resource bundles, see Working with Resource Bundles.

New resource keys (created in the **Customization Developer** role) are saved to an application-level override bundle (in XLIFF format), and JDeveloper adds an entry to the `adf-config.xml` file like the one shown in the following example to configure the application-level override bundle.

You must also configure the `adf-config.xml` file to support the overriding of the base resource bundle. As the following example shows, you must tag the `bundleId` element with `override="true"` to make it overrideable. After it is marked as overridden, customizations of that bundle are stored in the application's override bundle.

```
<adf-resourcebundle-config xmlns="http://xmlns.oracle.com/adf/resourcebundle/
config">
  <applicationBundleName>
    path-to-resource-bundle/bundle-name
  </applicationBundleName>
  <bundleList>
    <bundleId override="true">
      package.BundleID
    </bundleId>
  </bundleList>
</adf-resourcebundle-config>
```

> **Note:**
>
> If an application is not configured for customization, you can open it in the **Customization Developer** role and define new resource keys by choosing **Application > Edit Resource Bundles** from the main menu. However, you cannot change a document to use the new resource keys if it is not configured for customization.

How to Package and Deploy Customized Applications

You need to deploy your ADF application after you have made the necessary customizations. The customized metadata is packaged into a MAR file and made ready for deployment.

After you customize the application, you will want to deploy it. Before you deploy the customized application, you must follow the configuration procedures for setting up your MDS repository, as described in Managing the MDS Repository of the *Administering Oracle Fusion Middleware*.

An enterprise application can contain model and user interface projects, and both types of projects can contain customized metadata. The customized metadata is packaged into a MAR for deployment. By default, the customizations from both types of projects are added to a single MAR. For information about how to create a MAR profile, see Creating a MAR Deployment Profile.

Implicitly Creating a MAR Profile

When you use JDeveloper to package a Fusion web application, JDeveloper creates an auto-generated MAR profile that includes default metadata (such as customizations), when *either* of the following conditions are met.

- The **Enable User Customizations > Across Sessions using MDS** checkbox is selected on the ADF View settings page of the Project Properties dialog for the user interface project.
- The MDS configuration section of the `adf-config.xml` file contains a `<metadata-store-usage>` element that is marked as `deploy-target="true"`, as shown in the following example.

```
< . . . >
  <persistence-config>
    <metadata-namespaces>
      <namespace path="/oracle/apps" metadata-store-usage="repos1"/>
    </metadata-namespaces>
    <metadata-store-usages>
      <metadata-store-usage id="repos1" deploy-target="true">
      . . .
      </metadata-store-usage>
    </metadata-store-usages>
  </persistence-config>
< . . . >
```

Explicitly Creating a MAR Profile

For customizations created in JDeveloper to take effect in the application when it is deployed, these customizations need to be made available to the application at runtime. There are two techniques you can use to accomplish this:

- Package the customizations along with the application using a MAR profile.

  Create a MAR profile that includes the customization metadata. The MAR profile should be included in the deployed EAR file to ensure that the customizations are available at runtime. Your customization classes must be packaged in the EAR file such that they are in the application-level class loader.

> **Note:**
>
> If you have only seeded customizations, you do not need to create a MAR profile to import them into the MDS repository unless you also want to support runtime customizations. If you have seeded customizations and do not have cross-session persistence enabled, the seeded customizations will be packaged in the EAR file by default and loaded from the class path.

- Import the customizations to the runtime repository used by the application.

  You typically use this approach if customizations to library metadata need to be applied to an application that is deployed separately. Using this approach, you package the customizations into a JAR file and then use the command `importMetadata` with Oracle WebLogic Scripting Tool (WLST) to import them to the MDS runtime repository. For more information about this and other WLST commands, see the *WLST Command Reference for Infrastructure Components*.

If your application has customizations on objects from an ADF library, the customization metadata is implicitly included when you create the MAR profile. If you change the location of ADF library customizations in the Project Properties dialog, you must re-create the MAR profile before packaging.

If you plan to use design time at runtime capabilities to add or edit resource keys in the override bundle at runtime, the override bundle must be packaged as part of the MAR profile. By default, the override bundle is packaged as part of the auto-generated MAR profile if the application contains seeded customizations. However, if the application doesn't contain seeded customizations, you must explicitly create the MAR deployment profile to package the override bundle. When the MAR profile is created explicitly, the override bundle is added and included by default as part of the user metadata.

When you package and deploy the completed customized application, you should do so from the **Studio Developer** role, rather than from the **Customization Developer** role. For information about how to create a MAR profile, see How to Create Deployment Profiles.

Extended Metadata Properties

Extended metadata provides additional settings for customization of the Fusion web application. You can edit the extended metadata properties and identify who and which parts of metadata can be customized at runtime.

Extended metadata is data that describes the metadata content. The extended metadata file contains additional information about the metadata file. One use of this extended information is to identify which parts of the metadata can be customized at runtime (design time at runtime customizations) and who can customize them. For information about this use of extended metadata properties, see How to Enable Customization for Design Time at Runtime.

You can open a metadata file (such as a `.jspx` file) in JDeveloper and use the Properties window to view and edit its extended metadata properties. When you open a metadata file, its extended metadata properties are displayed in the Properties window. These properties can be edited to add metadata information at either of the following levels:

- File-level: These properties are displayed in the Properties window when the root element is selected in the Structure window.
- Element-level: These properties are displayed in the Properties window when an element is selected in the Structure window. The selected element should have a non-null identifier.

Extended metadata properties are supported for file types that support customizations and can be packaged in a MAR, such as `.jsff` and `.jspx` files.

Extended metadata for a metadata document is stored in an associated resource description framework (RDF) file. RDF is a W3C standard used to define an XML framework for defining metadata. The RDF file associated with the metadata document is created when the first property value is specified using the Properties window. Extended metadata properties are editable only when JDeveloper is in the **Studio Developer** role. RDF files are read-only in the **Customization Developer** role.

The RDF file is stored in the `mdssys` directory. For example, if the metadata being described is stored in the file system as `/myapp/data/page1.jspx`, the corresponding extended metadata document would be stored as `/myapp/data/mdssys/mdx/page1.jspx.rdf`. The extended metadata document must then be packaged with the corresponding metadata base file and added to the same deployment profile. For information about creating a MAR deployment profile, see How to Create Deployment Profiles.

> **Note:**
>
> Don't edit the extended metadata documents directly. Use the Properties window.

How to Edit Extended Metadata Properties

You can use extended metadata properties to provide additional metadata information that is not covered in the metadata file (such as a `.jspx` file). When you open the metadata file in JDeveloper, extended metadata properties are displayed in the Properties window, which you can use to edit these properties when you are using JDeveloper in the **Studio Developer** role.

For example, suppose you want to deliver a metadata file in some form to external customers. Along with metadata file, you need to provide additional information about the file, such as the creator, subject, description, format, and rights. You can accomplish this by creating an extended metadata property file for your metadata file.

Before you begin:

It may be helpful to have an understanding of how extended metadata is used. For more information, see Extended Metadata Properties.

You may also find it helpful to understand additional customization functionality that can be added to your applications. For more information, see Additional Functionality for Customization.

You will also need to complete this task:

- Launch JDeveloper using the **Studio Developer** role, and open the application.

To edit extended metadata properties:

1. In the Applications window, select the object for which you want to edit extended metadata properties.
2. In the Structure window, select the appropriate element (typically the root element).
3. In the Properties window, expand the appropriate node to edit the properties.

To display the Properties window with the values for the selected component, choose **Window > Properties** from the main menu.

4. Edit the value for the desired property and press Enter.
5. From the main menu, choose **File > Save** to save your changes.

If you have edited extended metadata properties for a metadata file, you must package your extended metadata (or RDF) files with the metadata files when you deploy the application.

How to Enable Customization for Design Time at Runtime

You can also use extended metadata properties to provide information about which parts of the metadata can be customized at runtime and who can customize the metadata content.

By default, all components that you can add to a `.jspx` page are preconfigured to allow runtime customization to support implicit user personalization (such as changing the order of columns in a table).

Components that are preconfigured to allow customization need no further modification to enable design time at runtime customizations. If you use them in a `.jspx` page, then they are customizable by default.

Depending on the requirements of your application, you will need to modify customization properties in the following situations:

- For components that are configured to allow customization, you can override the default settings to disallow customization.
- For components that are not configured to allow customization and metadata objects (such as `.jspx` pages), you can override the settings to allow customization.
- For components that are configured to allow customization, you can optionally restrict who is allowed to perform customizations at runtime.

If you edit customization properties, you must package your extended metadata (RDF) files with the metadata files when you deploy the application.

Additionally, you can restrict customizations on a entire subtree of components by marking the root of the subtree to disallow customizations. You can then mark individual components of that subtree as customizable to allow customizations for those specific components.

Editing Customization Properties in the Properties window

In the Customization group in the Properties window, there are two properties that allow you to specify whether customizations for an object are permitted at runtime and who is permitted to do them. The **CustomizationAllowed** property can be set on any element to specify whether or not it can be customized. The **CustomizationAllowedBy** property controls which users can customize the element. These settings are not enforced when you are implementing seeded customizations using JDeveloper, but are instead enforced when the application is customized at runtime (design time at runtime customizations).

For example, say you have a `.jspx` page with a form that contains two panels, and you need to allow runtime customization for content in Panel1, but not Panel2.

You would set **CustomizationAllowed** to `true` for Panel1, and set it to `false` for Panel2. If you need to allow runtime customization on an entire page, you would set **CustomizationAllowed** to `true` for the `.jspx` page root.

Before you begin:

It may be helpful to have an understanding of how extended metadata is used. For more information, see Extended Metadata Properties.

You may also find it helpful to understand additional customization functionality that can be added to your applications. For more information, see Additional Functionality for Customization.

You will also need to complete these tasks:

- Create your customizable application, as described in Developing a Customizable Application.
- Launch JDeveloper using the **Studio Developer** role, and open the application.

To edit customization properties:

1. In the Applications window, select the object for which you want to edit customization properties.
2. In the Structure window, select the appropriate element.
3. In the Properties window, expand the **Customization** node to edit the customization properties.

   To display the Properties window with the values for the selected component, choose **Window > Properties** from the main menu.
4. Edit the property value and press Enter.

   For example, to allow runtime customizations on a `.jspx` file, select the file and set the **CustomizationAllowed** property to `true`.
5. From the main menu, choose **File > Save** to save your changes.

Using a Standalone Annotations File to Specify Type-Level Customization Properties

You can optionally use a standalone annotations file to specify whether or not customizations are allowed for a given type of component. This file allows you to set customization properties on element types, rather than on instances. You can override the setting for a given type by explicitly setting `true` or `false` on a specific instance. The following example shows the contents of a sample standalone annotations file.

```
<?xml version="1.0" encoding="UTF-8"?>
<grammarMetadata xmlns="http://xmlns.oracle.com/bali/xml/metadata"
                 xmlns:md="http://xmlns.oracle.com/bali/xml/metadata"
                 xmlns:mmd="http://xmlns.oracle.com/bali/xml/metadata/model"
                 xmlns:mds="http://xmlns.oracle.com/mds"
                 namespace="http://www.oracle.com/mds/restrictCustomizations">
  <elementMetadata elementName="orderDetail">
    <attributeMetadata attributeName="itemTypeRef">
      <mds:customizationAllowedBy>sales admin</mds:customizationAllowedBy>
    </attributeMetadata>
    <attributeMetadata attributeName="description">
      <mds:customizationAllowed>true</mds:customizationAllowed>
      <mds:customizationAllowedBy>hr</mds:customizationAllowedBy>
    </attributeMetadata>
    <attributeMetadata attributeName="title">
      <mds:customizationAllowed>true</mds:customizationAllowed>
    </attributeMetadata>
  </elementMetadata>

  <elementMetadata elementName="bankTransfer">
    <mds:customizationAllowed>false</mds:customizationAllowed>
  </elementMetadata>

  <elementMetadata elementName="order">
    <mds:customizationAllowed>true</mds:customizationAllowed>
    <attributeMetadata attributeName="paymentRef">
      <mds:customizationAllowedBy>sales</mds:customizationAllowedBy>
    </attributeMetadata>
    <elementMetadata elementName="itemDetail">
      <attributeMetadata attributeName="itemsRef">
        <mds:customizationAllowedBy>hr</mds:customizationAllowedBy>
      </attributeMetadata>
    </elementMetadata>
  </elementMetadata>

</grammarMetadata>
```

If you use a standalone annotations file, you must register it with MDS in the `adf-config.xml` file. Create a `mdsc:standalone-definitions` section in the `mdsc:type-config` section of the `adf-config.xml` file, as shown in the following example.

```
<?xml version="1.0"?>
<adf-config xmlns="http://xmlns.oracle.com/adf/config"
            xmlns:mdsc="http://xmlns.oracle.com/mds/config">
 <mdsdata>
  <mdsc:mds-config version="11.1.1.000" >
   <mdsc:type-config>
     <mdsc:type-definitions>

       <mdsc:url>jar:file:/c:/jdev/lib/oafwk.jar!/oracle/apps/schemas/oa.xsd</mdsc:url>

       <mdsc:mds>/oracle/mds/TypeSeven.xml</mdsc:mds>
       <mdsc:mds>/oracle/mds/TypeEight.xml</mdsc:mds>
       <mdsc:file>typefile7.xsd</mdsc:file>
       <mdsc:file>typefile8.xsd</mdsc:file>
      </mdsc:type-definitions>
     <mdsc:standalone-definitions>

       <mdsc:url>jar:file:/c:/jdev/lib/oafwk.jar!/oracle/apps/schemas/
oa.xsd</mdsc:url>

       <mdsc:mds>/oracle/mds/standAloneSeven.xml</mdsc:mds>
       <mdsc:mds>/oracle/mds/standAloneEight.xml</mdsc:mds>
       <mdsc:file>standAloneFile7.xsd</mdsc:file>
       <mdsc:file>standAloneFile8.xsd</mdsc:file>
     </mdsc:standalone-definitions>
    </mdsc:type-config>
     ....
  </mdsc:mds-config>
 </mdsdata>
</adf-config>
```

Enabling Runtime Modification of Customization Configuration

For a given session, you can make modifications to the customized configuration at runtime.

You can prepare your customized application to accept overrides to the customization configuration (the `cust-config` section of the `adf-config.xml` file) at runtime on a per-session basis, thus allowing the user to change the way customizations are applied for a given session (or web request).

Consider a scenario where an application is configured with a `site` layer and a `user` layer and you want to make design time at runtime customizations to the `site` layer. If you use the application's customization configuration (defined in the `adf-config.xml` file), any customizations that you implement are applied to the `user` layer. So, you need to be able to adjust the customization configuration for a given session to allow your customizations to be applied to the `site` layer.

Or perhaps you have a requirement that an administrator wants to see the base metadata document with the `site` layer customizations alone. For cases like this, you need to specify a modified customization configuration, other than what was originally specified in application's `adf-config.xml` file.

For each web request, Oracle ADF creates an MDS session. For any MDS customization configuration modifications that apply to a session (web request), the user could programmatically provide modified MDS session options with a new customization configuration to Oracle ADF that would be applied on top of the original MDS configuration while creating a MDS session.

To implement this functionality, use the following ADF interfaces for `sessionOptionsFactory`:

- `oracle.adf.share.mds.SessionOptionsFactory` is the interface you use to specify modified MDS session options for a web request in a Fusion web application.

  ```
  SessionOptionsFactory :: oracle.mds.core.SessionOptions
  createSessionOptions(oracle.mds.core.SessionOptions defaultOptions)
  ```

  You implement this method to return the modified MDS `sessionOptions` to ADF.
- `oracle.adf.share.config.ConfigUtils` is the public class you can use to register your session options factory with ADF.

  ```
  ConfigUtils :: public static void setSessionOptionsFactory(ADFContext
  context, SessionOptionsFactory factory)
  ```

See your ADF API documentation for further detail on these interfaces.

You register your `sessionOptionsFactory` with Oracle ADF so that ADF can get modified session options from your implementation before the MDS session is created in request lifecycle.

The following example shows how you can implement `sessionOptionsFactory`. This example sets a modified customization configuration for the session to use the `site` customization layer alone regardless of the customization configuration specified in the `adf-config.xml` file. See the Javadoc for `oracle.mds` for more information.

```
package mycompany;

import oracle.adf.share.mds.SessionOptionsFactory;
import oracle.mds.config.CustClassListMapping;
import oracle.mds.config.CustConfig;
import oracle.mds.config.MDSConfigurationException;
import oracle.mds.core.SessionOptions;
import oracle.mds.cust.CustClassList;
import oracle.mds.cust.CustomizationClass;

public class MySessionOptionsFactory implements SessionOptionsFactory {
  public MySessionOptionsFactory() {
    super();
  }
  /**
   * Called to allow the application code to create a new SessionOptions object.
   * The application code should make sure to read the values from the
   * defaultOptions object as part of contruction of their new object and make
   * sure they only override the intended values.
   * @param defaultOptions
   * @return modified MDS session options
   */
  public SessionOptions createSessionOptions(SessionOptions defaultOptions) {
    // create new mds Customization configuration
    CustConfig custconfig = null;

    // create customization class array. Just put SiteCC implementation as we
    // wish to apply site customizations alone.
    CustomizationClass[] custclassarray = new CustomizationClass[] {new
SiteCC()};

    CustClassList custclasslist = new CustClassList(custclassarray);

    // specify the base metdata package namespace mapping on which site
    // customizations would apply
    CustClassListMapping[] mappings =
        new CustClassListMapping[] {new CustClassListMapping("mycompany/package",
                                                 null, null, custclasslist)};
    // create new customization configuration
    try{
      custconfig = new CustConfig(mappings);
    }
    catch (Exception ex){
      //do nothing
    }

    // now return modified sessionOptions to ADF with new mds customization
    // configuration. Only use newly created customization configuration in here.
    // For rest of option, use whatever available in defaultOptions.
    return new SessionOptions(defaultOptions.getIsolationLevel(),
                              defaultOptions.getLocale(),
                              custconfig,
                              defaultOptions.getVersionContext(),
                              defaultOptions.getVersionCreatorName(),
                              defaultOptions.getCustomizationPolicy(),
                              defaultOptions.getServletContextAsObject());
  }
}
```

52
Allowing User Customizations at Runtime

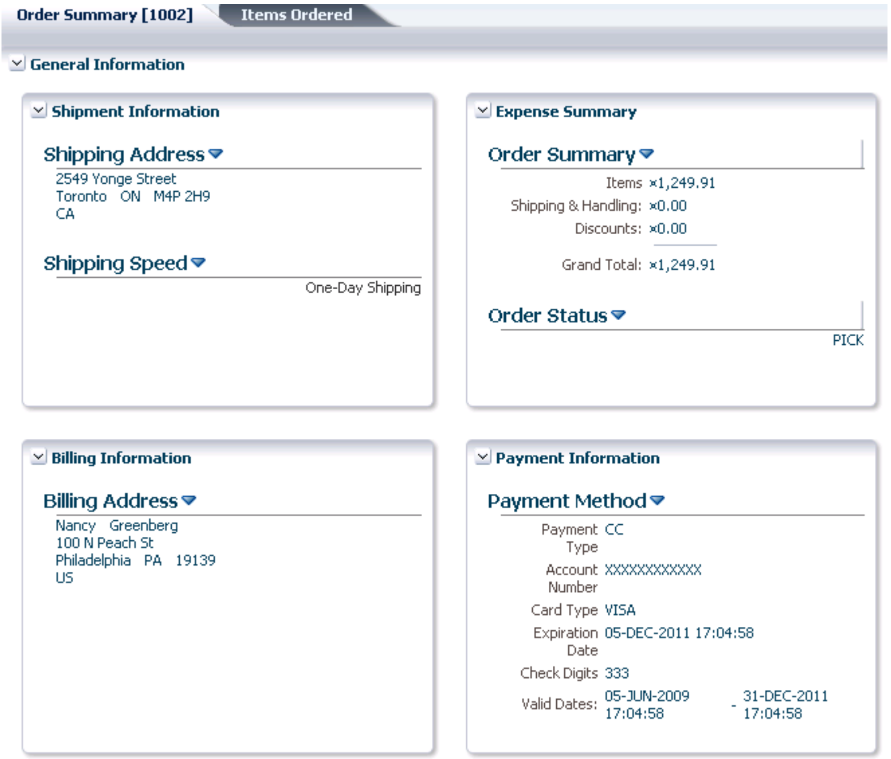
This chapter describes how to use the ADF Faces change persistence framework to create JSF pages that users can customize at runtime.
This chapter includes the following sections:

About User Customizations

JDeveloper allows you to save the attributes of ADF Faces components based on your specific requirements.

Certain ADF Faces components have attributes that can be saved for a specific user. For example, the value of the `disclosed` attribute on a `panelBox` component can be saved for a specific user during the current session. Say the `myOrders` page in an application contains four `panelBox` components that display order information. By default, they are expanded, as shown in Figure 52-1.

Figure 52-1 panelBox Components Are Expanded by Default



2549 Yonge Street
Toronto ON M4P 2H9
CA

One-Day Shipping

Items ¤1,249.91
Shipping & Handling: ¤0.00
Discounts: ¤0.00
Grand Total: ¤1,249.91

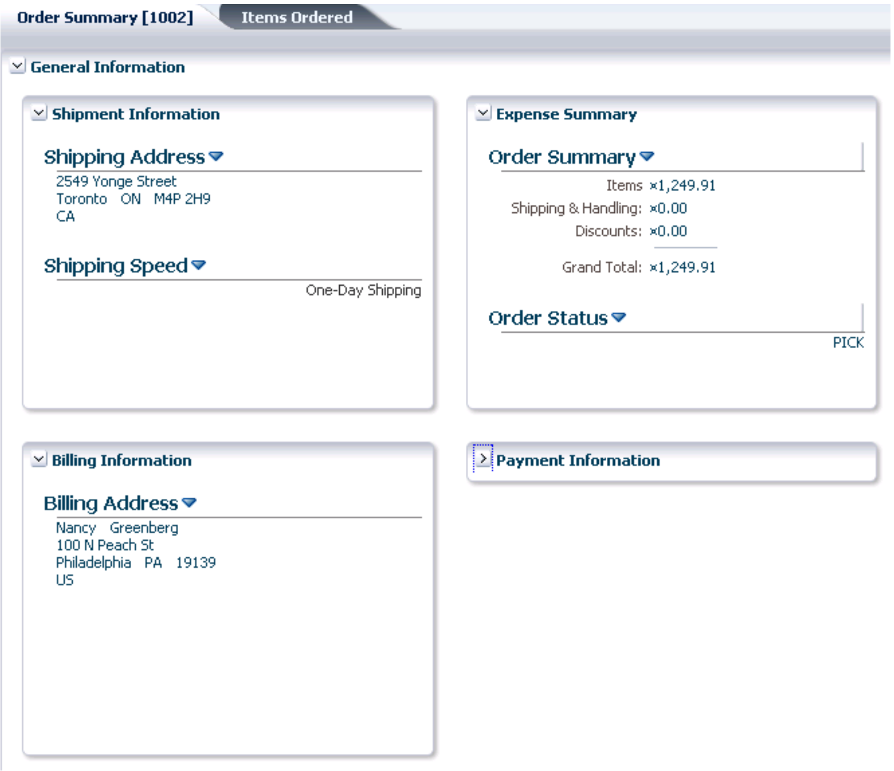
PICK

Nancy Greenberg
100 N Peach St
Philadelphia PA 19139
US

Payment Type CC
Account Number XXXXXXXXXXXX
Card Type VISA
Expiration Date 05-DEC-2011 17:04:58
Check Digits 333
Valid Dates: 05-JUN-2009 17:04:58 - 31-DEC-2011 17:04:58

However, suppose a user decides to collapse one of the boxes, as shown in Figure 52-2.

Figure 52-2 panelBox Component Remains Collapsed



2549 Yonge Street
Toronto ON M4P 2H9
CA

One-Day Shipping

Items ¤1,249.91
Shipping & Handling: ¤0.00
Discounts: ¤0.00
Grand Total: ¤1,249.91

PICK

Nancy Greenberg
100 N Peach St
Philadelphia PA 19139
US

Because this application is configured to allow user customizations, then during the user's session, anytime that user returns to the page, the Payment Information box remains collapsed. You need only to enable user customizations for the project in order for these changes to be persisted to the user's session.

> **Note:**
>
> The user session begins when the user logs in to the application, and ends when the user leaves the application. It is possible that while using an application, the user could navigate across application boundaries (for example, to a peer application) and thereby leave the application, at which point the user session would end.

Table 52-1 shows the attribute value changes persisted by an ADF Faces application, after you configure the application to allow user customizations.

Table 52-1 Implicitly Persisted Attribute Values

Component	Attribute	Affect at Runtime

`panelBox`
`showDetail`
`showDetailHeader`
`showDetailItem`
	`disclosed`	Users can display or hide content using an icon in the header. Detail content will either display or be hidden, based on the last action of the user.

`showDetailItem` (used in a `panelAccordion` component)	`flex`	The heights of multiple `showDetailItem` components are determined by their relative value of the flex attribute. The `showDetailItem` components with larger flex values will be taller than those with smaller values. Users can change these proportions, and the new values will be persisted.
`showDetailItem` (used in a `panelAccordion` component)	`inflexibleHeight`	Users can change the size of a panel, and that size will remain.
`panelSplitter`	`collapsed`	Users can collapse either side of the splitter. The collapsed state will remain as last configured by the user.
`panelSplitter`	`splitterPosition`	The position of the splitter in the panel will remain where last moved by user.
`richTextEditor`	`editMode`	The editor will display using the mode (either WYSIWYG or source) last selected by the user.
`calendar`	`activeDay`	The day considered active in the current display will remain the active day.
`calendar`	`view`	The view (day, week, month, or list) that currently displays activities will be retained.
`panelWindow` `dialog`	`contentHeight`	Users can change the height of a `panelWindow` or `dialog` popup component, and that height will remain.
`panelWindow` `dialog`	`contentWidth`	Users can change the width of a `panelWindow` or `dialog` popup component, and that width will remain.

`button`
`link`
`commandMenuItem`
`commandNavigationItem`
	`windowHeight`	When users change the `contentHeight` attribute value of a `panelWindow` or `dialog` component, any associated `windowHeight` value on a command component is also changed and will remain.

Table 52-1 (Cont.) Implicitly Persisted Attribute Values

Component	Attribute	Affect at Runtime
button link commandMenuItem commandNavigationItem	windowWidth	When users change the contentWidth attribute value of a panelWindow or dialog component, any associated windowWidth value on a command component is also changed and will remain.
column	displayIndex	ADF Faces columns can be reordered by the user at runtime. The displayIndex attribute determines the order of the columns. (By default, the value is set to -1 for each column, which means the columns will display in the same order as the data source). When a user moves a column, the value on each column is changed to reflect the new order. These new values will be persisted.
column	frozen	ADF Faces columns can be frozen so that they will not scroll. When a column's frozen attribute is set to true, all columns before that column (based on the displayIndex value) will not scroll. When you use the table with a panelCollection component, you can configure the table so that a button appears that allows the user to freeze a column.
column	noWrap	The content of the column will either wrap or not. You need to create code that allows the user to change this attribute value. For example, you might create a context menu that allows a user to toggle the value from true to false.
column	selected	The selected column is based on the column last selected by the user.
column	visible	The column will either be visible or not, based on the last action of the user. You will need to write code that allows the user to change this attribute value. For example, you might create a context menu that allows a user to toggle the value from true to false.
column	width	The width of the column will remain the same size as the user last set it.

Table 52-1 (Cont.) Implicitly Persisted Attribute Values

Component	Attribute	Affect at Runtime
`table`	`filterVisible`	ADF Faces tables can contain a component that allows users to filter the table rows by an attribute value. For a table that is configured to use a filter, the filter will either be visible or not, based on the last action of the user. You will need to write code that allows the user to change this attribute value. For example, you might create a button that allows a user to toggle the value from `true` to `false`.

`dvt:areaGraph`
`dvt:barGraph`
`dvt:bubbleGraph`
`dvt:comboGraph`
`dvt:horizontalBarGraph`
`dvt:lineGraph`
`dvt:scatterGraph`
	`timeRangeMode`	The time range for the data displayed on a graph time axis can be specified for all data visualization graph components. By default, all data is displayed. The time range can also be set for a relative time range from the last or first data point, or an explicit time range. You will need to write code that allows the user to change this attribute value. For example, you might create a dropdown list to choose the time range for a graph.

`dvt:ganttLegend`	`visible`	The legend for data visualization project, resource utilization, and scheduling Gantt chart components will either be visible or not inside the information panel. You will need to write code that allows the user to change this attribute value, for example, a hide and show button to display the legend.
`dvt:hierarchyViewer`	`layout`	The data visualization hierarchy viewer component supports nine hierarchy layout options including a top-to-bottom vertical, tree, circle, radial, and so on. Users can change the layout in the map control panel and the last selected layout will be retained.
`dvt:map`	`mapZoom`	This data visualization geographic map component attribute specifies the beginning zoom level of the map. The zoom levels are defined in the map cache instance as part of the base map. You will need to write code that allows the user to change this attribute value.

Table 52-1 (Cont.) Implicitly Persisted Attribute Values

Component	Attribute	Affect at Runtime
`dvt:map`	`srid`	This data visualization geographic map component attribute specifies the srid (spatial reference id) of all the coordinates of the map, which includes the center of the map, defined by startingX and startingY, and all the points in the point theme. You will need to write code that allows the user to change this attribute value.
`dvt:map`	`startingX`, `startingY`	This data visualization geographic map component attribute specifies the X and Y coordinate of the center of the map. The srid for the coordinate is specified in the srid attribute. If the srid attribute is not specified, this attribute assumes that its value is the longitude of the center of the map. You will need to write code that allows the user to change this attribute value.

`dvt:projectGantt`
`dvt:resourceUtilizationGantt`
`dvt:schedulingGantt`
	`splitterPosition`	The position of the splitter in the panel will remain where last moved by user.

`dvt:timeAxis`	`scale`	Data visualization components for project, resource utilization, and scheduling Gantt charts use this facet to specify the major and minor time axes in the Gantt chart. The time scale (`twoyears`, `year`, `halfyears`, `quarters`, `twomonths`, `months`, `weeks`, `twoweeks`, `days`, `sixhours`, `threehours`, `hours`, `halfhours`, `quarterhours`) can be set by the user using the menu bar **View** menu and the selection will be retained. Note that a custom time scale can also be named for this component value.
`dvt:timeSelector`	`explicitStart`, `explicitEnd`	Data visualization area, bar, combo, line, scatter, and bubble graph components use this child tag attribute to specify the explicit start and end dates for the time selector. Only value-binding is supported for this attribute. You will need to write code that allows the user to change this attribute value.

Runtime User Customization Use Cases and Examples

You can configure an application so that the value of the attributes listed in Table 52-1 can be persisted across sessions using the MDS repository. For example, if an application allowed persistence to the MDS repository, then a user could collapse the Payment Information box, as shown in Figure 52-2, and it would be collapsed the next time that user entered the application.

> **Note:**
>
> Before you can enable persistence to the repository, you must first follow all MDS configuration procedures as documented in Managing the MDS Repository of the *Administering Oracle Fusion Middleware*.

Along with the automatic persistence available through ADF Faces, you can create your own custom user customization capabilities for the following types of changes:

- Changing an attribute value
- Adding or removing a facet
- Adding or removing a child component
- Reordering child components
- Moving a child component to a different parent

If you want to create these types of custom user customizations, you need to add code (for example, in an event handler) that will call the APIs to handle the persistence.

Enabling an application to use the change persistence framework requires that you first enable your application to allow user customizations. Part of this process is determining where those changes should be persisted, either to the session or the MDS repository.

If you choose to persist changes to the session, by default all values as shown in Table 52-1 will be saved for the user's session. However if you choose to persist changes to a repository, you must explicitly configure which of these attribute values will be persisted to the repository. Instead of persisting all these attribute values, you can restrict changes so that only certain attribute value changes for a component are persisted, or so that only specific instances of the components persist changes.

> **Note:**
>
> You cannot persist changes to a component that is contained inside (anywhere in the subtree) of `af:forEach` or `af:iterator` tags using the `addComponentChange()` method. While such structure results in multiple copies of a component in the view tree, each component has only a single representation in the JSP document. However, the `addDocumentChange()` method does allow persistence for such components. See the discussion of the `DocumentChange` and `ComponentChange` classes in What You May Need to Know About the Change Persistence Framework API.

For any applications that persist changes to an MDS repository, when you deploy your application, you must create a metadata archive (MAR) profile in the application's EAR assembly. See How to Create Deployment Profiles.

Additional Functionality for Runtime User Customization

You may find it helpful to understand other features before you start working with user customizations. Following are links to other functionality that may be of interest.

- For information about the MDS architecture and metadata repositories (database- and file-based) and archives (EAR, MAR), see Managing the MDS Repository in *Administering Oracle Fusion Middleware*.
- The MDS framework allows you to create customizable applications that can be customized and subsequently deployed by a customer. See Customizing Applications with MDS .

Enabling Runtime User Customizations for a Fusion Web Application

In order to enable an ADF application to use the change persistence framework, you need to edit the web.xml and adf-config.xml files.

Enabling an application to allow user customizations (whether for the default changes that some ADF Faces components provide, or for custom capabilities that you create) requires that you configure your application to use the change persistence framework and that you also determine where those changes should be persisted (either the session or the MDS repository).

> **Note:**
>
> If you are planning on persisting changes to the MDS repository, before configuring an ADF Faces application to use change persistence, you must first follow all MDS configuration procedures as documented in Managing the MDS Repository of the *Administering Oracle Fusion Middleware*.

How to Enable User Customizations

You enable your application to use the change persistence framework by editing the `web.xml` and `adf-config.xml` files.

Before you begin:

It may be helpful to have an understanding of the features of runtime user customization. For more information, see Enabling Runtime User Customizations for a Fusion Web Application.

You may also find it helpful to understand additional customization functionality that can be added to your applications. For more information, see Additional Functionality for Runtime User Customization.

You will need to launch JDeveloper using the **Studio Developer** role, and open (or create) the application in which you want to enable runtime user customization.

To enable user customizations:

1. In the Applications window, double-click the web project in your application.
2. In the Project Properties dialog, select the **ADF View** node.
3. On the ADF View page, select the **Enable User Customizations** checkbox. If you want the changes to be persisted to only the session, select the **For Duration of Session** radio button. If you want the changes to persist to the MDS repository, select **Across Sessions Using MDS**.
4. If you choose to persist to the repository, you now need to declare each component tag and associated attribute values that you want persisted to the repository (if you choose to persist only to the session, all values will be persisted).

   For procedures to accomplish this task, see Configuring User Customizations. Once that configuration is complete, you can override those settings on a per component instance basis. For procedures, see Controlling User Customizations in Individual JSF Pages.

   > **Note:**
   >
   > If you have created custom user customization capabilities as documented in Implementing Custom User Customizations, then you also need to declare those attribute values or operations.

What Happens When You Enable User Customizations

When you elect to save changes only to the session, JDeveloper adds the CHANGE_PERSISTENCE context parameter to the web.xml file, and sets the value to session. This context parameter registers the ChangeManager class that will be used to handle persistence. If you instead elect to save the changes to the MDS repository, the value is set to oracle.adf.view.rich.change.FilteredPersistenceChangeManager, as shown in the following example.

```
<context-param>
  <param-name>org.apache.myfaces.trinidad.CHANGE_PERSISTENCE</param-name>
  <param-value>
    oracle.adf.view.rich.change.FilteredPersistenceChangeManager
  </param-value>
</context-param>
```

> **Tip:**
>
> If needed, you can manually set this value to oracle.adf.view.rich.change.MDSDocumentChangeManager if you do not want any customizations to be restricted based on configurations in the adf-config.xml file or on the individual JSF pages, and you always want the changes to be persisted to the MDS repository and not the session.

When you elect to persist to the repository, JDeveloper also does the following:

- Adds the following JARs to the class path if they don't exist:
  - `javatools-nodep.jar`
  - `facesconfigmodel.jar`
  - `taglib.jar`
- Adds another context parameter to `web.xml` to register the `MDSJSPProviderHelper` class to handle merging MDS customization documents with the base JSP document, as shown in the following example.

  ```
  <context-param>
    <param-name>oracle.adf.jsp.provider.0</param-name>
    <param-value>oracle.mds.jsp.MDSJSPProviderHelper</param-value>
  </context-param>
  ```

- Adds the ADF Faces Change Manager Runtime 11 library to the project.
- In the `adf-config.xml` descriptor file, sets the `persistent-change-manager` element to the `MDSDocumentChangeManager`, which is the class that will be used to persist the changes. The following example shows the configuration for persisting to the MDS repository.

  ```
  <persistent-change-manager>
    <persistent-change-manager-class>
      oracle.adf.view.rich.change.MDSDocumentChangeManager
    </persistent-change-manager-class>
  </persistent-change-manager>
  ```

- Creates JSF JSP pages as XML documents. See Controlling User Customizations in Individual JSF Pages.

Additionally, when you elect to save the changes to the MDS repository, and you include facelets in your application, JDeveloper adds the `faceletCache` context parameter to the `web.xml` file. The value of this parameter is set to `oracle.adfinternal.view.faces.facelets.rich.MDSFaceletCache`, as shown in the following example.

```
<context-param>
  <param-name>com.sun.faces.faceletCache</param-name>

  <param-value>oracle.adfinternal.view.faces.facelets.rich.MDSFaceletCache</param-value>

</context-param>
```

The facelets cache enhances the performance of your application and the cache capacity can be configured in MDS on the server. See Managing the MDS Repository in *Administering Oracle Fusion Middleware*.

Configuring User Customizations

You can configure your ADF application to use any type of change persistence (the session or a repository). By default, the attribute values are persisted to the user's session. You must declare the attributes values if you choose to persist the changes to the repository.

If you choose to persist changes to an MDS repository, you must decide which of the attribute values that are by default persisted to the session (as shown in Table 52-1) should also be persisted to the repository. Alternatively, you can configure which changes you do not want persisted.

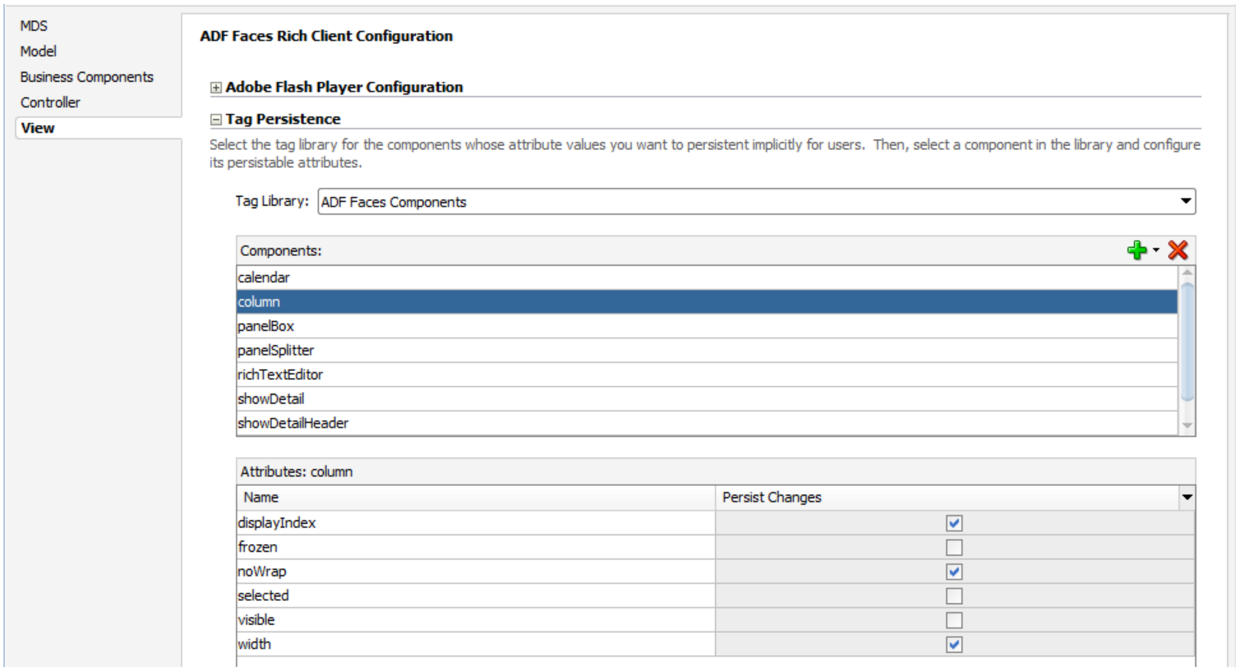
> **Tip:**
>
> Often, a system administrator is the one to set the configurations in the `adf-config.xml`. The `persist` and `dontPersist` attributes on a component allow page authors to override that setting as needed.

For example, suppose you decide that you don't want the value for the `width` attribute on columns to be persisted to the repository, but you do want all other default attribute changes for columns to be persisted. You must explicitly set the other default column values that you want to be persisted, and you also must explicitly configure the application to NOT persist the `width` attribute.

> **Note:**
>
> If you have created custom user customization capabilities as documented in Implementing Custom User Customizations, then you must explicitly declare those attribute values or operations as well.

You set (and unset) these values using the overview editor for the `adf-config.xml` file. Figure 52-3 shows the overview editor where only certain attribute values for the `column` component will be persisted.

Figure 52-3 Overview Editor for the adf-config.xml File



ADF Faces Rich Client Configuration

⊞ **Adobe Flash Player Configuration**

⊟ **Tag Persistence**

Select the tag library for the components whose attribute values you want to persistent implicitly for users. Then, select a component in the library and configure its persistable attributes.

Tag Library: ADF Faces Components

Components:
calendar
column
panelBox
panelSplitter
richTextEditor
showDetail
showDetailHeader

Attributes: column

Name	Persist Changes
displayIndex	☑
frozen	☐
noWrap	☑
selected	☐
visible	☐
width	☑

Once set, you can override persistence for a specific component on a page. For example, suppose you want to disallow change on the `width` attribute on only one table's columns. You want the rest of the tables in the application to persist changes to that attribute. You would configure the columns to globally persist changes to the `width` attribute, but then for that one table, you would override the global configuration directly on the JSF page. For more information, see Controlling User Customizations in Individual JSF Pages.

> **Note:**
>
> If you've enabled just session persistence, then all attribute values shown in Table 52-1 will be persisted to the session. There is no way to override this either globally or on an instance.

How to Configure Change Persistence

By default, when you configure your application to use any type of change persistence (that is, to either the session or a repository), the values of all attributes shown in Table 52-1 will always be persisted to the user's session. If you configured your changes to be persisted to a repository, then you must declare the attributes whose values should be persisted to that repository. If there are any values that you don't want persisted, then you need to configure those values as well.

Before you begin:

It may be helpful to have an understanding of how runtime user customization are persisted. For more information, see Configuring User Customizations.

You may also find it helpful to understand additional customization functionality that can be added to your applications. For more information, see Additional Functionality for Runtime User Customization.

You will need to launch JDeveloper using the **Studio Developer** role, and open (or create) the application in which you want to enable runtime user customization.

To declare attribute value persistence to a repository:

1. In the Application Resources panel, expand the **Descriptors** and **ADF META-INF** nodes, and then double-click **adf-config.xml**.
2. In the overview editor, click the **View** navigation tab.
3. In the View page, in the **Tag Persistence** section, in the **Components** table, select the component whose changes you want to persist (or not persist) to the repository.

   If the component does not appear in the table, click the **Add** icon to select and add it.

> **Note:**
>
> The filter rules specified in the `adf-config.xml` file are applicable only when you have chosen to persist to the MDS repository (see How to Enable User Customizations). These rules do not apply for persistence within session scope.
>
> If persistence fails for any reason (for example if one of the filter rules fails or there are MDS repository errors), then the values will be stored only within the session scope.

The **Attributes** table displays all the attributes for the selected component whose values can be persisted.

4. In the **Attributes** table, select **Persist Changes** for each of the attributes whose values you want persisted. Deselect any if you do not want the values persisted.

> **Note:**
>
> If you are implementing custom user customizations (see Implementing Custom User Customizations), then you will need to edit the `adf-config.xml` manually to add the configuration. See What Happens When You Configure Change Persistence for an example on how to configure user customizations.

What Happens When You Configure Change Persistence

When you select the component tags and attribute values to be persisted in the `adf-config.xml` file, JDeveloper enters tag library information for the components and attributes that are to be persisted. The following example shows the entry for persisting the value of the `disclosed` attribute on the `panelBox` component.

```
<taglib-config>
  <taglib uri="http://xmlns.oracle.com/adf/faces/rich">
    <tag name="panelBox">
      <attribute name="disclosed">
        <persist-changes>
          true
        </persist-changes>
      </attribute>
...
    </tag>
  </taglib>
</taglib-config>
```

Controlling User Customizations in Individual JSF Pages

Using an ADF Faces component’s persist and dontPersist attributes, you can override any globally set persistence configuration for a component.

Once you have enabled your application to allow user customizations, you can control user customizations for specific components on the page.

By default, the framework persists changes for all component instances, based on the configuration in the `adf-config.xml` file. You can override this default behavior by explicitly setting what should be persisted and what should not be persisted on each component instance using the `persist` and `dontPersist` attributes.

> **Note:**
>
> The filter rules specified using the `persist` and `dontPersist` attributes are applicable only when you have chosen to persist to the MDS repository (see How to Enable User Customizations). These rules do not apply for persistence within session scope.
>
> If persistence fails for any reason (for example if one of the filter rules fails or there are MDS repository errors), then the values will be stored only within the session scope.

The following components support the `persist` and `dontPersist` attributes:

- `panelBox`
- `showDetail`
- `showDetailHeader`
- `showDetailItem`
- `column`
- `tree`
- `treeTable`
- `panelSplitter`
- `calendar`
- `dvt:projectGantt`
- `dvt:resourceUtilizationGantt`
- `dvt:schedulingGantt`
- `dvt:ganttLegend`
- `dvt:hierarchyViewer`
- `dvt:timeAxis`

How to Control User Customizations on a JSF Page

You can override any globally set persistence configuration for a component using its `persist` and `dontPersist` attributes.

> **Tip:**
>
> Often, a system administrator is the one to set the configurations in the `adf-config.xml`. The `persist` and `dontPersist` attributes allow page authors to override that setting as needed.

Before you begin:

It may be helpful to have an understanding of how runtime user customizations can be restricted on an individual page. For more information, see Controlling User Customizations in Individual JSF Pages.

You may also find it helpful to understand additional customization functionality that can be added to your applications. For more information, see Additional Functionality for Runtime User Customization.

You will need to launch JDeveloper using the **Studio Developer** role, open the application, and open the page for which you want to modify user customization behavior.

To implement user customizations on a JSF Page:

1. In the Applications window, double-click the page that contains components that will be persisting changes.
2. If you want to persist all persistable attributes for a component:
   a. In the Properties window, expand the **Advanced** section.
   b. From the dropdown list for the **Persist** field, select **All Available**.
3. If you do not want to persist any attributes, repeat Step 2 for the **Don'tPersist** field.
4. If more than one attribute can be persisted for the component, and you do not want to persist all of them:
   a. In the Properties window, from the dropdown menu to the right of the **Persist** field, choose **Edit**.
   b. In the Edit Property dialog, shuttle any attributes to be persisted from **Available** to **Selected**.
5. If you do not want to persist an attribute value, repeat Step 4 for the **Don't Persist** field.

Note:

The filter rules specified using the `persist` and `dontPersist` attributes take precedence over any `adf-config.xml` configuration set for global component-level restrictions.

Values specified for the `dontPersist` attribute take precedence over values specified for the `persist` attribute. For example, if for a `panelBox` component you set `disclosed` as the value for both the `persist` and `dontPersist` attributes, the value of the `disclosed` attribute will not be persisted.

If you set the value of the `persist` or `dontPersist` attribute to `All Available`, then any values entered as choices using the Edit dialog and the shuttle will be ignored and all available attribute values will be persisted or not persisted.

What Happens at Runtime: How Changes are Persisted

When an application is configured to persist changes to the session, any changes made during the session are recorded in a session variable in a data structure that is indexed according to the view ID and the component's ID attribute value. Every time the page is requested, in the subsequent create View or Restore View phase, all changes are applied in the same order as they were added. This means that the changes registered through the session will be applied only during subsequent requests in the same session.

When an application is configured to persist changes to the MDS repository, any changes made during the session are recorded by mutating the Document Object Model that MDS maintains for the JSP document behind the view. A JSF phase listener registered by ADF controller triggers a commit on the MDS session during the appropriate lifecycle phase, resulting in the change document being persisted in the MDS store. Every time the page is requested, Oracle's JSP engine seeks the JSP document from an MDS JSP provider, which provides a flattened document after merging the stored changes to the base document. MDS records the change against the unique value of the component's ID attribute.

Tip:

If changes are applied in response to a partial submit of the page (for example, a `button` with the `partialSubmit` attribute set to `true`), the component for which changes are applied must be set as the value for the `partialTarget` attribute.

Additionally, be aware that when you run the application from JDeveloper in the Integrated WebLogic Server, MDS creates a local file-based repository to persist metadata customizations. In contrast, when the application is deployed to a test or production environment, customizations are persisted to the configured MDS repository. For more information about MDS repository configuration, see the *Administering Oracle Fusion Middleware*. For more information about deploying an application, see Deploying the Application.

What You May Need to Know About Using Change Persistence on Templates and Regions

How changes are persisted for components on templates or regions is handled differently, depending on whether the changes are persisted to the session or to the MDS repository. With session persistence, changes are recorded and restored on components against the `viewId` for the given session. As a result, when the change is applied on a component that belongs to a region or page template, that change is applicable only in scope of the page that uses the region or template. It does not span all pages that consume the region or template.For example, suppose you have `pageOne.jspx` and `pageTwo.jspx`, and they both contain the region defined in `region.jsff`, which in turn contains a `showDetail` component. When `pageOne.jspx` is rendered and the `disclosed` attribute on the `showDetail` component changes, the implicit attribute change is recorded and will be applied only for `pageOne.jspx`. If the user navigates to `pageTwo.jspx`, no attribute change is applied.

When you persist changes to the MDS repository, MDS records and restores customizations for a document identified by a combination of the JSP page path and customization name/value configuration setting as set on the customization class (for more information, see Customization Classes). As a result, for a given page that is rendered, when MDS applies a change on a component within a region or template, it is applicable for all pages that consume the region or template and that have the same customization name and value as the source page.

In the previous example, assume that the `showDetail` component uses the ID of `myShowDetail`. When `pageOne.jspx` is rendered and the `disclosed` attribute on the `showDetail` component changes, the attribute change is recorded for `region.jsff` (and not the page that consumes it). This change is applied when any page that contains the region is rendered, as long as the ID remains the same.

Implementing Custom User Customizations

Apart from the in-built user customization capabilities present in specific ADF Faces components, the change persistence framework allows you to create your own custom user customization capabilities.

In addition to the user customization capabilities built in to certain ADF Faces components, you can create your own custom user customization capabilities. The change persistence framework supports the following types of user customizations:

- Changing an attribute value
- Adding or removing a facet
- Adding or removing a child component
- Reordering child components
- Moving a child component to a different parent

To create custom user customizations, you must create a customization class for each type of user customization and then configure your application to use that class. You also need to set up the layers of customization for your application. For more information about both of these procedures, see Developing a Customizable Application.

Once those prerequisites are satisfied, you add logic that calls methods on the ADF Faces classes that handle persisting change either to the session or the MDS repository. To handle the change, you create code that uses the APIs from one of the ADF Faces specialized component change classes. For most cases, you add this code to the event handler method on a managed bean associated with the page the persisting component is on. If you want all instances of a component to persist the same change, you need to add this code for each page on which that component appears.

If you are creating a custom component, you can implement user customizations for the component by adding code directly to the custom component class. In that case, you will need to add the code only to the component class, and not once for each instance of the component. See Creating Implicit Change Persistence in Custom Components.

What You May Need to Know About the Change Persistence Framework API

To better understand what you need to do to create custom user customizations, it may help to have a deeper understanding of the change persistence and MDS frameworks. When you elect to persist changes to the MDS repository, the change persistence framework works in conjunction with the MDS framework. Where and how the customizations are saved are determined by how you set up your MDS repository, your customization layers, and your customization classes. Details about the MDS framework and the repository and how to use it are covered in Customizing Applications with MDS .

The change persistence framework uses the underlying change manager classes from Apache MyFaces Trinidad (in the `org.apache.myfaces.trinidad.change` package) along with a few ADF Faces-specific classes (in the `oracle.adf.view.rich.change` package). The instance of the registered `ChangeManager` class is accessible through the `RequestContext` object. It is responsible for gathering changes as they are created and added during a request, and then persisting them. The `SessionChangeManager` class is an implementation of `ChangeManager` which handles persistence within a session only, while the `MDSDocumentChangeManager` class is an implementation that persists to the MDS repository only. The `FilteredPersistenceChangeManager` class is an implementation of `ChangeManager` that stores the changes that pass the filter rules into the repository using the registered persistence change manager. Any change that does not get persisted to the repository will be persisted to the session when `FilteredPersistenceChangeManager` is used.

Additional classes are used to describe the changes to a component. You use these APIs to handle persisting any changes to components other than the implicit value changes the ADF Faces framework provides (as shown in Table 52-1). `ComponentChange` is the base class for all classes used to implement specific changes that act on the JSF component hierarchy, such as adding or removing a facet or a child component. These changes are automatically applied during subsequent creation of the view, in the same order in which they were added. Classes that extend the `ComponentChange` class and that also implement the `DocumentChange` interface can directly persist changes to the MDS repository. Classes that do not implement the `DocumentChange` interface can persist changes only to the session.

Table 52-2 describes the specialized classes that handle specific customizations. If "yes" appears in the Repository column, then the class implements the `DocumentChange` interface and it can persist changes to the MDS repository.

Table 52-2 Classes Used to Handle Change Persistence

Class Name	Repository	Description
`AddChildDocumentChange` `AddComplexChildDocumentChange`	Yes Yes	Adds a child or complex child (`dvt:`) component using document mark up. While applying this change, the child or complex child component is created and added to the document.
`AddComplexChildAttributeChange`	Yes	Adds a complex child (`dvt:`) component attribute.
`AttributeComponentChange` `ComplexAttributeComponentChange`	No Yes	Changes the value of an attribute.
`AttributeDocumentChange` `ComplexAttributeDocumentChange`	Yes Yes	Changes the value of a facet attribute.
`MoveChildComponentChange`	Yes	Moves a child from one container to another.
`RemoveChildComponentChange`	Yes	Removes a child component.
`RemoveComplexChildAttributeComponentChange` `RemoveComplexChildAttributeDocumentChange`	Yes Yes	Removes a complex child (`dvt:`) component attribute.
`SetFacetChildComponentChange`	No	Adds a child component to the facet using a document markup. While applying this change, the markup will be added to the document.
`SetFacetChildDocumentChange`	Yes	Adds a child component to a facet. While applying this change, the DOM element corresponding to the child component is added to the document. If the facet doesn't exist, it will be created. If the facet does exist, all of its content will be removed and the new content added.
`RemoveFacetComponentChange`	Yes	Removes a facet.
`ReorderChildrenComponentChange`	Yes	Reorders children of a component.

Aside from a ChangeManager class, you may also need to implement and register the `DocumentChangeFactory` interface with the `ChangeManager` class. If the `DocumentChangeFactory` implementation can provide an equivalent `DocumentChange` for a `ComponentChange`, the `ChangeManager` will use it to persist the `DocumentChange` to the repository.

How to Create Code for Custom User Customizations

You need to add code to handle any explicit changes you want to create, and to configure the components on the JSF page to handle customization. As with the default user customizations, you also must register the custom changes in the `adf-config.xml` file.

> **Note:**
>
> When the changes are expressible in more than one form, the change must be recorded in the form with highest precedence. For example:
>
> - Attribute change for a component: The attribute can be specified on the component tag or it can be expressed using the `<f:attribute>` tag. In a JSF JSP document, `<f:attribute>` takes lesser precedence over the attribute specified on the component tag. Therefore, the attribute change on the component tag will be recorded for customization.
> - Column header text in a column component: The header text for the column can be specified using either the `headerText` attribute or using header facet. In this case, the facet component will have precedence.

Before you begin:

It may be helpful to have an understanding of the features of custom runtime user customization. For more information, see Implementing Custom User Customizations.

You may also find it helpful to understand additional customization functionality that can be added to your applications. For more information, see Additional Functionality for Runtime User Customization.

You will need to launch JDeveloper using the **Studio Developer** role, open the application, and open the page for which you want to modify user customization behavior.

To create custom user customizations:

1. Create a managed bean for the page that contains the component, as described in Using a Managed Bean in a Fusion Web Application.

   For more information regarding managed beans and how they are used as backing beans for JSF pages, see the "Creating and Using Managed Beans" section in *Developing Web User Interfaces with Oracle ADF Faces*.

2. Add code to the event handler method for the component that will be used to make the change. This code should obtain the component that contains the change. It should then use the component and the appropriate APIs to create, record, and persist the change.

   The following example shows the code on the action event handler for a command button for a change that will be persisted to the MDS repository. When a user clicks the button, that source graphic file changes. The event handler method accesses the component and changes the source attribute for the graphic. It then calls the private `addAttributeChange` method, which first uses the component API to record the change, and then uses the `AttributeComponentChange` class to set the new source attribute value.

   ```
   public void modifyObjectImage(ActionEvent event) {
          UIComponent uic = event.getComponent().findComponent("oi1");
          String source = "/images/mediumAd.gif";
          uic.getAttributes().put("source", source);
          _addAttributeChange(uic, "source", source);
       }
   . . .
   private static void _addAttributeChange(UIComponent uic, String attribName,
   ```

```
                                                     Object attribValue) {
        FacesContext fc = FacesContext.getCurrentInstance();
        ChangeManager cm =
             RequestContext.getCurrentInstance().getChangeManager();
        ComponentChange cc =
             new AttributeComponentChange(attribName, attribValue);
        cm.addComponentChange(fc, uic, cc);
     }
```

> **Note:**
>
> When you persist changes, in addition to explicitly recording a change on the component (which is done in the example above using `uic.getAttributes().put("source", source)` method), you must also directly apply the change using the component API, as was done using the private `_addAttributeChange(uic, "source", source)` method. Applying the change in this way allows the user to see the change in response to the same request. If the change is recorded on the component, then the change will not be seen until a subsequent request.
>
> Additionally, if you know that the component will always persist to the repository regardless of any restricted change persistence settings, you can instead call the `AdfFacesContext.getCurrentInstance().getPersistentChangeManager()` method.

3. The `ChangeManager` class provides support for automatically converting an `AttributeComponentChange` into an `AttributeDocumentChange`, thereby allowing persistence to a repository. However, if you need to convert another type of change and you use a specialized change manager class that does not implement the `DocumentChange` class, you need to create a custom `DocumentFactory` implementation that converts the component change to a document change.

> **Note:**
>
> Automatic conversion of `AttributeComponentChange` into an `AttributeDocumentChange` assumes that the component attribute is represented as an attribute of the same name on the associated element in the JSPX document.
>
> Only those attribute values that are expressible in the JSPX document can be persisted using `AttributeDocumentChange`. In other words, `CharSequence`, `Number`, `Boolean` and `ValueExpression` are the only supported data types.
>
> Only values that implement `java.io.Serializable` can be persisted using `AttributeComponentChange`.

4. If you create a custom `DocumentFactory` implementation, you need to register it with the appropriate change manager class using the following method in your bean:

```
public static void registerDocumentFactory(String targetClassName,
                                           String converterClassName)
```

Where targetClassName is the name of the ComponentChange class and converterClassName is the name of your DocumentChangeFactory extension that is capable of converting the target ComponentChange into a DocumentChange. The semantics of name for these classes is same as that of getName() in the java.lang.Class class.

5. If the class you use to create the component change adds a child that has a subtree of components, and you want to persist the changes to the repository, you must create a DocumentFragment to represent the change.

   The following example shows how to use the AddComponentDocumentChange specialized class to create a DocumentChange object and use a DocumentFragment to represent the change.

```
public void appendChildToDocument(ActionEvent event)
{
  UIComponent eventSource = event.getComponent();
  UIComponent uic = eventSource.findComponent("pg1");
  // only allow the image to be added once
  if (_findChildById(uic,"oi3") != null)
    return;
  FacesContext fc = FacesContext.getCurrentInstance();
  DocumentFragment imageFragment = _createDocumentFragment(_IMAGE_MARK_UP);
  DocumentChange change = new AddChildDocumentChange(imageFragment);
  ChangeManager apm = RequestContext.getCurrentInstance().getChangeManager();
  apm.addDocumentChange(fc, uic, change);
}
 private static final String _IMAGE_MARK_UP =
 "<af:objectImage id='oi3' height='100' width='120' " +
     "source='http://www.somewhere.com/someimage.jpg' " +
     "xmlns:af='http://xmlns.oracle.com/adf/faces'/>";

private static DocumentFragment _createDocumentFragment(
    String markUp)
{
 // prepend XML declaration
  markUp = "<?xml version = '1.0' encoding = 'ISO-8859-1'?>" + markUp;
  DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
  factory.setNamespaceAware(true);
  factory.setValidating(false);
  DocumentBuilder builder;
  try
  {
    builder = factory.newDocumentBuilder();
  }
  catch (ParserConfigurationException pce)
  {
    _LOG.log(Level.WARNING, "Unable to get XML Parser:", pce);
     return null;
  }
  try
  {
    // use a version explicitly with ISO-8859-1 instead
    byte[] markupBytes = markUp.getBytes();
    Document newDoc = builder.parse(new ByteArrayInputStream(markupBytes));
    DocumentFragment fragment = newDoc.createDocumentFragment();
    // add the document's root element to the fragment
    fragment.appendChild(newDoc.getDocumentElement());
    return fragment;
  }
  catch (SAXException se)
  {
    _LOG.log(Level.WARNING, "Unable to parse markup:" + markUp, se);
    return null;
  }
  catch (IOException ioe)
  {
    _LOG.log(Level.WARNING, "IO Problem with markup:" + markUp, ioe);
    return null;
  }
}
```

6. Register the user customizations in the `adf-config.xml` file, as documented in Configuring User Customizations. If the custom changes are of any type other than `AttributeDocumentChange`, you will need to manually edit the `adf-config.xml` file and indicate that all changes are allowed for the component, as shown in the following example.

   ```
   <tag name="inputText">
     <attribute name="label">
       <persist-changes>true</persist-changes>
     </attribute>
     <persist-operations>ALL</persist-operations>
   </tag>
   ```

Creating Implicit Change Persistence in Custom Components

Changes are persisted as component changes in the session. Attribute changes are handled by ADF Faces runtime code. Implicit change persistence applies to component specific list of attributes.

When you create a custom component, you may decide that you want certain attribute values on that component to be persisted whenever change persistence is enabled in an application. Setting implicit change on a custom component is similar to setting explicit change persistence on existing components. You add code that executes the actual persistence, but instead of you placing that code on a managed bean, that code can be handled directly by the component class. If your component's attribute values are synchronized with the server using events, then you can use the broadcast method to persist the changes. If the attribute value that you want to persist does not use events, then you need to add code in the renderer and component class.

How to Set Implicit Change Persistence For Attribute Values that Use Events

When an attribute value uses events, you need to add code to the component class.

Before you begin:

It may be helpful to have an understanding of how persistence is implemented in custom components. For more information, see Creating Implicit Change Persistence in Custom Components.

You may also find it helpful to understand additional customization functionality that can be added to your applications. For more information, see Additional Functionality for Runtime User Customization.

You will need to launch JDeveloper using the **Studio Developer** role, and open the application.

To set implicit change persistence for attribute values that use events:

1. In the Applications window, double-click the custom component class Java file.
2. Add code to the broadcast method that will use the specialized class to create a new ComponentChange object and then call the ChangeManager to add the change.

   The following example shows the code added to the UIXShowDetail class that persists a change to the disclosed attribute. In this case, the AttributeComponentChange class is used.

```
public class UIXShowDetail extends UIXComponentBase
{
  ...
  public void broadcast(FacesEvent event) throws AbortProcessingException
  {
    super.broadcast(event);
    ...
    if (event instanceof DisclosureEvent)
    {
      boolean isDisclosed = ((DisclosureEvent) event).isExpanded();
      setDisclosed(isDisclosed);
      //Record a Change for 'disclosed' attribute
      AttributeComponentChange aa =
       new AttributeComponentChange('disclosed', isDisclosed ?
Boolean.TRUE : Boolean.FALSE);
      AdfFacesContext adfContext = AdfFacesContext.getCurrentInstance();
      adfContext.getChangeManager().addComponentChange(getFacesContext(),
this, aa);
      ...
    }
  }
  ...
```

How to Set Implicit Change Persistence For Other Attribute Values

When an attribute does not use events, you need to place code in the component's renderer class.

Before you begin:

It may be helpful to have an understanding of how persistence is implemented in custom components. For more information, see Creating Implicit Change Persistence in Custom Components.

You may also find it helpful to understand additional customization functionality that can be added to your applications. For more information, see Additional Functionality for Runtime User Customization.

You will need to launch JDeveloper using the **Studio Developer** role, and open the application.

To set implicit change persistence for other attribute values:

1. In the Applications window, double-click the custom component's render class Java file.

2. Use the `findTypeConstants` method, which takes a `ClientMetadata` instance and use the `addPersistedProperty` method to mark certain properties as persisted. The following example shows code from the renderer class used for the ADF Faces `PanelSplitter` component, which implicitly persists the `splitterPosition` attribute value.

```
// Code from PanelSplitterRenderer.java
protected void findTypeConstants(
    FacesBean.Type type,
    ClientMetadata metadata)
  {
    super.findTypeConstants(type, metadata);
    metadata.addRequiredProperty(
      _orientationKey = type.findKey("orientation"));
    metadata.addRequiredProperty(
      _positionedFromEndKey = type.findKey("positionedFromEnd"));
    metadata.addRequiredProperty(
      _disabledKey = type.findKey("disabled"));
    metadata.addRequiredProperty(
      _splitterPositionKey = type.findKey("splitterPosition"));
    metadata.addPersistedProperty(_splitterPositionKey);
  }
```

3. In the JavaScript component peer class, define the attribute value to be persisted, using the `setProperty` function. This function needs to be invoked with the attribute name (as defined in the renderer in the previous step), the value, and "`true`", meaning the value of the attribute will be set. The following example shows code from the `panelSplitter` class that sets the splitter position.

```
// Code from AdfDhtmlPanelSplitterPeer.js file where we set the
   splitter position

  var component = this.getComponent();
 component.setProperty("splitterPosition",position, true);
// position is the value to be set
```

53 Using the Active Data Service

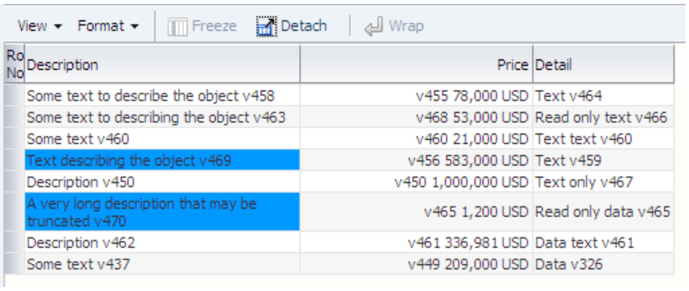
This chapter describes how to work with the ADF Model layer and Active Data Service (ADS) to provide real-time updates to ADF Faces components.
This chapter includes the following sections:

About the Active Data Service

ADF Faces provides a mechanism called Active Data Service (ADS) that allows us push changes from the server to the client. Through ADS, you can inform the browser about the fact that changes are available.

The Fusion technology stack includes Active Data Service (ADS), which is a server-side push framework that allows you to provide real-time data updates for ADF Faces components. You bind ADF Faces components to a data source and ADS pushes the data updates to the browser client without requiring the browser client to explicitly request it. For example, you may have a table bound to attributes of an ADF data control whose values change on the server periodically, and you want the updated values to display in the table. You can configure your application and the component so that whenever the data changes on the server, the ADF Model layer notifies the component and the component rerenders the changed data with the new value highlighted, as shown in Figure 53-1.

Figure 53-1 Table Displays Updated Data as Highlighted



Ro No	Description	Price	Detail
	Some text to describe the object v458	v455 78,000 USD	Text v464
	Some text to describing the object v463	v468 53,000 USD	Read only text v466
	Some text v460	v460 21,000 USD	Text text v460
	Text describing the object v469	v456 583,000 USD	Text v459
	Description v450	v450 1,000,000 USD	Text only v467
	A very long description that may be truncated v470	v465 1,200 USD	Read only data v465
	Description v462	v461 336,981 USD	Data text v461
	Some text v437	v449 209,000 USD	Data v326

Active Data Service Use Cases and Examples

Using ADS is an alternative to using automatic partial page rendering (PPR) to rerender data that changes on the backend as a result of business logic associated with the ADF data control bound to the ADF Faces component. Whereas automatic PPR requires sending a request to the server (typically initiated by the user), ADS enables changed data to be pushed from the data store as the data arrives on the server. Also, in contrast to PPR, ADS makes it possible for the component to rerender only the changed data instead of the entire component. This makes ADS ideal for situations where the application needs to react to data that changes periodically.

To use this functionality, you must configure the application to use ADS. If your application services do not support ADS, then you also need to create a proxy of the service so that the components can display the data as it updates in the source.

Any ADF Faces page can use ADS. However, you can configure only the following ADF Faces components and ADF Data Visualization (DVT) components to work with active data:

- `activeImage`
- `activeOutputText`
- `chart` (all types)
- `gauge` (all types)
- `pivotTable`
- `tree`
- `treeTable`
- `geoMap` (`mapPointTheme` only)
- `sunburst`
- `treemap`

For specific information about ADS support for DVT components, see Active Data Support in *Developing Web User Interfaces with Oracle ADF Faces*.

Additionally, note that collection-based components (such as `table`, `tree`, and `pivotTable`) support ADS only when the `outputText` component or `sparkChart` is configured to display the active data; other components are not supported inside the collection-based component.

Limitations of the Active Data Service Framework

The framework for ADS has the following limitations.

- ADS does not support active data on ADF Faces table components with filtering enabled. Once a table is filtered at runtime, active data cannot be displayed.
- ADS does not time out from user inactivity by default. To ensure that an active session is not maintained indefinitely, you can configure the `web.xml` context-parameter `oracle.adf.view.rich.poll.TIMEOUT` to specify how long ADS should run before it times out from user inactivity. For more information, see How to Configure Session Timeout for Active Data Service.

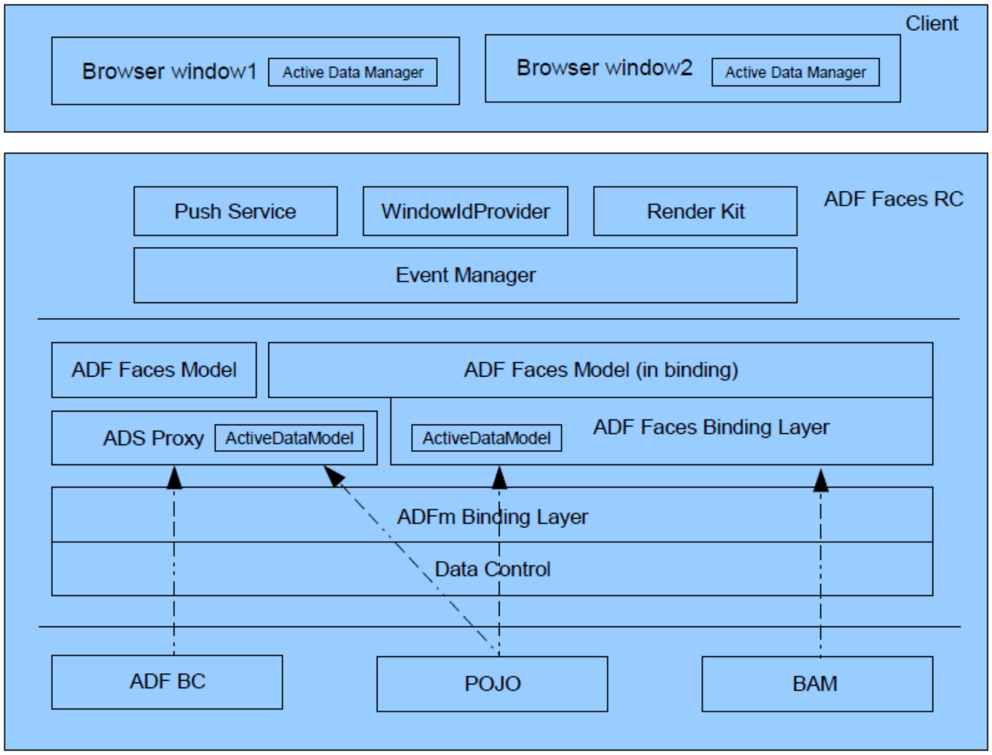
- ADS does not restart if the user attempts to navigate away from the browser page and then chooses to stay on the page by canceling the exit action in the browser-displayed warning dialog. To workaround this limitation, ADF Faces provides client listener support to handle `beforeunload` events for browsers that support displaying a Confirm on Page Exit dialog. The ADF document that handles this event will allow the user to cancel the exit before ADS stops processing.
  For more information, see What You May Need to Know About Navigating Away From the ADS Enabled Page in *Developing Web User Interfaces with Oracle ADF Faces*.
- ADS executes on a non-request thread and does not support the forced execution of component-level EL expressions. Therefore EL expressions on ADF Faces components that you may use to calculate component properties (such as fill color in a chart configured to use active data) will not be executed when the component value is updated by ADS. To execute EL in this scenario, you should calculate the property value in the model and use EL to bind the component property to the model. Alternatively, your page may apply JavaScript inline to apply the property value change after the active data update occurs. This latter solution effectively forces ADF Faces to rerender the component.

> **Note:**
>
> The ADF DVT components may impose additional limitations. For details, see Active Data Support in *Developing Web User Interfaces with Oracle ADF Faces*.

Active Data Service Framework

The framework for ADS contains a number of components that work together to send the active data from the source to the UI component. When a data event occurs, if the associated ADF Model layer binding is configured for active data, the Active Data model delivers the data to the Event Manager. The Event Manager then retrieves the data and invokes the Push Service, which delivers the data to the correct component, based on how the service is configured (for more information, see Data Transport Modes). The component then applies the new data pushed from the server. Figure 53-2 shows the ADS framework.

Figure 53-2 Active Data Service Framework



In order to use the Active Data Service, you need to have a data store that publishes events when data is changed, and you need to create business services that react to those events. By default, **ADF Business Components** does not react to data events. The Active Data Proxy framework allows all types of data sources, including ADF Business Components, to work with ADS. It combines the `ActiveDataModel` with the JSF model, so that you need to override functionality only on this proxy rather than on both the `ActiveDataModel` and the JSF model.

The following comprise the ADS framework:

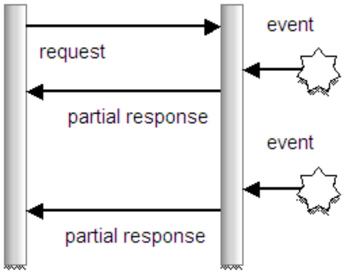
- `ActiveDataModel` interface: Abstraction of the active data model. Its responsibilities include:
  - Starting and stopping active data
  - Keeping track of the current active data event ID
  - Letting the renderer know whether the model needs active data or not.
- Event Manager: A server-side component that works with the ADF Model layer. It is responsible for the following:
  - Listening to binding events
  - Retrieving active data
  - Managing active data encoding
  - Invoking the Push service to send the encoded active data

- Push service: A delivery channel that interacts with the Event Manager on the server side and with the Active Data Manager on the client side. It provides the following:
  - Establishing and maintaining the connection between the server and the client
  - Transmitting the active data over this connection from the server to the client
  - Ensuring that active data gets delivered within desired parameters and forcing component update if not
- Active Data Manager: A client-side component that distributes the active data to the correct component. Specifically, it is responsible for the following;
  - Delivering events from the server side that are coming through the channel, using an event delivery service
  - Handling multiple browser windows through a shared channel
  - Dispatching active data events to rich client components, so that the components can render the change accordingly
- Active Data Proxy: A proxy that allows all types of data sources to enable push functionality. Specifically, the proxy is responsible for the following:
  - Implementing and delegating `ActiveDataModel` functionality
  - Delegating to JSF models
  - Listening to data change events from the data source
  - Generating active data events based on the data change events

Data Transport Modes

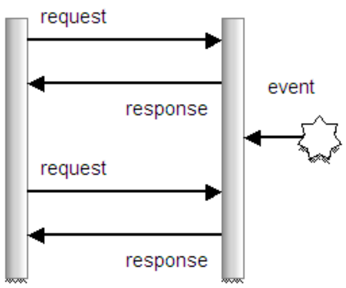
Active data is sent to the client using data streaming (push) or one of two types of data polling. With data streaming, there is only one request, which stays open. When a data change event occurs, a partial response is sent (the response is not closed), the client is notified, and the associated component is updated to show the new data, as shown in Figure 53-3.

Figure 53-3 Streaming Mode



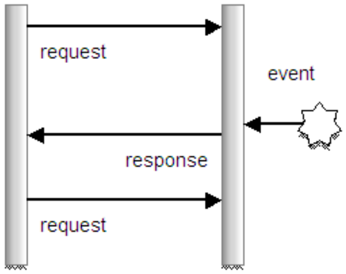
With data polling, the application is configured to poll the data source at specified intervals, as shown in Figure 53-4. With each request, a response is sent and closed, whether or not a data change event has occurred. If the data has changed, then the client is notified and the component is updated.

Figure 53-4   Poll Mode



Long polling is similar to streaming. When the page is rendered, a request is sent to the active channel. However, a response is not returned until there is a data change event. At that point, the connection is closed. As soon as the connection is closed, a new connection is initiated, which results in the connection being active most of the time: there are no specific intervals. Long polling results in the majority of data change events being received when they occur, because the connection is already established and ready to send a response to the client, as shown in Figure 53-5. See What You May Need to Know About Configuring an ADS Transport Mode.

Figure 53-5   Long Polling Mode



To use ADS, you need to configure your application to determine the method of data transport, as well as other performance options.You also need to configure the bindings for your components so that they can use ADS. If you are using ADF Business Components, you need to modify your model to implement the `ActiveModel` interface to register itself as the listener for active data events using the Active Data Proxy.

Configuring the Active Data Service

The adf-config.xml file is used to configure ADS. You can configure your ADF application and the component so that whenever the data changes on the server, the component is notified and rerenders with the new value highlighted. ADS can use any of the three modes to determine active data to the component: streaming, polling, or long polling.

You need to configure ADS to determine the data transport mode, as well as to set other configurations, such as a latency threshold and reconnect information.

> **Note:**
>
> If you enable ADS but do not configure it, the ADS framework will use the default values shown in Table 53-1.

How to Configure the Active Data Service

Configuration for running the Fusion web application with ADS in JDeveloper and Integrated WebLogic Server is done in the `adf-config.xml` file. For information about the `adf-config.xml` file, including how to create one if you need to, see the Configuration in adf-config.xml section of *Developing Web User Interfaces with Oracle ADF Faces*.

> **Note:**
>
> To support automatic replication and failover for web applications within a clustered environment, Oracle WebLogic Server supports mechanisms for replicating HTTP session state across clusters. You can configure **Oracle Application Development Framework** (Oracle ADF) to ensure the Fusion web application's state can be restored from any server in the cluster. To support failover for pages configured to display active data, it is necessary to enable failover for the ADF Business Components **application module** and to enable ADF Controller to track changes to the ADF memory scopes. With these ADF settings configured, when failover occurs, the ADF server will detect the failover and request all pages configured to display active data to refresh themselves, after that ADS restarts and data is pushed to the client. For details about how to enable failover for the Fusion web application in a high availability environment, see Configuring Oracle ADF for High Availability.

Before you begin:

It may be helpful to have an understanding of the ADS configuration choices. See Configuring the Active Data Service and What You May Need to Know About Configuring an ADS Transport Mode.

You may also find it helpful to understand functionality that can be added using other **Oracle ADF** features. See About the Active Data Service.

To configure the Active Data Service:

1. In the Application Resources panel, expand the **Descriptors** and **ADF Meta-INF** nodes, and then double-click **adf-config.xml**.
2. Click the **Source** tab to open the file in the source editor, and create an entry for each of the elements shown in Table 53-1.

Table 53-1 ADS Configuration Elements in adf-config.xml

Element	Description	Default Value (in milliseconds)	Minimum Value (in milliseconds)
`<transport>`	The method by which data will be delivered to the client. Value values are: • `streaming` (default) • `polling` • `long-polling`		
`<latency threshold>`	Latency threshold in milliseconds. Active data messages with network delays greater than this threshold will be treated as late.	10000	1000
`<keep-alive-interval>`	Frequency in milliseconds for sending keep-alive messages when no events are generated.	10000	5000
`<polling-interval>`	When `<transport>` set to `polling`, frequency in milliseconds of the poll request.	5000	1000
`<max-reconnect-attempt-time>`	Maximum period of time in milliseconds a client will attempt to reconnect the push channel to the server upon getting disconnected.	1800000 (30 minutes)	0
`<reconnect-wait-time>`	Time interval in milliseconds to wait between reconnect attempts.	10000	1000

Performance Tip:

Keep the following in mind when configuring the ADS.

- Set the latency threshold to more than 1000 to avoid frequent component refreshing.
- Set the keep-alive interval and reconnect wait time to be less than the browser's keep-alive timeout.
- Set the max reconnect time to be less than your web server's session timeout.

The following example shows a sample configuration that has content pushed to the client.

```
<?xml version="1.0" encoding="utf-8" ?>
<adf-config xmlns="http://xmlns.oracle.com/adf/config"
            xmlns:ads="http://xmlns.oracle.com/adf/activedata/config">
```

```
    <ads:adf-activedata-config xmlns=
                          "http://xmlns.oracle.com/adf/activedata/config">
      <transport>streaming</transport>
      <latency-threshold>5000</latency-threshold>
      <keep-alive-interval>10000</keep-alive-interval>
      <max-reconnect-attempt-time>90000</max-reconnect-attempt-time>
      <reconnect-wait-time>8000</reconnect-wait-time>
    </ads:adf-activedata-config>
  </adf-config>
```

3. Synchronize the clocks on the data server and on the application server. If these are not synchronized, then events may appear to ADS to have occurred in the future.

What You May Need to Know About Configuring an ADS Transport Mode

ADS can use one of three transport modes to deliver active data to the component: streaming, polling, or long polling.

When you configure ADS to use the streaming mode, data is pushed to the client whenever a change event is raised by the data. On the client side, after the push channel is established, if there is no activity within the time of the value for the `latency-threshold` element plus the `keep-alive-interval`, an `establish-channel-request` will be sent out repeatedly based on the value of the `reconnect-wait-time` element, until the amount of time passed reaches the value of the `max-reconnect-attempt-time` element. After that, the connection will be considered disconnected. For example, given the values shown in How to Configure the Active Data Service, if there is no activity in 15,000 milliseconds, an establish channel request will be sent out every 8,000 milliseconds for up to 90,000 milliseconds. After that, the connection will be considered disconnected.

On the server side, the server disconnects the push channel and starts a timer to clean up with a `cleanup-delay-time` when there is an empty active data message or when it fails to send out an active data message. The `cleanup-delay-time` is calculated as `max-reconnect-attempt-time` + 30 * 1000 ms. Its default value is 30 minutes.

When you configure ADS to use the polling mode, on the client side the polling request is scheduled to be sent out repeatedly after the value of the `polling-interval` element has been reached. If no response is received after the value of the `max-reconnect-attempt-time` has elapsed, the connection is treated as disconnected and no more requests will be sent. After receiving a polling response, if the time the response has taken is greater than the `polling-interval` element, the service sends the next request out right away. If it is less, the next request will be sent as scheduled.

For the server side, the session ends after the polling response is returned. At that point, a timer with a `cleanup-delay-time` is set up to trigger cleanup. If a new request comes in before the timer fires, the old timer is canceled, and new timer is created.

When you configure ADS to use the long polling mode, requests are made as they are in streaming mode; however, as soon as the connection is treated as disconnected, a new connection is initiated. The result is a significant reduction in latency.

Table 53-2 compares the three different modes.

Table 53-2 Comparison of Streaming, Polling, and Long-Polling Modes

Comparison Benchmark	Streaming	Polling	Long-Polling
Latency	Very good. Directly after an event occurs on the server side, a partial response is sent to the client. If there is another event, immediately, it is also sent as a partial response. There is almost no latency with this approach.	Poor, depending on the polling interval. If the polling interval is short (for example, 0.5 seconds), it will slow down the network because the connections are repeatedly opened. It is also expensive on the client- and server-side resources.	Good. However, when there is a new event immediately after a response has been closed, there is some latency until the new data appears on the client side. On average, this is not a problem.
HTTP Proxy	Poor. For some older servers, because the response is never released, when a proxy is sitting between client and server, it is possible that the proxy will buffer responses. This is an unfortunate optimization that prevents real-time data from flowing into the browser. Long polling should be used if a proxy is used.	Good.	Good.
Number of live connections	Poor. Many concurrent connections, as the stream is always live.	Good. Connections are live only during the actual poll. Note however that if there is a high polling rate then the number of concurrent connections will also be high.	Poor. Many concurrent connections, as the stream is almost always live.
Communication channel	`HTTP GET` This can result in the display of a "busy" cursor or the animation of a browser's "throbber" icon.	`XMLHttpRequest` (XHR) `HTTP GET`	`XMLHttpRequest` (XHR) `HTTP GET`

How to Configure Session Timeout for Active Data Service

By default, applications that you configure to use ADS will not time out. In polling mode, the client periodically sends polling request to server. In long polling mode, the client sends a new polling request immediately after a polling request is returned from the server. In steaming mode, the server holds the GET request response and will indefinitely write active data to it. Due to the behavior of each data transport mode, ADS will maintain the application session active whether or not the user interacts with the application.

To prevent this situation of maintaining an active session indefinitely due to ADS, the `web.xml` configuration file contains the `oracle.adf.view.rich.poll.TIMEOUT` context-parameter, which specifies how long ADS should run before it times out from user inactivity. The ADS client is considered eligible to time out if there is no keyboard or mouse activity. The default timeout period is set at ten minutes. So if the user remains inactive for ten minutes, that is, does not use the keyboard or mouse, then ADS will stop polling or will abort its connection with server, and from that point on, the application participates in the standard server-side session timeout. For example, if the server-side session timeout is 65 minutes, and the polling timeout parameter is set to 15 minutes, then it is expected that the session should expire when the user is inactive for more than 80 minutes.

You can override this timeout setting for a specific web page using the ADF Faces poll component `timeout` attribute. The poll component allows you to deliver a heartbeat to the server to prevent users from being timed out of their session for specific pages. For more information about the `oracle.adf.view.rich.poll.TIMEOUT` context-parameter and individual web pages, see the Using Polling Events to Update Pages section of *Developing Web User Interfaces with Oracle ADF Faces*.

Before you begin:

It may be helpful to have an understanding of the ADS configuration choices. For more information, see Configuring the Active Data Service.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see About the Active Data Service.

To configure the Active Data Service to timeout after session becomes inactive:

1. In the Applications window, in the user interface project, expand the **Web Content** and **WEB-INF** nodes, and then double-click **web.xml**.
2. In the overview editor for the `web.xml` file, click the **Application** navigation tab and in the Context Initialization Parameters section, click the **Create** icon to add the polling timeout parameter as follows:

   Name: Enter the context parameter `oracle.adf.view.rich.poll.TIMEOUT`.

   Value: Enter the amount of time in milliseconds after which all pages configured to use active data will time out and ADS will stop. To disable ADF time out due to user inactivity, enter `-1`. The default value for ADS to timeout is `600000` milliseconds (ten minutes).

Configuring Components to Use the Active Data Service

The process of configuring components to use Active Data Service (ADS) is dependent on the usage of the Active Data proxy. You can configure components to use ADS with or without the proxy.

How you configure components to use ADS depends on whether or not you must use the Active Data proxy. If your application uses a data store that publishes events when data is changed, and your business services react to those events (for example, if your application uses BAM), then you do not need to use the Active Data proxy.

If your business services do not react to those events (for example, if your application uses ADF Business Components), then you must use the Active Data proxy and follow different procedures for configuring your components.

> **Note:**
>
> If your business service requires the use of the Active Data proxy, then you can only use the following components with active data:
>
> - `table`
> - `tree`
> - `treeTable`
> - ADF Data Visualization `chart`, `gauge`, `mapPointTheme`, `pivotTable`, `pivotFilterBar`, `sunburst`, and `treemap` components

How to Configure Components to Use the Active Data Service Without the Active Data Proxy

To use ADS without the proxy, you need to set a value on the binding element in the corresponding page definition file.

Before you begin:

It may be helpful to have an understanding of the ADS configuration choices. For more information, see Configuring Components to Use the Active Data Service.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see About the Active Data Service.

To configure a component to display active data without the Active Data proxy:

1. From the Data Controls panel, drag and drop a collection onto a JSF page, and from the context menu, choose the desired component.
2. If you are dropping the collection as an ADF bound tree or tree table, you need to ensure the following:
   - That the binding represents homogeneous data (that is, only one rule), although an accessor can still access a like accessor.
   - That the binding rule contains a single attribute.
   - That the table does not use filtering.
   - That the tree component's `nodeStamp` facet contains a single `outputText` tag and contains no other tags, as described in What You May Need to Know About Displaying Active Data in ADF Trees.
3. From the Application window, right-click the page and choose **Go to Page Definition**.

   The editor for the page definition file opens for the selected page.

4. In the Structure window, select the node that represents the attribute binding for the component.
5. In the Properties window, expand the **Advanced** section, and select **Push** from the **ChangeEventPolicy** dropdown menu.

> **Tip:**
>
> You can use the `statusIndicator` component to indicate the server state. For more information, see the Displaying Application Status Using Icons section of *Developing Web User Interfaces with Oracle ADF Faces*.

How to Configure Components to Use the Active Data Service with the Active Data Proxy

To use ADS with the proxy, you need bind the value of your component to decorator class that will use the proxy. For more information about this class, see Using the Active Data Proxy to set a value on the binding element in the corresponding page definition file.

Before you begin:

It may be helpful to have an understanding of the ADS configuration choices. For more information, see Configuring Components to Use the Active Data Service.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see About the Active Data Service.

To configure a component to display active data with the Active Data proxy:

1. From the Data Controls panel, drag and drop a collection onto a JSF page, and from the context menu, choose the desired component.
2. If you are using an ADF bound tree or tree table, you need to ensure the following:
   - That the binding represents homogeneous data (that is, only one rule), although an accessor can still access a like accessor.
   - That the binding rule contains a single attribute.
   - That the table does not use filtering.
   - That the tree component's `nodeStamp` facet contains a single `outputText` tag and contains no other tags, as described in What You May Need to Know About Displaying Active Data in ADF Trees.
3. From the Application window, right-click the page and choose **Go to Page Definition**.

   The editor for the page definition file opens for the selected page.
4. In the Structure window, select the node that represents the attribute binding for the component.
5. In the Properties window, change the value attribute to be bound to a decorator class that you will create for use with the proxy.

   For more information, see Using the Active Data Proxy.

What You May Need to Know About Displaying Active Data in ADF Trees

When you create an ADF Faces tree (or tree table) component, you configure a nodeStamp facet, which is a holder for the component used to display the data for each node of the tree. Each node is rendered (stamped) once, repeatedly for all nodes.

Because of this stamping behavior, only certain types of components are supported as children inside an ADF Faces tree component. When the tree component is not bound to an active data source, all components that have no behavior are supported. However, when you configure the tree to use ADS, only the outputText component is supported inside of the nodeStamp facet, as shown in the following example. The nodeStamp facet must not contain any other tags. So, for example, active data will not work if you add panelGroupLayout tags to the nodeStamp facets of a tree configured to use ADS.

```
<f:facet name="nodeStamp">
  <af:outputText value="#{row.str2}"/>
</f:facet>
```

What Happens at Runtime: How Components Render When Bound to Active Data

After you configure your application and the component for ADS, whenever the data changes on the server, the component is notified and rerenders with only the changed data. In contrast, a component not configured for either active data or automatic PPR will need to be explicitly refreshed after a data change occurs. The explicit refresh will refetch all data that is visible on the client, including data that has not changed. Consequently, this will force the entire component to rerender. Additionally, when the component is bound to an active data source (with active data policy "Push"), the component rerenders with the new value highlighted.

What You May Need to Know About Running ADS in a Google Chrome Browser

When the Fusion web application runs in the Google Chrome web browser and a user presses Ctrl+N (or Ctrl+T) to open a new window (or tab) and then copies the URL from the original window into the new window, active data in the original window will stop streaming. According to the Google Chrome process model, the new browser window will be created in a separate process and both windows will share the same HTTP session. However, because browser windows that are created in two separate processes cannot communicate with each other, ADS will become out of sync between the client and server and will stop streaming. As a workaround, to allow active data in multiple Google Chrome windows, before copying the URL from the original window into the new window users must press Ctrl-Shift-N to open the browser window in incognito mode (private browsing). Because Ctrl-Shift-N opens the window in a separate process and a separate HTTP session, ADS will not attempt to synchronize between the windows and streaming will be unaffected.

Using the Active Data Proxy

In order to make business services react to events, you need to use the Active Data proxy in the ADF application. Certain procedures must be followed to configure components.

You use the active data proxy when your business services do not react to data events. If you want your components to update based on events passed into ADF Business Components, then you need to use the Active Data Proxy.

> **Tip:**
>
> If your application uses BAM for the business service, then you do not need to use the Active Data Proxy.

> **Note:**
>
> If your business service requires the use of the Active Data Proxy, then you can only use the following components with active data:
>
> - `table`
> - `tree`
> - `treeTable`
> - ADF Data Visualization `mapPointTheme`, `pivotTable`, `pivotFilterBar`, `sunburst`, and `treemap` components

How to Use the Active Data Proxy

You will create a Java class that subclasses one of the following decorator classes:

- `ActiveCollectionModelDecorator` class
- `ActiveGeoMapDataModelDecorator` class

These classes are wrapper classes that delegate the active data functionality to a default implementation of `ActiveDataModel`. The `ActiveDataModel` class listens for data change events and interacts with the Event Manager. Specifically, it does the following:

- Starts and stops the active data and the `ActiveDataModel` object, and registers and unregisters listeners to the data source.
- Wraps the JSF model interface. For example, the `ActiveCollectionModelDecorator` class wraps the `CollectionModel` class.
- Generates active data events based on data change events from the data source.
- Manages listeners from the Event Manager and pushes active data events to the Event Manager.

You need to implement methods on this Java class that registers itself as the listener of the active data source and gets the model to which the data is being sent.

Note:

The Active Data framework does not support complicated business logic or transformations that require the ADF runtime context, such as a user profile or security. For example, the framework cannot convert an ADF context locale-dependent value and return a locale-specific value.

As an example of complicated business logic, say you have logic that allows a user to move an order from `open` status to `pending`. This change results in an `update` event, which should cause the order to be removed from a data object called "Open Orders." The framework cannot handle this event type transformation based on business logic. Instead, you need to have your data source handle this before publishing the data change event.

Before you begin:

It may be helpful to have an understanding of the active data proxy. For more information, see Using the Active Data Proxy.

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see About the Active Data Service.

You will need to complete this task:

Implement the logic to fire the active data events asynchronously from the data source. For example, this logic might be a business process that updates the database, or a JMS client that gets notified from JMS.

To use the active data service:

1. Create a Java class that extends the decorator class appropriate for your component. The following example shows a class created for a table.

```
package view;

import oracle.adf.view.rich.model.ActiveCollectionModelDecorator;
import oracle.adf.view.rich.activedata.ActiveDataEventUtil;
import oracle.adf.view.rich.activedata.JboActiveDataEventUtil;

/**
 * This code wraps the existing collection model in the page and implements
the
   ActiveDataModel interface to enable ADS for the existing page.
 */
public class DeptModel
  extends ActiveCollectionModelDecorator
{
}
```

2. Implement the method that returns the model. The following example shows an implementation of the `getCollectionModel()` method that relies on expression language (EL) to avoid needing to typecast to an internal class. The method returns the `DepartmentsView1` collection from the binding container.

```
public CollectionModel getCollectionModel()
{
   if (_model == null)
   {
       FacesContext fc = FacesContext.getCurrentInstance();
       Application app = fc.getApplication();
       ExpressionFactory elFactory = app.getExpressionFactory();
       ELContext el = fc.getELContext();

       // This is EL to avoid typecasting to an Internal class.
       ValueExpression ve = elFactory.createValueExpression(el,
                    "#{bindings.DepartmentsView1.collectionModel}",
Object.class);

       // Now GET the collectionModel
       _model = (CollectionModel)ve.getValue(el);
   }
   return _model;
}
```

3. Create an inner class that is your own implementation of an `ActiveDataModel` class, which the decorator can use to start and stop the active data and connect and disconnect from the data source. This class should also use the `changeCount` API to maintain data read consistency, as shown in the following example. For more information, see What You May Need to Know About Maintaining Read Consistency.

```
public class MyActiveDataModel extends BaseActiveDataModel
{
    protected void startActiveData(Collection<Object> rowKeys,
                                   int startChangeCount)
    {
      _listenerCount.incrementAndGet();
      if (_listenerCount.get() == 1)
      {
        System.out.println("start up");

        Runnable dataChanger = new Runnable()
        {
          public void run()
          {
            System.out.println("MyThread starting.");
            try
            {
              Thread.sleep(2000);
              System.out.println("thread running");
              triggerDataChange(MyActiveDataModel.this);
            }
            catch (Exception exc)
            {
              System.out.println("MyThread exceptioned out.");
            }
            System.out.println("MyThread terminating.");
          }
        };
        Thread newThrd = new Thread(dataChanger);
        newThrd.start();
      }
    }

    protected void stopActiveData(Collection<Object> rowKeys)
```

```
        {
          _listenerCount.decrementAndGet();
          if (_listenerCount.get() == 0)
          {
            System.out.println("tear down");
          }
        }

        public int getCurrentChangeCount()
        {
          return _currEventId.get();
        }

        public void bumpChangeCount()
        {
          _currEventId.incrementAndGet();
        }

        public void dataChanged(ActiveDataUpdateEvent event)
        {
          fireActiveDataUpdate(event);
        }

        private final AtomicInteger _listenerCount = new AtomicInteger(0);
        private final AtomicInteger _currEventId = new AtomicInteger();
    }
```

4. Implement the method that will return the `ActiveDataModel` class, as shown in the following example.

```
public ActiveDataModel getActiveDataModel()
  {
    return _activeDataModel;
  }
```

5. Create a method that creates application-specific events that can be used to insert or update data on the active model.

   The following example shows the `triggerDataChange()` method, which uses the active model (an instance of `MyActiveDataModel`) to create `ActiveDataUpdateEvent` objects to insert and update data. You will need to import `oracle.adf.view.rich.activedata.ActiveDataEventUtil` and `oracle.adf.view.rich.activedata.JboActiveDataEventUtil` to implement this method.

```
public void triggerDataChange(MyActiveDataModel l)
  throws Exception
  {


    // do an update on dept 1
      l.bumpChangeCount();
      ActiveDataUpdateEvent event =
        ActiveDataEventUtil.buildActiveDataUpdateEvent(ActiveDataEntry.ChangeType.UPDATE,
                                     l.getCurrentChangeCount(),
                                     JboActiveDataEventUtil.convertKeyPath(new Key
                                            (new Object[]{ new Long(1), new Integer(0) })),
                                     null,
                                     new String[] { "name" },
                                     new Object[] { "Name Pushed" });

      l.dataChanged(event);
```

```
        try
        {
          Thread.sleep(2000);
        }
        catch (InterruptedException ie)
        {
          ie.printStackTrace();
        }
        // insert dept 99

          l.bumpChangeCount();
           event =

    ActiveDataEventUtil.buildActiveDataUpdateEvent(ActiveDataEntry.ChangeType.INSERT_AFTER,
                                         l.getCurrentChangeCount(),
                                         JboActiveDataEventUtil.convertKeyPath(new Key
                                               (new Object[]{ new Long(99), new Integer(0) })),
                                         JboActiveDataEventUtil.convertKeyPath(null),
                                         new String[]{ "addr1", "name" },
                                         new Object[]{ "Addr Inserted", "Name Inserted" });

          l.dataChanged(event);


        try
        {
          Thread.sleep(2000);
        }
        catch (InterruptedException ie)
        {
          ie.printStackTrace();
        }

        // delete dept 10

          l.bumpChangeCount();
           event =
            ActiveDataEventUtil.buildActiveDataUpdateEvent(ActiveDataEntry.ChangeType.REMOVE,
                                         l.getCurrentChangeCount(),
                                         JboActiveDataEventUtil.convertKeyPath(new Key
                                               (new Object[]{ new Long(9), new Integer(0) })),
                                         null, null, null);

          l.dataChanged(event);


        try
        {
          Thread.sleep(2000);
        }
        catch (InterruptedException ie)
        {
          ie.printStackTrace();
        }

        // refresh the entire table

          l.bumpChangeCount();
           event =
            ActiveDataEventUtil.buildActiveDataUpdateEvent(ActiveDataEntry.ChangeType.REFRESH,
                                             l.getCurrentChangeCount(),
                                             null, null, null, null);

          l.dataChanged(event);
        }


      private MyActiveDataModel _activeDataModel = new MyActiveDataModel();

      private CollectionModel _model = null;

    }
```

What You May Need to Know About Maintaining Read Consistency

Read consistency guarantees a consistent view of the data in ADF application runtime. In order to use active data, your application must maintain read consistency.

Using active data means that your component has two sources of data: the active data feed and the standard data fetch. Because of this, you must make sure your application maintains read consistency.

For example, say your page contains a table and that table has active data enabled. The table has two methods of delivery from which it updates its data: normal table data fetch and active data push. Say the back end data changes from `foo` to `bar` to `fred`. For each of these changes, an active data event is fired. If the table is refreshed *before* those events hit the browser, the table will display `fred` because standard data fetch will always get the latest data. But then, because the active data event might take longer, some time *after* the refresh the data change event would cause `foo` to arrive at the browser, and so the table would update to display `foo` instead of `fred` for a period of time. Therefore, you must implement a way to maintain the read consistency.

To achieve read consistency, the `ActiveDataModel` has the concept of a change count, which effectively timestamps the data. Both data fetch and active data push need to maintain this `changeCount` object by monotonically increasing the count, so that if any data returned has a lower `changeCount`, the active data event can throw it away. For an example of an implementation of the `ActiveDataModel` class that maintains read consistency, see How to Use the Active Data Proxy.

Using the Active Data with a Scalar Model

ADS is very much model driven and requires no extra setups in the declarative view. It supports ADF components such as activeCommandToolbarButton, activeImage, activeOutputText, table, tree, and DVT components. Let’s take a look on how to use activeOutputText with a Java Bean, as you may not have a model.

ADF components that display collection-based data can be configured to work with ADS and require no extra setup in the view layer. However, imagine that your JSPX page uses an `activeOutputText` component to display new counts based on a Java timer. In this case, you will replace the model layer with scalar or "flat" data that you display from a Java Bean.

How to Use the Active Data with a Scalar Model

To implement the scalar model, follow these basic steps (as illustrated in the following example):

1. Use the `ActiveModelContext` API to register the bean with ADS so the bean (a scalar model) imitates an actual model.

   Add the registration code to the active attribute's getter method. Adding the registration code to the bean's constructor can cause ADS to fail.

2. Implement a custom mechanism to push the data to the view layer, as shown in the following example.

```
package oracle.adfdemo.view.feature.rich;

import java.util.Collection;
import java.util.Timer;
import java.util.TimerTask;
import java.util.concurrent.atomic.AtomicInteger;

import oracle.adf.view.rich.activedata.ActiveModelContext;
import oracle.adf.view.rich.activedata.BaseActiveDataModel;
import oracle.adf.view.rich.activedata.ActiveDataEventUtil;
import oracle.adf.view.rich.event.ActiveDataEntry;
import oracle.adf.view.rich.event.ActiveDataUpdateEvent;


public class CounterBean extends BaseActiveDataModel
// Example using a Java timer to create new counts
{
  public CounterBean()
  {
    timer.schedule(new UpdateTask(), 2000, 2000);
  }

  public String getState()
  {
    // 1. Use the ActiveModelContext API to register the scalar model key path
for
    //    the "state" attribute.
    ActiveModelContext context = ActiveModelContext.getActiveModelContext();
    Object[] keyPath = new String[0];
    context.addActiveModelInfo(this, keyPath, "state");

    return String.valueOf(counter);
  }

  // Not needed. We do not need to connect to a (real) active data scource.
  protected void startActiveData(Collection<Object> rowKeys, int
          startChangeCount) {}

  // Not needed. We do not need to connect to a (real) active data scource.

  protected void stopActiveData(Collection<Object> rowKeys) {}

  public int getCurrentChangeCount()
  {
    return counter.get();
  }
  protected class UpdateTask extends TimerTask
  {
    public void run()
    {
      counter.incrementAndGet();

      // 2. Use ActiveDataEventUtil to create an event object to update the
model.
      ActiveDataUpdateEvent event =
        ActiveDataEventUtil.buildActiveDataUpdateEvent(
          ActiveDataEntry.ChangeType.UPDATE,
          counter.get(),
          new String[0],
          null,
          new String[] { "state" },
          new Object[] { counter.get() });
      fireActiveDataUpdate(event);
    }
  }

  private static final Timer timer = new Timer();
  private final AtomicInteger counter = new AtomicInteger(0);
}
```

After you create the bean, register the bean as a managed bean in the faces-config.xml file, as the following example for counterBean illustrates:

```
...
<managed-bean>
  <managed-bean-name>counterBean</managed-bean-name>
  <managed-bean-class>
      oracle.adfdemo.view.feature.rich.CounterBean
  </managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

Once the bean is registered, you can use ADS to stream the data to the view layer. Your ADF Faces component use expression language to receive the pushed data, as illustrated by the activeOutputText component value attribute in the following example:

```
...
<f:view>
   <af:document title="Active Data Visual Design Demo"
                  binding="#{templateBindings.documentComponent}"
                 smallIconSource="#{aboutBean.smallIconSource}"
                 largeIconSource="#{aboutBean.largeIconSource}" theme="dark"
                  id="d1">
      <af:pageTemplate id="dmoTpl" viewId="#{templates.componentTemplate}">
        <f:attribute name="tagName" value="Active Data Visual Design"/>
        <f:attribute name="demoKind" value="visualDesign"/>
        <f:attribute name="customEditorPresent" value="true"/>
        <f:facet name="center">
          <af:panelGroupLayout layout="scroll">
             <af:activeOutputText
                value="#{counterBean.state}"
                inlineStyle=
                   "color:brown;
                    font-size:100px;
                    font-weight:bold;
                    text-align:center;"
              />
          </af:panelGroupLayout>
        </f:facet>
      </af:pageTemplate>
   </af:document>
</f:view>
```

54

Configuring High Availability for Fusion Web Applications

This chapter describes considerations for developing Fusion web applications that you intend to deploy to a high availability, clustered server environment.
This chapter includes the following topics:

- About Oracle ADF High Availability
- Configuring Oracle ADF for High Availability
- Troubleshooting Fusion Web Applications for High Availability

About Oracle ADF High Availability

Oracle ADF is used to build large enterprise applications. And if you want to have your application run on high availability servers, then you should understand what is high availability. High availability for a Fusion web application refers to the ability of users to access a system without loss of service.

High availability for the Fusion web application is the ability of the application to be available when it is needed.

A high availability environment ensures that users can access the application without loss of service. Deploying the Fusion web application to a high availability environment minimizes the time when the application is down, or unavailable, and maximizes the time when it is running, or available.

High availability comes from redundant systems and components. You can categorize high availability solutions by their level of redundancy into active-active solutions and active-passive solutions.

An **active-active solution** deploys two or more active servers and can be used to improve scalability and provide high availability. In active-active deployments, all instances handle requests concurrently. Oracle recommends active-active solutions for all single-site middleware deployments. **Active-passive solutions** deploy an active instance that handles requests and a passive instance that is on standby.

For information about high availability environments, see What is High Availability in *High Availability Guide*.

Oracle ADF Scope and Session State Considerations

At runtime, ADF objects such as the binding container and managed beans are instantiated. Each of these objects has a defined life span set by its scope attribute.

For more information about the life span of ADF objects, see About Object Scope Lifecycles.

Oracle ADF Failover and Expected Behavior Considerations

An Oracle WebLogic cluster provides application high availability. If one member of the cluster is unavailable, any other available member of the cluster is able to handle the request. You should familiarize yourself with failover requirements and the expected behavior of an application in the event of a failover.

Session Failover Requirements

For seamless failover of a Fusion web application, the application must meet the following conditions:

- The application is in a cluster and at least one member of the application cluster is available to serve the request.
- For stateful applications, state replication is configured correctly, as described in Configuring Oracle ADF for High Availability.
- If you use Oracle HTTP Server, the server is configured with the WebLogicCluster directive to balance among all available application instances.
- If you use a hardware load balancer, the load balancer is:
  - Routing traffic to all available instances
  - Configured correctly with a health monitor to mark unavailable instances
  - Configured to support persistence of session state

Expected Behavior for Application Failover

If the environment is configured correctly, application users do not notice when an application instance in a cluster becomes unavailable. The sequence of events in an application failover is, for example, as follows:

1. A user makes a request and is routed by a hardware load balancer to Instance A of the application.
2. Instance A of the application becomes unavailable because of node failure, process failure, or network failure.
3. The hardware load balancer marks Instance A as unavailable.
4. The user makes a subsequent request. The request is routed to Instance B.
5. Instance B is configured as a replication partner of Instance A and has the user's session state.
6. The application resumes using the session state on Instance B and the user continues working without interruption.

ADF Application Module for Oracle RAC Considerations

When you configure the ADF application module to access a highly available database system, such as redundant databases or Oracle Real Application Clusters (Oracle RAC) as the backend, the data source must be container-defined. In this scenario, you must use a GridLink data source or a multi data source. However, from the standpoint of the application module configuration, the naming convention for the multi data source or GridLink data source is the same as it is for a non-multi data source or

GridLink data source. This naming convention ensures that the correct data source is used at runtime.

To more information about configuring GridLink data sources or multi data sources for high availability applications, see the Database Considerations chapter of the *High Availability Guide*.

For information about high availability considerations related to modifying the mdsDS data source URL post deployment of the Fusion web application, see the Configuring High Availability for Other Components chapter of the *High Availability Guide*.

Configuring Oracle ADF for High Availability

When a failover happens in the cluster environment, the ADF application state must be saved in a shared location either in a file system or database or in memory and this should be available to all the nodes in the cluster to replicate the state of the session. Most of the configurations for a high availability will be done in the application metadata files and the development needs to be done keeping in mind that code should support high availability for any kind of J2EE applications.

To support automatic replication and failover for web applications in a clustered environment, Oracle WebLogic Server supports mechanisms for replicating HTTP session state across clusters. You can configure Oracle ADF to ensure the Fusion web application's state can be restored from any server in the cluster.

How to Configure Application Modules for High Availability

An **application module** is the transactional component that the Fusion web application uses to work with application data. It defines an updateable data model and top-level procedures and functions, called *service methods*, related to a logical unit of work related to an end-user task. An application module supports *passivating*, or storing, its transaction state as a snapshot in the database. It also supports the reverse operation of activating the transaction state from one of these saved snapshots.

For more information on application module state management, see About Fusion Web Application State Management.

To enable support for ADF Business Components failover, set the `jbo.dofailover` parameter to `true` so that the application module state is saved on release. This enables Oracle ADF to restore the application module state from a snapshot saved from a previous check in. By contrast, when the failover feature is disabled, which it is by default, then application module state is saved only when the application is reused by a subsequent user session and only when the application module pool cannot find an unused application module.

You can set this parameter in your application module configuration on the **Pooling and Scalability** tab of the overview editor for application module configurations (on the `bc4j.xcfg` file).

To configure application modules for high availability:

1. Launch JDeveloper and open the application.
2. In the Applications window, expand the project that contains the data model and double-click the application module.

3. In the overview editor for the application module, select the **Configurations** navigation tab and click the configuration hyperlink for the configuration you want to configure for high availablity.
4. In the overview editor for application module configurations (on the `bc4j.xcfg` file), click the **Database and Scalability** tab.
5. Select the **Failover Transaction State Upon Managed Release** checkbox and save the change.

How to Enable Support for Replicating HTTP Session State

To enable support for replicating HTTP session state, you must assign a value to the `persistent-store-type` element in the Oracle WebLogic Server `weblogic.xml` file. The value `replicated_if_clustered` ensures that the in-effect persistent store type will be replicated so that sessions on the clustered environment are stored in accordance with the value set for the cluster of servers to which this server belongs.

> **Note:**
>
> Oracle ADF applications such as the Oracle WebCenter Portal Suite are preconfigured and do not need additional configuration.

To configure the weblogic.xml file for high availability:

1. Launch JDeveloper and open the application.
2. In the Applications window, expand the project that contains the web application and expand the **WEB-INF** folder.
3. Double-click the **weblogic.xml** file, and click the **Source** tab to edit the file.
4. In the file, add the `persistent-store-type` definition to the `session-descriptor` element:

```
<weblogic-web-app>
 <session-descriptor>
   <persistent-store-type>
     replicated_if_clustered
   </persistent-store-type>
 </session-descriptor>
</weblogic-web-app>
```

How to Ensure Oracle ADF Receives Notifications From Managed Bean Changes

When you design an application to run in a clustered environment, you must ensure that Oracle ADF is aware of changes to managed beans stored in ADF scopes (view scope and page flow scope).

When a value within a managed bean in either view scope or page flow scope is modified, the application must notify Oracle ADF so that it can ensure the bean's new value is replicated.

To enable ADF Controller to track changes to ADF memory scopes and replicate the page flow scope and view scope within the server cluster, you must set the ADF Controller parameter `<adf-scope-ha-support>` in the application's `adf-config.xml` file to `true`. For example, when set to `true` for an application and that application adds or removes a bean from a page flow scope during a request, the change will automatically replicated within a cluster.

The `adf-config.xml` file is the central configuration file for all ADF components. The file contains sections to configure the runtime behavior for ADF components, including, ADF Controller.

> **Note:**
>
> If your application uses MDS and will use an Oracle database that supports failover, Oracle recommends enabling MDS retry on failover. To do this, add the following `retry-connection` entry to the MDS configuration section of adf-config.xml.
>
> ```
> <persistence-config>
>       <metadata-namespaces>...
>       <metadata-store-usages>...
>       <external-change-detection enabled="false" />
>       <read-only-mode enabled="true"/>
>       <retry-connection enabled="true"/>
> </persistence-config>
> ```

To configure the adf-config.xml file for high availability:

1. In the Application window, expand the **Application Resources**.
2. Select **Descriptors**, and then select the **ADF META-INF** node.
3. Double-click the **adf-config.xml** file, and click the **Source** tab to edit the file.
4. Add the following to the file:

```
<adf-controller-config xmlns="http://xmlns.example.com/adf/controller/
config">
 <adf-scope-ha-support>true</adf-scope-ha-support>
</adf-controller-config>
```

For more information about using the `adf-config.xml` file to configure ADF, see adfc-config.xml.

What You May Need to Know About Using Application Scoped Managed Beans in a Clustered Environment

JSF Application Scope beans are backed by a servlet context which can only have one instance per cluster node. Therefore, for each design time configured bean, there will be a runtime instance of the bean for each node in a cluster. Because there is no synchronization across nodes, each instance of the application running on different nodes will have separate instances of that bean with separate values. Therefore, application scoped managed beans should not be used to carry application-wide values that can be updated to be shared across all nodes in a cluster. They can, however, be used to carry constants, read-only values, or values to reflect the state of a single node within the cluster (such as the number of users on a node).

How to Disable Design Time Optimizations

The `org.apache.myfaces.trinidad.CHECK_FILE_MODIFICATION` context parameter provides optimization for developing and testing the Fusion web application in the JDeveloper. Leaving this parameter set to `true` when the application runs in a high availability environment can lead to errors after failover occurs. Note that this limitation should not affect your work because you typically use this parameter in a development environment only, not a high availability environment.

To configure the CHECK_FILE_MODIFICATION parameter for high availability:

1. In the Applications window, expand the user interface project and expand the **WEB-INF** folder.
2. Double-click the **web.xml** file and click the **Application** navigation tab.
3. In the overview editor, expand the **Context Initialization Parameters** section and confirm that the `org.apache.myfaces.trinidata.CHECK_MODIFICATION` parameter definition is set to `false`.

   The `web.xml` file definition will be:

```
<web-app>
 ...
  <context-param>
    <param-name>org.apache.myfaces.trinidad.CHECK_FILE_
                                                         MODIFICATION</param-name>
    <param-value>false</param-value>
  </context-param>
</web-app>
```

Troubleshooting Fusion Web Applications for High Availability

There are procedures to troubleshoot common issues with Fusion web applications at design time and also after deployment.

This section describes procedures for troubleshooting possible issues with Fusion web applications at design time and after deployment.

How to Test the Fusion Web Application With Failover Mode Enabled

When running or debugging an application that uses failover support (`jbo.dofailover` is enabled) within the JDeveloper environment, you are frequently starting and stopping the application server. The ADF failover mechanism has no way of knowing whether you stopped the server to simulate an application server failure, or whether you stopped it because you want to retest something from scratch in a fresh server instance. If you intend to do the latter, exit out of your browser before restarting the application on the server. This eliminates the chance that you will be confused by the correct functioning of the failover mechanism when you didn't intend to be testing that aspect of your application.

How to Troubleshoot High Availability Issues at Design Time

When you develop the Fusion web application in Oracle JDeveloper, the integrated development environment provides support for detecting potential high availability issues. The warnings that JDeveloper provides are generated by the audit framework and are triggered to appear in the JDeveloper source editors. The warnings the editors show are based on the audit rules for high availability applications.

The high availability audit rules that JDeveloper enables by default are:

- **ADF Controller Configuration - High Availability for ADF Scopes is not Enabled** warns the developer that the `adf-scope-ha-support` flag in the `adf-config.xml` file is set is not set to `true`. This audit rule fires only when the `<adf-controller-config>` element is in the ADF application-level configuration file (`adf-config.xml`).
- **ADF Page Flows - Bean in Scope Map is Modified** warns the developer when the some code calls a setter method on a bean to indicate that the code did not subsequently call the `ControllerContext.markScopeDirty()` method. This audit rule fire only when the `adf-scope-ha-support` flag in the `adf-config.xml` file is set to true.
- **ADF Page Flows - EL Bean is Modified** warns the developer when some code evaluates an EL expression that mutates a bean to indicate that the code did not subsequently call the `ControllerContext.markScopeDirty()` method. This audit rule fire only when the `adf-scope-ha-support` flag in the `adf-config.xml` file is set to true.
- **ADF Page Flows - Managed Bean Class Not Serializable** warns the developer that a managed bean has a non-serializable class defined in `viewScope`, `pageFlowScope`, or `sessionScope`. This audit rule fire only when the `adf-scope-ha-support` flag in the `adf-config.xml` file is set to `true`.

You can modify the high availability audit rule settings using the Preference dialog in JDeveloper. From the JDeveloper toolbar, choose **Tools - Preferences**, under **Audit - Profiles** expand **ADF Controller Configuration** or **ADF Pages Flows** and make the desired audit rule selections.

You can also trigger the audit by choosing **Build - Audit *project.jpr*** from the JDeveloper toolbar.

How to Troubleshoot High Availability Issues After Deployment

There are two actions to take if you encounter Fusion web application deployment issues: verify the JRF Runtime and verify the status of application deployments.

Verifying the JRF Runtime Installation

The first step to troubleshooting an Fusion web application deployment is to verify that the JRF Runtime is installed on the WebLogic Server domain. ADF applications require the JRF and ADF runtime. You cannot run ADF applications with a standalone WebLogic Server domain or just the WebLogic Server portion of the Application Development product; you must extend it with the JRF extension template.

For more information on the JRF Template, see Oracle JRF and ADF Templates in *Domain Template Reference*.

Verifying the Success of All Application Deployments

Fusion web applications deploy when the Managed Server first starts. Use the Administration Console to check that all application deployments were successful:

1. Click **Deployments** in the left hand pane. The right hand pane shows the application deployments and their status. The state of all applications, assuming all the servers are running, should be ACTIVE.
2. If an application deployment has failed, the server logs may provide some indication of why the application was not deployed successfully. The server logs are located in the `DOMAIN_HOME/servers/server_name/logs` directory. Common issues include:

- Unavailability of external resources, such as database resources. Examine the error, fix it, and attempt to redeploy the application.
- The appropriate applications or libraries are not targeted correctly to the right Managed Server or Cluster.

How to Troubleshoot Oracle ADF Replication and Failover Issues

State Replication is most prominent in failover scenarios. A user working on one server may discover that, upon failover:

- Windows may close or the state might reset.
- Screens may require a reset.
- The application may redirect to the logon screen.

To diagnose and troubleshoot state replication issues.

1. Confirm that this is not a known replication issue.

   See Oracle ADF Failover and Expected Behavior Considerations for possible expected behaviors. Before proceeding to further diagnose the issue, first confirm that the failover behavior is not an expected behavior.
2. Check load balancer settings.

   For replication and failover to function correctly, the load balancer must be configured with the appropriate persistence settings. For more details on configuring Hardware Load Balancers for Oracle WebLogic Server, see Load Balancing in a Cluster in *Administering Clusters for Oracle WebLogic Server*.
3. Check the cluster status.

   Replication occurs within the context of a cluster. For failover to be successful, there must be at least one other healthy member of the cluster available. You can check cluster status in one of two ways:

   - Check the cluster status using the Oracle WebLogic Server Administration Console - In the Left-hand pane, click on **Servers**. Verify the state of all servers in the cluster.
   - Check the cluster status using the `weblogic.Admin` utility. You can use the `weblogic.Admin` utility to query the state of all servers in a specific cluster. For example:

```
$ java weblogic.Admin -url Adminhost:7001 -username <username> -password
 <password> CLUSTERSTATE -clustername Spaces_Cluster
```

This example returns:

```
There are 2 server(s) in cluster: Spaces_Cluster
The alive servers and their respective states are listed below:
Application Server---RUNNING
Managed Server---RUNNING
```

4. Check cluster communications.

   Even if all cluster members are running, there may be communication issues which prevent them from communicating replication information to each other. There are two types of cluster communication configurations. Troubleshooting depends on the cluster type:

   - Checking Unicast cluster communications - For Unicast clusters, Managed Servers must be able to access each other's hosts and each other's default listening port.

     Ensure that all individual Managed Servers have their Listen Address set correctly. You can find this setting by selecting **Configuration**, **General** for each Managed Server.

   - Checking Multicast cluster communications - For multicast clusters, servers must be able to intercept the same multicast traffic. Ensure that multicast is configured correctly by running the WebLogic utility `utils.MulticastTest` on each machine. For example:

     ```
     $ java utils.MulticastTest -H
     ```

5. Confirm Oracle WebLogic Server application configuration.

   Oracle WebLogic Server is not configured by default for failover. In-memory replication takes place only with the proper setting in the `weblogic.xml` file:

   ```
   <session-descriptor>
   <persistent-store-type>replicated_if_clustered</persistent-store-type>
   </session-descriptor>
   ```

   A persistent-store-type of replicated is also acceptable. This setting can be made in JDeveloper, as described in How to Enable Support for Replicating HTTP Session State .

6. Confirm Oracle ADF Business Components configuration.

   Oracle ADF is not configured by default for failover. Failover is supported only with the proper setting in the ADF Business Components configuration file (`bc4j.xcfg`):

   ```
   <AppModuleConfig ...
    <AM-Pooling jbo.dofailover="true"/>
   </AppModuleConfig>
   ```

   This setting is made in JDeveloper through the Edit Business Components Configuration dialog, as described in How to Configure Application Modules for High Availability.

7. Confirm Oracle WebLogic Server connection pool parameter.

   Set an appropriate value for the `weblogic-application.xml` deployment descriptor parameter `inactive-connection-timeout-seconds` on the element `<connection-check-params>` `pool-params`.

When enabling application module state passivation, a failure can occur when Oracle WebLogic Server is configured to forcibly release connections back into the pool. The failure creates an exception "`Connection has already been closed`" that gets saved to the server log. The user interface does not show this exception.

Set `inactive-connection-timeout-seconds` to several minutes. In most cases, this setting avoids forcing the inactive connection timeout and passivation failure. Adjust the setting as needed for your environment.

8. Confirm ADF Controller configuration.

   Oracle ADF is not configured by default to replicate changes to ADF objects in ADF memory scopes. ADF object replication is supported only with the proper setting in the ADF application-level configuration file (`adf-config.xml`):

   ```
   <adfc:adf-controller-config>
    <adfc:adf-scope-ha-support>true</adfc:adf-scope-ha-support>
   </adfc:adf-controller-config>
   ```

   This setting is made in JDeveloper through the source editor. See How to Configure Application Modules for High Availability.

9. Check default logger messages.

   By default the ADF log shows high-level messages (`INFO` level). The default logging often reports problems with serialization and replication without the need to enable more detailed log messages.

10. Enable log messages for ADF high availability applications.

    Configure the ADF logger to output runtime messages for high availability. By default the ADF log shows high-level messages (`INFO` level). You enable high availability diagnostics for ADF Controller by setting the logging level in Fusion Middleware Control to `FINE`.

    When enabled, the logger outputs a warning if the `adfc:adf-scope-ha-support` setting in the `adf-config.xml` file is not set.

11. Enable debug.

    Check the server logs for any unusual messages on Managed Server startup. In particular, if the Managed Server is unable to locate other members of the cluster. The server logs are located in the `*DOMAIN_HOME*/servers/*SERVER_NAME*/logs` directory.

    For further debugging, enable the flags `DebugCluster`, `DebugClusterAnnouncements`, `DebugFailOver`, `DebugReplication`, and `DebugReplicationDetails`. You can enable each flag with the `weblogic.Admin` utility:

    ```
    $ java weblogic.Admin -url Adminhost:7001 -username <username> -password
     <password> SET -type ServerDebug -property DebugCluster true
    ```

12. Enable component state serialization checking.

    Enable server checking to ensure no unserializable state content on session attributes is detected. This check is disabled by default to reduce runtime overhead. Serialization checking is supported by the Java server system property `org.apache.myfaces.trinidad.CHECK_STATE_SERIALIZATION`.

    Table 54-1 shows the options you can use with the property. Use commas to delimit the options.

Table 54-1 CHECK_STATE_SERIALIZATION Options

Option	Description
tree	Checks whether the entire component tree is serializable. This is the fastest component check. Most testing should be performed with this flag enabled.
component	Checks each component individually for serializability. This option is much slower than "tree." It is typically turned on only after testing with "tree" reports an error. This option narrows down the problematic component.
property	Checks each component attribute individually for serializability. This is slower than "component" and narrows down the specific problematic component attribute after a failure has been detected in the "tree" or "component" modes.
session	Checks that all attributes in the JSF Session Map that are marked as Serializable are serializable.
application	Checks that all attributes in the JSF Application Map that are marked as Serializable are serializable.
beans	Checks that any serializable object in the appropriate map has been marked as dirty if the serializable content of the object changes during the request.
all	Checks everything.

For high availability testing, start off by validating that the Session and JSF state is serializable by launching the application server with the system property:

```
-Dorg.apache.myfaces.trinidad.CHECK_STATE_SERIALIZATION=session,tree
```

Add the `beans` option to check that any serializable object in the appropriate map has been marked as dirty if the serialized content of the object has changed during the request:

```
-Dorg.apache.myfaces.trinidad.CHECK_STATE_SERIALIZATION=session,tree,beans
```

If a JSF state serialization failure is detected, relaunch the application server with the system property to enable component and property flags and rerun the test:

```
-Dorg.apache.myfaces.trinidad.CHECK_STATE_SERIALIZATION=all
```

These are Java system properties and you must specify them when you start the application server.

55
Deploying Fusion Web Applications

This chapter describes how to deploy Oracle ADF applications to a target application server. It describes how to create deployment profiles, how to create deployment descriptors, and how to load ADF runtime libraries. It includes instructions for running an application in the Integrated WebLogic Server as well as deploying to a standalone Oracle WebLogic Server.
This chapter includes the following sections:

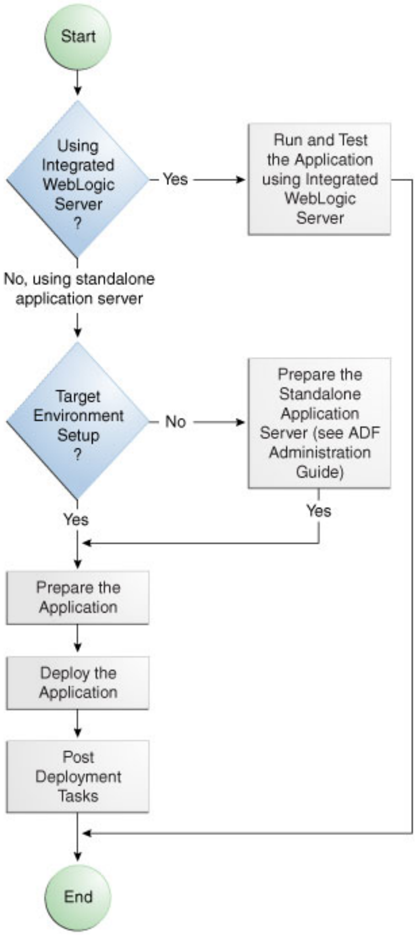
About Deploying Fusion Web Applications

In order to run a web application, it needs to be deployed to a web server. Using JDeveloper, you can directly deploy an Oracle ADF application to an integrated application server or deploy to any standalone application server.

Deployment is the process of packaging application files as an archive file and transferring that file to a target application server. You can use JDeveloper to deploy **Oracle ADF** applications directly to the application server (such as Oracle WebLogic Server), or indirectly to an archive file as the deployment target, and then install this archive file to the target server. For application development, you can also use JDeveloper to run an application in Integrated WebLogic Server. JDeveloper supports deploying to server clusters, but you cannot use JDeveloper to deploy to individual Managed Servers that are members of a cluster.

Figure 55-1 shows the flow diagram that describes the overall deployment process. Note that preparing the target application server for deployment by installing the ADF runtime is described in Deploying ADF Applications in *Administering Oracle ADF Applications*.

Figure 55-1 Deployment Overview Flow Diagram


Start
Using Integrated WebLogic Server ?
Yes
Run and Test the Application using Integrated WebLogic Server
No, using standalone application server
Target Environment Setup ?
No
Prepare the Standalone Application Server (see ADF Administration Guide)
Yes
Yes
Prepare the Application
Deploy the Application
Post Deployment Tasks
End

Note:

Normally, you use JDeveloper to deploy applications for development and testing purposes. If you are deploying Oracle ADF applications for production purposes, you can use Enterprise Manager or scripts to deploy to production-level application servers.

For more information about deployment to later-stage testing or production environments, see Deploying Using Oracle Enterprise Manager Fusion Middleware Control in *Administering Oracle ADF Applications*.

ADF Java EE applications are based on standardized, modular components and can be deployed to the following application server:

- Oracle WebLogic Server

  Oracle WebLogic Server provides a complete set of services for those modules and handles many details of application behavior automatically, without requiring programming. For information about which versions of Oracle WebLogic Server are compatible with JDeveloper, see the certification information website at http://www.oracle.com/technetwork/developer-tools/jdev/documentation/index.html.

Deploying a Fusion web application is slightly different from deploying a standard Java EE application. JSF applications that contain ADF Faces components have a few additional deployment requirements:

- ADF Faces requires Sun's JSF Reference Implementation 1.2 and MyFaces 1.0.8 (or later).

You can use JDeveloper to:

- Run applications in Integrated WebLogic Server

  You can run and debug applications using Integrated WebLogic Server and then deploy to standalone WebLogic Server.
- Deploy directly to the standalone application server

  You can deploy applications directly to the standalone application server by creating a connection to the server and choosing the name of that server as the deployment target.
- Deploy to an archive file

  You can deploy applications indirectly by choosing an EAR file as the deployment target. The archive file can subsequently be installed on a target application server.

The Summit sample application for Oracle ADF demonstrates the use of the Fusion web application technology stack to create transaction-based web applications. You can run the Summit sample application in JDeveloper using Integrated WebLogic Server. For more information about the Summit ADF sample applications, see Introduction to the ADF Sample Application.

Developing Applications with Integrated WebLogic Server

If you are developing an application in JDeveloper and you want to run the application in Integrated WebLogic Server, you do not need to perform the tasks required for deploying directly to Oracle WebLogic Server or to an archive file. JDeveloper has a default connection to Integrated WebLogic Server and does not require any deployment profiles or descriptors. Integrated WebLogic Server has a preconfigured domain that includes the ADF libraries, as well as the `-Djps.app.credential.overwrite.allowed=true` setting, that are required to run Oracle ADF applications. You can run an application by choosing **Run** from the JDeveloper main menu.

You debug the application using the features described in Testing and Debugging ADF Components.

Developing Applications to Deploy to Standalone Application Server

Typically, for deployment to standalone application servers, you test and develop your application by running it in Integrated WebLogic Server. You can then test the application further by deploying it to standalone Oracle WebLogic Server in development mode to more closely simulate the production environment.

In general, you use JDeveloper to prepare the application or project for deployment by:

- Creating a connection to the target application server

- Creating deployment profiles (if necessary)
- Creating deployment descriptors (if necessary, and that are specific to the target application server)
- Updating `application.xml` and `web.xml` to be compatible with the application server (if required)
- Enabling the application for Real User Experience Insight (RUEI) in `web.xml` (if desired)
- Migrating application level security policy data to a domain-level security policy store
- Configuring the Oracle Single Sign-On (Oracle SSO) service and properties in the domain `jps-config.xml` file when you intend the web application to run using Oracle SSO

You must already have an installed application server. For Oracle WebLogic Server, you can use the Oracle 12**c** Installer or the Oracle Fusion Middleware 12**c** Application Developer Installer to install one. For other application servers, follow the instructions in the applications server documentation to obtain and install the server.

You must also prepare the application server for Fusion web application deployment. See Preparing the Standalone Application Server for Deployment in *Administering Oracle ADF Applications*.

- Extending Oracle WebLogic Server domains to be ADF-compatible using the ADF runtime
- For WebLogic, setting the Oracle WebLogic Server credential store overwrite setting as required (`-Djps.app.credential.overwrite.allowed=true` setting).
- Creating a global JDBC data source for applications that require a connection to a data source

After the application and application server have been prepared, you can:

- Use JDeveloper to:
  - Directly deploy to the application server using the deployment profile and the application server connection.
  - Deploy to an EAR file using the deployment profile. For Oracle ADF applications, WAR and MAR files can be deployed only as part of an EAR file.
- Use Enterprise Manager, scripts, or the application server's administration tools to deploy the EAR file created in JDeveloper. See Deploying Using Oracle Enterprise Manager Fusion Middleware Control in *Administering Oracle ADF Applications*.

Running a Fusion Web Application in Integrated WebLogic Server

In order to test run an ADF application during development, you can use JDeveloper to run an application in Integrated WebLogic Server. You do not have to manually complete many of the steps that are necessary for deployment to a standalone server.

JDeveloper is installed with Integrated WebLogic Server which you can use to test and develop your application. For most development purposes, Integrated WebLogic

Server will suffice. When your application is ready to be tested, you can select the run target and then choose the **Run** command from the main menu.

> **Note:**
>
> The first time you run an application in Integrated WebLogic Server, the Create Default Domain dialog appears for you to define an administrative password for the new domain.

When you run the application target, JDeveloper detects the type of Java EE module to deploy based on artifacts in the projects and workspace. JDeveloper then creates an in-memory deployment profile for deploying the application to Integrated WebLogic Server. JDeveloper copies project and application workspace files to an "exploded EAR" directory structure. This file structure closely resembles the EAR file structure that you would have if you were to deploy the application to an EAR file. JDeveloper then follows the standard deployment procedures to register and deploy the "exploded EAR" files into Integrated WebLogic Server. The "exploded EAR" strategy reduces the performance overhead of packaging and unpackaging an actual EAR file.

In summary, when you select the run target and run the application in Integrated WebLogic Server, JDeveloper:

- Detects the type of Java EE module to deploy based on the artifacts in the project and application
- Creates a deployment profile in memory
- Copies project and application files into a working directory with a file structure that would simulate the "exploded EAR" file of the application.
- Performs the deployment tasks to register and deploy the simulated EAR into Integrated WebLogic Server
- Automatically migrates identities, credentials, and policies

  Later on, if you plan to deploy the application to a standalone WebLogic Server instance, you will need to migrate this security information. See How to Deploy Applications with ADF Security Enabled.

> **Note:**
>
> JDeveloper ignores the deployment profiles that were created for the application when you run the application in Integrated WebLogic Server.

The application will run in the base domain in Integrated WebLogic Server. This base domain has the same configuration as a base domain in a standalone WebLogic Server instance. In other words, this base domain will be the same as if you had used the Oracle Fusion Middleware Configuration Wizard to create a base domain with the default options in a standalone WebLogic Server instance.

JDeveloper will extend this base domain with the necessary domain extension templates, based on the JDeveloper technology extensions. For example, if you have installed JDeveloper Studio, JDeveloper will automatically configure the Integrated

WebLogic Server environment with the ADF runtime template (JRF Fusion Middleware runtime domain extension template).

You can explicitly create a default domain for Integrated WebLogic Server. You can use the default domains to run and test your applications. Open the Application Servers window, right-click **IntegratedWebLogicServer** and choose **Create Default Domain**.

How to Run an Application in Integrated WebLogic Server

You can test an application by running it in Integrated WebLogic Server. You can also set breakpoints and then run the application within the ADF Declarative Debugger.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you run an application in the Integrated WebLogic Server. For more information, see Running a Fusion Web Application in Integrated WebLogic Server.

To run an application in Integrated WebLogic Server:

1. In the Applications window, select the project, unbounded task flow, JSF page, or file as the run target.
2. Right-click the run target and choose **Run** or **Debug**.

   The Create Default Domain dialog appears for the first time you run your application and start a new domain in Integrated WebLogic Server. Use the dialog to define an administrator password for the new domain. Passwords you enter can be eight characters or more and must have a numeric character.

How to Run an Application with Metadata in Integrated WebLogic Server

When an application is running in Integrated WebLogic Server, the MAR (Metadata Archive) profile itself will not be deployed to a repository, but a simulated MDS repository will be configured for the application that reflects the metadata information contained in the MAR. This metadata information is simulated, and the application runs based on this location in source control.

Any customizations or documents created by the application that are not configured to be stored in other MDS repositories are written to this simulated MDS repository directory. For example, if you customize an object, the customization is written to the simulated MDS repository. If you execute code that creates a new metadata object, then this new metadata object is also written to the same location in the simulated MDS repository. You can keep the default location for this directory (`ORACLE_HOME\jdeveloper\`**systemXX.XX**`\o.mds.dt\adrs\`**Application**`\AutoGeneratedMar\mds_adrs_writedir`), or you can set it to a different directory. You also have the option to preserve this directory across different application runs, or to delete this directory before each application run.

If your workspace has different working sets, only the metadata from the projects defined in the working set and their dependent projects will be included in the MAR. You can view and change a project's dependencies by right-clicking the project in the Applications window, choosing **Project Properties**, and then selecting **Dependencies**. For instance, an application may have several projects but `workingsetA` is defined to be `viewcontroller2` and `viewcontroller5`; and `viewcontroller5` has a dependency on `modelproject1`. When you run or debug `workingsetA`, only the metadata for `viewcontroller2`, `viewcontroller5`, and `modelproject1` will be included in the MAR for deployment.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you run an application with metadata in the Integrated WebLogic Server. For more information, see Running a Fusion Web Application in Integrated WebLogic Server.

You will need to complete this task:

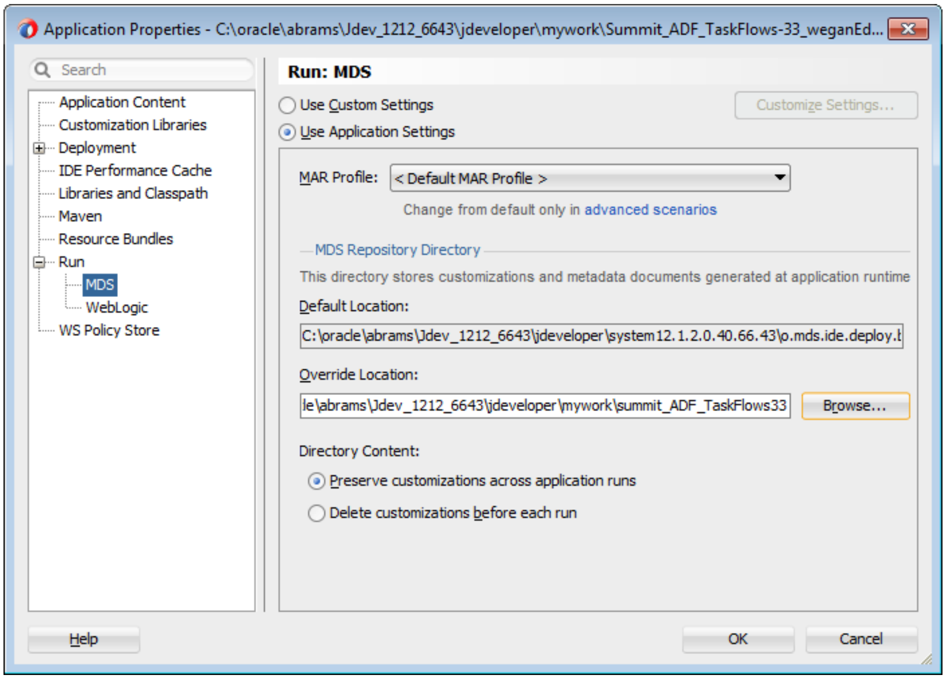
Created a MAR profile, either manually or automatically by JDeveloper.

To deploy the MAR profile to Integrated WebLogic Server:

1. In the Applications window, right-click the application and choose **Application Properties**.
2. In the Application Properties dialog, expand **Run** and choose **MDS**.
3. On the Run MDS page:
   - Select the MAR profile from the **MAR Profile** dropdown list
   - Enter a directory path in **Override Location** if you want to customize the location of the simulated MDS repository.
   - Select the **Directory Content** option. You can chose to preserve the customizations across application runs or delete customizations before each run.

   Select the MAR profile from the **MAR Profile** dropdown list. Figure 55-2 shows **Demometadata1** selected as the MAR profile.

Figure 55-2 Setting the Run MDS Options



Application Properties - C:\oracle\abrams\Jdev_1212_6643\jdeveloper\mywork\Summit_ADF_TaskFlows-33_weganEd...

Run: MDS

○ Use Custom Settings

● Use Application Settings

MAR Profile: < Default MAR Profile >

Change from default only in advanced scenarios

MDS Repository Directory

This directory stores customizations and metadata documents generated at application runtime

Default Location: C:\oracle\abrams\Jdev_1212_6643\jdeveloper\system12.1.2.0.40.66.43\o.mds.ide.deploy.t

Override Location: le\abrams\Jdev_1212_6643\jdeveloper\mywork\summit_ADF_TaskFlows33

Directory Content:

● Preserve customizations across application runs

○ Delete customizations before each run

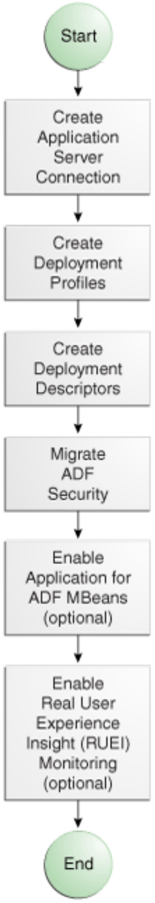
Preparing the Application

Once your application is developed, you must perform some essential tasks to deploy the Fusion web application to an integrated or standalone WebLogic Server.

Before you deploy a Fusion web application to a standalone application server, you must perform prerequisite tasks within JDeveloper to prepare the application for deployment.

Figure 55-3 show the process flow to prepare the application for deployment. After the application has been prepared and the application server has been prepared as described in Preparing the Standalone Application Server for Deployment in *Administering Oracle ADF Applications*, you can proceed to deploy the application as described in Deploying the Application.

Figure 55-3 Preparing the Application for Deployment Flow Diagram


Start
Create Application Server Connection
Create Deployment Profiles
Create Deployment Descriptors
Migrate ADF Security
Enable Application for ADF MBeans (optional)
Enable Real User Experience Insight (RUEI) Monitoring (optional)
End

How to Create a Connection to the Target Application Server

You can deploy applications to the application server via JDeveloper application server connections.

If your application involves customization using MDS, you should register your MDS repository with the application server:

- WebLogic: register the MDS into the WebLogic Domain

  For information about registering MDS in WebLogic, see Registering and Deregistering a Database-Based MDS Repository in *Administering Oracle Fusion Middleware*.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you prepare an application for deployment. See Preparing the Application.

You will need to complete this task:

Installed an application server.

To create a connection to an application server:

1. Launch the Application Server Connection wizard.

   You can:

   - In the Applications Servers window, right-click **Application Servers** and choose **New Application Server**.
   - In the New Gallery, expand **General**, select **Connections** and then **Application Server Connection**, and click **OK**.
   - In the Resources window, choose **New** > **IDE Connections > Application Server**.
2. In the Create Application Server Connection dialog, on the Usage page, select **Standalone Server** and click **Next**.
3. On the Name and Type page, enter a connection name.
4. In the **Connection Type** dropdown list, choose:
   - **WebLogic** to create a connection to Oracle WebLogic Server
5. Click **Next**.
6. On the Authentication page, enter a user name and password for the administrative user authorized to access the application server.
7. Click **Next**.
8. On the Configuration page, enter the information for your server:

   For WebLogic:

   - The Oracle WebLogic host name is the name of the WebLogic Server instance containing the TCP/IP DNS where your application (`.jar`, `.war`, `.ear`) will be deployed.
   - In the **Port** field, enter a port number for the WebLogic Server instance on which your application (`.jar`, `.war`, `.ear`) will be deployed.

     If you don't specify a port, the port number defaults to 7001.
   - In the **SSL Port** field, enter an SSL port number for the WebLogic Server instance on which your application (`.jar`, `.war`, `.ear`) will be deployed.

     Specifying an SSL port is optional. It is required only if you want to ensure a secure connection for deployment.

  If you don't specify an SSL port, the port number defaults to 7002.

  - Select **Always Use SSL** to connect to the WebLogic Server instance using the SSL port.
  - Optionally enter a **WebLogic Domain** only if WebLogic Server is configured to distinguish non-administrative server nodes by name.

9. Click **Next**.
10. Click **Next**.
11. If the SSl Signer Exchange Prompt dialog appears, click **Y**.
12. On the Test page, click **Test Connection** to test the connection.

    JDeveloper performs several types of connections tests. The JSR-88 test must pass for the application to be deployable. If the test fails, return to the previous pages of the wizard to fix the configuration.
13. Click **Finish**.

How to Create Deployment Profiles

A **deployment profile** defines the way the application is packaged into the archive that will be deployed to the target environment. The deployment profile:

- Specifies the format and contents of the archive file that will be created
- Lists the source files, deployment descriptors, and other auxiliary files that will be packaged
- Describes the type and name of the archive file to be created
- Highlights dependency information, platform-specific instructions, and other information

You need a WAR deployment profile for each web user interface project that you want to deploy in your application. If you want to package seeded customizations or place base metadata in the MDS repository, you need an application level metadata archive (MAR) deployment profile as well. For more information about seeded customizations, see Customizing Applications with MDS . If the application has customization classes, you need a JAR file for those classes and you need to add that JAR when you create the EAR file. Finally, you need an application level EAR deployment profile and you must select the projects (such as WAR and MAR profiles and customization classes JAR files) to include from a list. When the application is deployed, the EAR file will include all the projects that were selected in the deployment profile.

> **Note:**
>
> If you create your project or application using the ADF Fusion Web Application template, JDeveloper automatically creates default WAR, EAR, MAR, and JAR deployment profiles. Typically, you would not need to edit or create deployment profiles manually.

For Oracle ADF applications, you can deploy the application only as an EAR file. The WAR and MAR files that are part of the application should be included in the EAR file when you create the deployment profile.

> **Note:**
>
> If your Fusion web application has business services that you want to deploy, you will need to create a Business Component Service Interface deployment profile and deploy it. For more information about business services, see How to Deploy Web Services to Oracle WebLogic Server.

Creating a WAR Deployment Profile

You will need to create a WAR deployment profile for each web-based project you want to package into the application. Typically, the WAR profile will include the dependent data model projects it requires.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create a WAR deployment profile. For more information, see Preparing the Application.

You will need to complete this task:

Create web-based projects. If you used the ADF Fusion Web Application template, you should already have a default WAR deployment profile.

To create WAR deployment profiles for an application:

1. In the Applications window, right-click the web project that you want to deploy and choose **New > From Gallery**.

   You will create a WAR profile for each web project.

2. In the New Gallery, expand **General**, select **Deployment Profiles** and then **WAR File**, and click **OK**.

   If you don't see **Deployment Profiles** in the **Categories** tree, click the **All Features** tab.

3. In the Create Deployment Profile -- WAR File dialog, enter a name for the project deployment profile and click **OK**.

4. In the Edit WAR Deployment Profile Properties dialog, choose items in the left pane to open dialog pages in the right pane. Configure the profile by setting property values in the pages of the dialog.

   - If you have customization classes in your application, they must be loaded from the EAR-level application class loader and not from the WAR. You will later add these customization classes to the EAR.

     By default, customization classes that are implemented in the data model project are added to the data model project's WAR class path. So for each WAR, you must exclude the customization classes.

     If you created your customization classes in an extension project of the application and included them using the **Libraries and Classpath** page of the Application Properties dialog, they will be packaged at the EAR level by default. No further action is necessary to remove them from the WAR profile.

     However, if you created your customization classes in the data model project of the application, they must be removed from the WAR profile. Deselect any customization classes in the Edit WAR Deployment Profiles Properties dialog Filters page for each user interface project. If you are using a `customization.properties` file, it should also be deselected.

- You might also want to change the Java EE web context root setting. To do so, choose **General** in the left pane.

  By default, when **Use Project's Java EE Web Context Root** is selected, the associated value is set to the project name, for example, `Application1-Project1-context-root`. You need to change this if you want users to use a different name to access the application.

  If you are using custom JAAS LoginModule for authentication with JAZN, the context root name also defines the application name that is used to look up the JAAS LoginModule.

5. Click **OK** to exit the Deployment Profile Properties dialog.
6. Click **OK** again to exit the Project Properties dialog.
7. Repeat Steps 1 through 7 for all web projects that you want to deploy.

Creating a MAR Deployment Profile

If you have seeded customizations or base metadata that you want to place in the MDS repository, you need to create a MAR deployment profile.

The namespace configuration under `<mds-config>` for MAR content in the `adf-config.xml` file is generated based on your selections in the MAR Deployment Profile Properties dialog.

Although uncommon, an enterprise application (packaged in an EAR) can contain multiple web application projects (packaged in multiple WARs), but the metadata for all these web applications will be packaged into a single metadata archive (MAR). The metadata contributed by each of these individual web applications can be global (available for all the web applications) or local to that particular web application.

To avoid name conflicts for metadata with global scope, make sure that all metadata objects and elements have unique names across all the web application projects that forms part of the enterprise application.

JDeveloper creates an auto-generated MAR when the **Enable User Customizations** and **Across Sessions using MDS** options are selected in the ADF View page of the Project Properties dialog or when you explicitly specify the deployment target directory in the `adf-config.xml` file.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create a MAR deployment profile. See Preparing the Application.

You will need to complete this task:

> Create an MDS repository for your customization requirements to deploy metadata using the MAR deployment profile. For more information, see the section about managing the MDS repository in Managing the MDS Repository in *Administering Oracle Fusion Middleware*. If you used the ADF Fusion Web Application template, you should already have a default MAR deployment profile.

To create a MAR deployment profile:

1. In the Applications window, right-click the application in which you want to create a MAR profile and choose **New > From Gallery**.

   You will create a MAR profile if you want to include customizations.

2. In the New Gallery, expand **General**, select **Deployment Profiles** and then **MAR File,** and click **OK**.

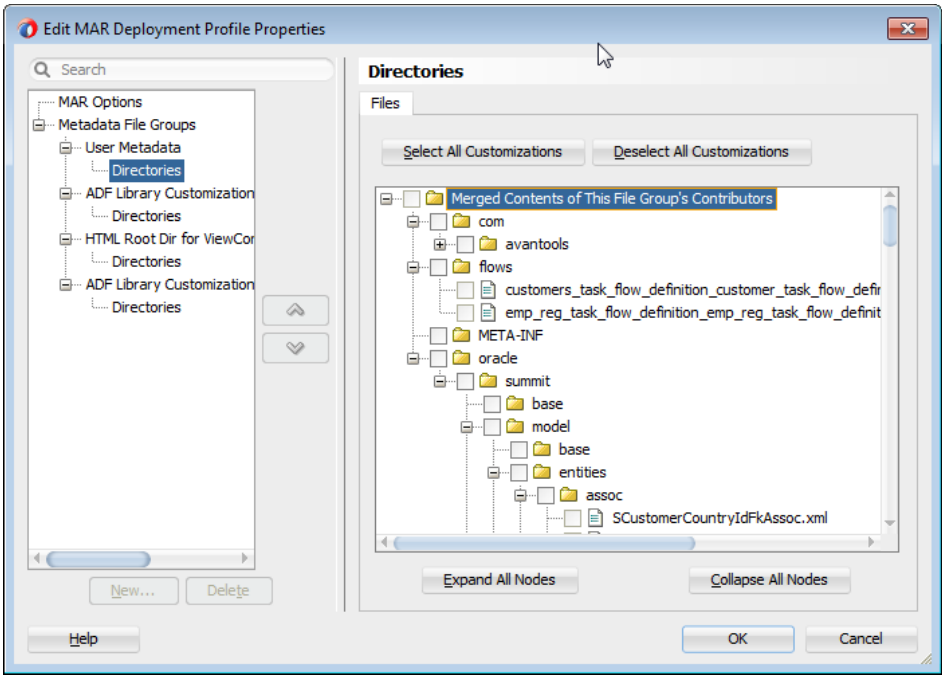
   If you don't see **Deployment Profiles** in the **Categories** tree, click the **All Features** tab.

3. In the Create Deployment Profile -- MAR File dialog, enter a name for the MAR deployment profile and click **OK**.

4. In the Edit MAR Deployment Profile Properties dialog, choose items in the left pane to open dialog pages in the right pane.

   Figure 55-4 shows a sample **User Metadata** directory tree.

Figure 55-4 Selecting Items for the MAR Deployment Profiles


Edit MAR Deployment Profile Properties
Search
MAR Options
Metadata File Groups
User Metadata
Directories
ADF Library Customization
Directories
HTML Root Dir for ViewCor
Directories
ADF Library Customization
Directories
New...
Delete
Directories
Files
Select All Customizations
Deselect All Customizations
Merged Contents of This File Group's Contributors
com
avantools
flows
customers_task_flow_definition_customer_task_flow_defir
emp_reg_task_flow_definition_emp_reg_task_flow_definit
META-INF
oracle
summit
base
model
base
entities
assoc
SCustomerCountryIdFkAssoc.xml
Expand All Nodes
Collapse All Nodes
Help
OK
Cancel

Note the following important points:

- To include all customizations, you need only create a file group with the desired directories.
- ADF Model and ADF view directories are added by default. No further action is required to package the ADF Model and ADF view customizations into the MAR. ADF view content is added to **HTML Root dir**, while ADF Model and Business Components content is added to **User Metadata**.
- To include the base metadata in the MDS repository, you need to explicitly select these directories in the dialog.

  When you select the base document to be included in the MAR, you also select specific packages. When you select one package, all the documents (including subpackages) under that package will be used. When you select a package, you cannot deselect individual items under that package.

- To include files from other than ADF Model and ADF view, users should create a new file group under **User Metadata** with the desired directories and explicitly select the required content in the Directories page.
- If a dependent ADF library JAR for the project contains seeded customizations, they will automatically be added to the MAR during MAR packaging. They will not appear in the MAR profile.
- If ADF Library customizations were created in the context of the consuming project, those customizations would appear in the MAR profile dialog by default.

5. Click **OK** to exit the Deployment Profile Properties dialog.
6. Click **OK** again to exit the Application Properties dialog.

Creating an Application Level EAR Deployment Profile

The EAR file contains all the necessary application artifacts for the application to run in the application server. If you used the ADF Fusion Web Application template, you should already have a default EAR deployment profile. For more information about the EAR file, see What You May Need to Know About EAR Files and Packaging.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create an EAR deployment profile. For more information, see Preparing the Application.

You will need to complete these tasks:

1. Add classes into a JAR file, as described in Adding Customization Classes into a JAR.
2. Create the WAR deployment profiles, as described in Creating a WAR Deployment Profile.

To create an EAR deployment profile for an application:

1. In the Applications window, right-click the application and choose **New > From Gallery**.

   You will create an EAR profile for the application.

2. In the New Gallery, expand **General**, select **Deployment Profiles** and then **EAR File,** and click **OK**.

   If you don't see **Deployment Profiles** in the Categories tree, click the **All Features** tab.

3. In the Create Deployment Profile -- EAR File dialog, enter a name for the application deployment profile and click **OK**.
4. In the Edit EAR Deployment Profile Properties dialog, choose items in the left pane to open dialog pages in the right pane. Configure the profile by setting property values in the pages of the dialog.

   Be sure that you:

- Select **Application Assembly** and then in the **Java EE Modules** list, select all the project profiles that you want to include in the deployment, including any **WAR** or **MAR** profiles.
- Select **Platform**, select the application server you are deploying to, and then select the target application connection from the **Target Connection** dropdown list.

> **Note:**
>
> If you are using a custom JAAS LoginModule for authentication with JAZN, the context root name also defines the application name that is used to look up the JAAS LoginModule.

5. If you have customization classes in your application, configure these classes so that they load from the EAR-level application class loader.

> **Note:**
>
> If you created your customization classes in an extension project of the application and included them using the **Libraries and Classpath** page of the Application Properties dialog, this step is not necessary because they will be packaged at the EAR level by default.

a. In the Edit EAR Deployment Profile Properties dialog, select **Application Assembly**.

b. Select the JAR deployment profile that contains the customization classes, and enter `lib` in the **Path in EAR** field at the bottom of the dialog.

> **Note:**
>
> You should have created this JAR as described in Adding Customization Classes into a JAR.

The JAR file containing the customization classes is added to the EAR file's `lib` directory.

> **Note:**
>
> If you have customization classes in your application, you must also make sure they are not loaded from the WAR. By default, customization classes that are added to the data model project's libraries and class path are packaged to the WAR class path.
>
> If you created your customization classes in the data model project of the consuming application, deselect any customization classes in the Edit WAR Deployment Profile Properties dialog Filters page.

6. Click **OK** again to exit the Edit EAR Deployment Profile Properties dialog.
7. Click **OK** again to exit the Application Properties dialog.

> **Note:**
>
> To verify that your customization classes are put correctly in the EAR class path, you can deploy the EAR profile to file system. Then you can examine the EAR to make sure that the customization class JAR is available in the EAR class path (the `EAR/lib` directory) and not available in the WAR class path (the `WEB-INF/lib` and `WEB-INF/classes` directories).

Adding Customization Classes into a JAR

If your application has customization classes, create a JAR that contains only these customization classes. When you create your EAR, you can add the JAR to the EAR assembly. And when you create WAR profiles for your web projects, you must make sure they don't include the customization classes JAR.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you add customization classes. For more information, see Preparing the Application.

You will need to complete this task:

> Make sure that your project has customization classes. You do not need to perform this procedure if the application does not have customization classes. For more information about customization classes, see How to Create Customization Classes.

To add customization classes into a JAR:

1. In the Applications window, right-click the data model project that contains the customization classes for which you want to create a JAR, and choose **New > From Gallery**.
2. In the New Gallery, expand **General**, select **Deployment Profiles** and then **JAR File**, and click **OK**.

   Alternatively, if you want to create a shared library, select **Shared Library JAR File** from the list of profile types, and click **OK**.

   > **Note:**
   >
   > If you don't see **Deployment Profiles** in the **Categories** tree, click the **All Features** tab.

3. In the Create Deployment Profile -- JAR File dialog, enter a name for the project deployment profile (for example, `CCArchive`) and click **OK**.
4. In the Edit JAR Deployment Profile Properties dialog, select **JAR Options**.
5. Enter the location for the JAR file.

6. Expand **Files Groups > Project Output > Filters** to list the files that can be selected to be included in the JAR.
7. In Filters page, in the **Files** tab, select the customization classes you want to add to the JAR file.

   If you are using a `customization.properties` file, it needs to be in the same class loader as the JAR file. You can select the `customization.properties` file to package it along with the customization classes in the same JAR.
8. Click **OK** to exit the Edit JAR Deployment Profile Properties dialog.
9. Click **OK** again to exit the Project Properties dialog.
10. In the Applications window, right-click the project containing the JAR deployment profile, and choose **Deploy > deployment profile > to JAR file**.

> **Note:**
>
> If this is the first time you deploy to a JAR from this deployment profile, you choose **Deploy > deployment profile** and select **Deploy to JAR** in the wizard.

Delivering Customization Classes as a Shared Library

As an alternative to adding your customization classes to the EAR, as described in Creating an Application Level EAR Deployment Profile, you can also include the customization classes in the consuming application as a shared library.

> **Note:**
>
> This procedure describes how to create and use a shared library if you are deploying to Oracle Weblogic Server.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create and use a shared library for customization classes. For more information, see Preparing the Application.

You will need to complete this task:

Add customization classes into a JAR using JDeveloper in Studio Developer role. Follow the procedure described in Adding Customization Classes into a JAR, and make sure that you select **Shared Library JAR File** as the type of archive to create.

To create and use a shared library for your customization classes:

1. In the Applications window, right-click the customization classes project, and choose **Deploy > deployment-profile**.
2. In the Deploy wizard, select **Deploy to Application Server** and click **Next**.
3. Select the appropriate application server, and click **Finish**.

This makes the shared library available on the application server. You must now add a reference to the shared library from the consuming application.

4. Open the application you want to customize in JDeveloper in the Studio Developer role.
5. In the Application Resources panel of the Applications window, double-click the **weblogic-application.xml** file.
6. In the overview editor, click the **Libraries** navigation tab.
7. On the Libraries page, in the **Shared Library References** section, click the **Add** icon.
8. In the **Library Name** field of the newly created row in the **Shared Library References** table, enter the name of the customization classes shared library you deployed, and save your changes.

Viewing and Changing Deployment Profile Properties

After you have created a deployment profile, you can view and change its properties.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you view deployment profiles. For more information, see Preparing the Application.

To view, edit, or delete a project's deployment profile:

1. In the Applications window, right-click the project which you want to view or edit and choose **Project Properties**.
2. In the Project Properties dialog, click **Deployment**.

   The **Deployment Profiles** list displays all profiles currently defined for the project.
3. In the Deployment Profiles section, select a deployment profile.
4. To edit or delete a deployment profile, click **Edit** or **Delete**.

How to Create and Edit Deployment Descriptors

Deployment descriptors are server configuration files that define the configuration of an application for deployment and that are deployed with the Java EE application as needed. The deployment descriptors that a project requires depend on the technologies the project uses and on the type of the target application server. Deployment descriptors are XML files that can be created and edited as source files, but for most descriptor types, JDeveloper provides dialogs or an overview editor that you can use to view and set properties. If you cannot edit these files declaratively, the XML file opens in the source editor for you to edit its contents.

In addition to the standard Java EE deployment descriptors (for example, `application.xml` and `web.xml`), you can also have deployment descriptors that are specific to your target application server. For example, if you are deploying to Oracle WebLogic Server, you can also have `weblogic.xml`, `weblogic-application.xml`, and `weblogic-ejb-jar.xml`.

For WebLogic Server, make sure that the application EAR file includes a `weblogic-application.xml` file that contains a reference to `adf.oracle.domain`, and that it includes an `ADFApplicationStateListener` to clean up application resources between deployment and undeployment actions. The following example shows a sample weblogic-application.xml file:

```
<weblogic-application xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.bea.com/ns/weblogic/weblogic-application.xsd"
 xmlns="http://www.bea.com/ns/weblogic/weblogic-application">
  <listener>
    <listener-class>oracle.adf.share.weblogic.listeners.
          ADFApplicationStateListener</listener-class>
  </listener>
  <listener>
    <listener-class>oracle.mds.lcm.weblogic.WLLifecycleListener</listener-class>
  </listener>
  <library-ref>
    <library-name>adf.oracle.domain</library-name>
  </library-ref>
</weblogic-application>
```

If you are deploying web services, you may need to modify your weblogic-application.xml and web.xml files as described in How to Deploy Web Services to Oracle WebLogic Server.

If you want to enable the application for Real User Experience Insight (RUEI) monitoring, you must add a parameter to the web.xml file, as described in Enabling the Application for RUEI and Click History.

During deployment, the application's security properties are written to the weblogic-application.xml file to be deployed with the application in the EAR file. For more information, see What Happens When You Configure Security Deployment Options.

Because Oracle WebLogic Server runs on the Java EE Platform, you may need to modify the application.xml and web.xml files to be compatible with the application server.

Creating Deployment Descriptors

JDeveloper automatically creates many of the required deployment descriptors for you. If they are not present, or if you need to create additional descriptors, you can use JDeveloper to create them.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create deployment descriptors. For more information, see Preparing the Application.

You will need to complete this task:

Check to see whether JDeveloper has already generated deployment descriptors.

To create a deployment descriptor:

1. In the Applications window, right-click the project for which you want to create a descriptor and choose **New > From Gallery**.
2. In the New Gallery, expand **General**, select **Deployment Descriptors** and then a descriptor type, and click **OK**.

   If you can't find the item you want, make sure that you chose the correct project, and then choose the **All Features** tab or use the **Search** field to find the descriptor. If the item is not enabled, check to make sure that the project does not already have a descriptor of that type. A project is allowed only one instance of a descriptor.

JDeveloper starts the Create Deployment Descriptor wizard and then opens the file in the overview or source editor, depending on the type of deployment descriptor you choose.

> **Note:**
>
> For EAR files, do not create more than one deployment descriptor file of the same type per application or workspace. These files can be assigned to projects, but have application workspace scope. If multiple projects in an application have the same deployment descriptor, the one belonging to the launched project will supersede the others. This restriction applies to `application.xml`, `weblogic-jdbc.xml`, `jazn-data.xml`, and `weblogic.xml`.
>
> The best place to create an application level descriptor is in the **Descriptors** node of the Application Resources panel in the Applications window. This ensures that the application is created with the correct descriptors.
>
> Application level descriptors created in the project will be ignored at runtime. Only the application resources descriptors or descriptors generated at the EAR level will be used by the runtime.

Viewing or Modifying Deployment Descriptor Properties

After you have created a deployment descriptor, you can change its properties by using JDeveloper dialogs or by editing the file in the source editor. The deployment descriptor is an XML file (for example, `application.xml`) typically located under the Application Sources node.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you view deployment descriptors. For more information, see Preparing the Application.

To view or change deployment descriptor properties:

1. In the Applications window or in the Application Resources panel, double-click the deployment descriptor.
2. In the editor window, select either the **Overview tab** or the **Source tab**, and configure the descriptor by setting property values.

   If the overview editor is not available, the file opens in the source editor.

Configuring the application.xml File for Application Server Compatibility

You may need to configure your `application.xml` file to be compliant with your Java EE version.

> **Note:**
>
> Typically, your project has an `application.xml` file that is compatible and you would not need to perform this procedure.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you configure `application.xml`. For more information, see Preparing the Application.

To configure the application.xml file:

1. In the Applications window, right-click the application and choose **New > From Gallery**.
2. In the New Gallery, expand **General**, select **Deployment Descriptors** and then **Java EE Deployment Descriptor Wizard**, and click **OK**.
3. In the Create Java EE Deployment Descriptor dialog, on the Select Descriptor page, select **application.xml** and click **Next**.
4. On the Select Version page, select **5.0** to be compatible with Java EE 5.0 or higher, and click **Next**.
5. On the Summary page, click **Finish**.
6. Edit the `application.xml` file with the appropriate values.

Configuring the web.xml File for Application Server Compatibility

You may need to configure your `web.xml` file to be compliant with your Java EE version. See web.xml.

> **Note:**
>
> Typically, your project has a `web.xml` file that is compatible and you would not need to perform this procedure. JDeveloper creates a starter `web.xml` file when you create a project.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you configure `web.xml`. See Preparing the Application.

To configure the web.xml file:

1. In the Applications window, right-click the project in which you want to configure a `web.xml` file and choose **New > From Gallery**.
2. In the New Gallery, expand **General**, select **Deployment Descriptors** and then **Java EE Deployment Descriptor**, and click **OK**.
3. In the Create Java EE Deployment Descriptor dialog, on the Select Descriptor page, select **web.xml** and click **Next**.

4. On the Select Version page, select the version you want to use and click **Next**.

   Version 3.0 is compatible with Java EE 6.0 or higher, and version 2.5 is compatible with Java EE 5.0.

5. On the Summary page, click **Finish**.

Enabling the Application for RUEI and Click History

Real User Experience Insight (RUEI) is a web-based utility to report on real-user traffic requested by, and generated from, your network. It measures the response times of pages and transactions at the most critical points in the network infrastructure. Session diagnostics allow you to perform root-cause analysis.

RUEI enables you to view server and network times based on the real-user experience, to monitor your Key Performance Indicators (KPIs) and Service Level Agreements (SLAs), and to trigger alert notifications on incidents that violate their defined targets. You can implement checks on page content, site errors, and the functional requirements of transactions. Using this information, you can verify your business and technical operations. You can also set custom alerts on the availability, throughput, and traffic of all items identified in RUEI.

For information about RUEI, see the *Oracle Real User Experience Insight documentation* at `http://docs.oracle.com/en/enterprise-manager/?tab=3`.

Click History is a feature that is built into ADF that reports user actions on UI components such as button clicks and mouse scrolling. Click History needs to be enabled on the WebLogic server as well as within the application using the `web.xml` and `weblogic.xml` files.

For information about Click History and how to enable it in WebLogic server, see Click History in *Administering Oracle ADF Applications*.

To enable RUEI and Click History in the web.xml file:

1. In the Applications window, expand **WEB-INF** and double-click **web.xml**.
2. In the overview editor, add the `context-param` tag to the `web.xml` file, as shown in the following example.

```
<context-param>
  <description>This parameter notifies ADF Faces that the
               ExecutionContextProvider service provider is enabled.
               When enabled, this will start monitoring and aggregating
               user activity information for the client initiated
               requests. By default this param is not set or is false.
  </description>
  <param-name>
        oracle.adf.view.faces.context.ENABLE_ADF_EXECUTION_CONTEXT_PROVIDER
  </param-name>
  <param-value>true</param-value>
</context-param>
```

For Click History, in addition to adding the parameter to the `web.xml` file, you also need to define the `odl.clickhistory.webapp` library in the `weblogic.xml` file.

To finish enabling Click History using the weblogic.xml file:

1. In the Applications window, expand **WEB-INF** and double-click **weblogic.xml**.

2. In the overview editor, add the library entry to the `weblogic.xml` file, as shown in the following example.

   ```
   <library-ref>
          <library-name>odl.clickhistory.webapp</library-name>
   </library-ref>
   ```

If there is an `EndUserMonitoringService` service provider available, you can use it to log performance metrics and data by registering it in your application. The service provider should have been implemented by extending the `EndUserMonitoringService` class. For more information about this class, see the Javadoc.

Note that the context-parameter `oracle.adf.view.faces.context.ENABLE_ADF_EXECUTION_CONTEXT_PROVIDER` needs to be set when an `EndUserMonitoringServive` provider is used within an ADF application.

To enable End User Monitoring in an application:

1. Verify that an `EndUserMonitoringService` service provider has been implemented.
2. Navigate to the application's `<`*`application_home`*`>/src/META-INF/services` folder using your operating system tools.

   If the `services` folder does not exist, create one.
3. In the `<`*`application_home`*`>/src/META-INF/services` folder, create a file named `oracle.adf.view.rich.monitoring.`*`EndUserMonitoringService`*

   where *`EndUserMonitoringService`* is the name of the service interface.
4. Open the file in an editor and add this entry: *`<package>.MyEndUserMonitoringService`*

   where *`package`* is the Java package name where the service provider resides and *`MyEndUserMonitoringService`* is the name of the `EndUserMonitoringService` service provider.

How to Deploy Applications with ADF Security Enabled

If you are developing an application in JDeveloper using Integrated WebLogic Server, application security deployment properties are configured by default, which means that the application and security credentials and policies will be overwritten each time you redeploy for development purposes. However, the application security deployment properties are the same for Integrated WebLogic Server and the standalone WebLogic Server.

You can change the default behavior in the Application Properties dialog, as described in How to Configure, Deploy, and Run a Secure Application in JDeveloper.

Applications That Will Run Using Oracle Single Sign-On (SSO)

Before you can deploy and run the web application with ADF Security enabled on the application server, the administrator of the target server must configure the domain-level `jps-config.xml` file for the Oracle Access Manager (OAM) security provider. To complete this configuration task, you can use Oracle WebLogic Scripting Tool (WLST) provided with the JDeveloper install. For details about running WLST (with command `addOAMSSOProvider(loginuri, logouturi, autologinuri)`), see Integrating Access Manager With Web Applications Using Oracle ADF Security and the OPSS SSO

Framework and Integrating Oracle ADF Applications with Access Manager SSO of the *Administrator's Guide for Oracle Access Management*.

Running the addOAMSSOProvider() command ensures that the ADF Security framework defers to the OAM service provider to clear the ObSSOCookie token. OAM uses this token to save the identity of authenticated users and, unless it is cleared during logout, the user will be unable to log out.

After the system administrator runs the command on the target server, the domain jps-config.xml file will contain the following security provider definition that is specific for ADF Security:

```
<propertySet name="props.auth.uri">
    <property name="login.url.FORM" value="/${app.context}/adfAuthentication"/>
    <property name="logout.url" value=""/>
</propertySet>
```

Additionally, the authentication type required by SSO is CLIENT-CERT. The web.xml authentication configuration for the deployed application must specify the <auth-method> element as one of the following CLIENT-CERT types.

WebLogic supports two types of authentication methods:

- For FORM-type authentication method, specify the elements like this:

```
<login-config>
  <auth-method>CLIENT-CERT,FORM</auth-method>
  <realm-name>myrealm</realm-name>
  <form-login-config>
     <form-login-page>/login.html</form-login-page>
     <form-error-page>/error.html</form-error-page>
  </form-login-config>
</login-config>
```

- For BASIC-type authentication method, specify the elements like this:

```
<login-config>
   <auth-method>CLIENT-CERT,BASIC</auth-method>
   <realm-name>myrealm</realm-name>
</login-config>
```

You can configure the web.xml file either before or after deploying the web application. For further details about setting up the authentication method for Single Sign-On, see Managing Access Manager SSO, Policies, and Testing of the *Administrator's Guide for Oracle Access Management*.

Configuring Security for Weblogic Server

In a development environment, JDeveloper will automatically migrate application level credentials, identities, and policies to the standalone WebLogic Server instance only if the server is set up to be in development mode. Integrated WebLogic Server is set up in development mode by default. You can set up a standalone WebLogic Server to be in development mode during Oracle WebLogic Server domain creation using the Oracle Fusion Middleware Configuration Wizard. For more information about configuring Oracle WebLogic Server domains, see Introduction to WebLogic Domains in *Creating WebLogic Domains Using the Configuration Wizard*.

JDeveloper will not migrate application level security credentials to WebLogic Server set up in production mode. Typically, in a production environment, administrators will use Enterprise Manager or WLST to deploy an application, including its security requirements.

When you deploy an application to WebLogic Server, credentials (in the `cwallet.sso` file) and security policies (in the `jazn-data.xml` file) will either overwrite or merge with the WebLogic Server's domain-level credential store, depending on whether an option in `weblogic-application.xml` is set to `OVERWRITE` or `MERGE`. In production-mode WebLogic Server, to avoid security risks, only `MERGE` is allowed. For development-mode WebLogic Server, you can set to `OVERWRITE` to test user names and passwords. You can also set the option by running `setDomainEnv.cmd` or `setDomainEnv.sh` with the following option added to the command (usually located in `ORACLE_HOME/user_projects/domains/`*`MyDomain`*`/bin`).

For `setDomainEnv.cmd`:

```
set EXTRA_JAVA_PROPERTIES=-Djps.app.credential.overwrite.allowed=true
    %EXTRA_JAVA_PROPERTIES%
```

For `setDomainEnv.sh`:

```
EXTRA_JAVA_PROPERTIES="-Djps.app.credential.overwrite.allowed=true
     ${EXTRA_JAVA_PROPERTIES}"
export EXTRA_JAVA_PROPERTIES
```

If the Administration Server is already running, you must restart it for this setting to take effect.

You can check to see whether WebLogic Server is in production mode by using the Oracle WebLogic Server Administration Console or by verifying the following line in WebLogic Server's `config.xml` file:

```
<production-mode-enabled>true</production-mode-enabled>
```

By default, JDeveloper sets the application's credential, identities, and policies to `OVERWRITE` mode. That is, the **Application Policies**, **Credentials**, and **Users and Groups** options are selected by default in the Application Properties dialog Deployment page. However, an application's credentials will be migrated only if the target WebLogic Server instance is set to development mode with `-Djps.app.credential.overwrite.allowed=true`

Policy migration only works in development-mode. Identity migration only works when using JDeveloper to directly deploy to WebLogic Server regardless of whether it is in development or production-mode.

When your application is ready for deployment to a production environment, you should remove the identities from the `jazn-data.xml` file or disable the migration of identities by deselecting **Users and Groups** from the Application Properties dialog. Application credentials must be manually migrated outside of JDeveloper.

> **Note:**
>
> Before you migrate the `jazn-data.xml` file to a production environment, check that the policy store does not contain duplicate permissions for a grant. If a duplicate permission (one that has the same name and class) appears in the file, the administrator migrating the policy store will receive an error and the migration of the policies will be halted. You should manually edit the `jazn-data.xml` file to remove any duplicate permissions from a grant definition.

For more information about migrating application credentials and other `jazn-data` user credentials, see the Configuring the OPSS Security Store chapter in *Securing Applications with Oracle Platform Security Services*.

Applications with JDBC URL for WebLogic

If your application has components that use JDBC URL connections, the connection user names and passwords are also stored in the application level credential and policy stores. For the deployed application to be able to connect to the database using the JDBC URL, these credentials and policies must be migrated. That is, if WebLogic Server is in production mode, system administrators must migrate this security information. If WebLogic Server is in development mode, it must have domain-level credential and policy stores set to `OVERWRITE` to allow the migration of security information.

Applications with JDBC Data Source for WebLogic

If your application uses application level JDBC data sources with password indirection for database connections, you may need to create credential maps in WebLogic Server to enable the database connection. See Creating a JDBC Data Source for Oracle WebLogic Server in *Administering Oracle ADF Applications*.

How to Replicate Memory Scopes in a Clustered Environment

If you are deploying an application that is intended to run in a clustered environment, you need to ensure that all managed beans with a lifespan longer than one request are serializable, and that Oracle ADF is aware of changes to managed beans stored in ADF scopes (view scope and page flow scope).

For more information, see How to Set Managed Bean Memory Scopes in a Server-Cluster Environment.

How to Enable the Application for ADF MBeans

A Fusion web application uses many XML files for setting configuration information. Three of these configuration files have ADF MBean counterparts that are deployed with the application. After the application has been deployed, you can change configuration properties by accessing the ADF MBeans using the Enterprise Manager Fusion Middleware Control MBean browser.

To enable ADF MBeans, register them in the web.xml file. The following example shows a web.xml file with listener entries for connections, configuration, and business components.

```
<listener>
   <listener-class>
        oracle.adf.mbean.share.connection.ADFConnectionLifeCycleCallBack
   </listener-class>
</listener>
<listener>
   <listener-class>
        oracle.adf.mbean.share.config.ADFConfigLifeCycleCallBack</listener-class>
</listener>
<listener>
   <listener-class>
        oracle.bc4j.mbean.BC4JConfigLifeCycleCallBack</listener-class>
</listener>
```

Additionally, you must configure a writable MDS repository to enable post-deployment configuration of connections.xml using ADF Connections MBeans and bc4j.xcfg using ADF Business Components MBeans.

MDS configuration entries in the adf-config.xml file for a database-based MDS are shown in the example below. For more information about configuring MDS, see Changing MDS Configuration Attributes for Deployed Applications in *Administering Oracle Fusion Middleware*.

```
<adf-mds-config xmlns="http://xmlns.oracle.com/adf/mds/config">
   <mds-config xmlns="http://xmlns.oracle.com/mds/config" version="11.1.1.000">
      <persistence-config>
         <metadata-store-usages>
            <metadata-store-usage
                default-cust-store="true" deploy-target="true" id="myStore">
            </metadata-store-usage>
         </metadata-store-usages>
      </persistence-config>
   </mds-config>
</adf-mds-config>
```

In a production environment, an MDS repository that uses a database is required. You can use JDeveloper, Enterprise Manager Fusion Middleware Control or scripts to switch from a file-based repository to a database MDS repository.

Additionally, if several applications are sharing the same MDS configuration, each application can achieve distinct customization layers by defining a adf:adf-properties-child property in the adf-config.xml file. JDeveloper automatically generates this entry when creating applications. If your adf-config.xml file does not have this entry, add it to the file with code similar to this one:

```
<adf:adf-properties-child xmlns="http://xmlns.oracle.com/adf/config/properties">
     <adf-property name="adfAppUID" value="Application3-4434"/>
     <adf-property name="partition_customizations_by_application_id"
            value="true"/>
</adf:adf-properties-child>
```

The value attribute is either generated by JDeveloper or you can set it to any unique identifier within the server farm where the application is deployed. This value can be set to the value attribute of the adfAppUID property.

When `adf-property name` is set to `adfAppUid`, then the corresponding `value` property should be set to the name of the application. By default, JDeveloper generates the `value` property using the application's package name. If the package name is not specified, JDeveloper generates the `value` property by using the workspace name and a four digit random number.

For more information about configuring ADF applications using ADF MBeans, see Modifying ADF Application Configurations Using MBeans in *Administering Oracle ADF Applications*.

What You May Need to Know About JDBC Data Source for Oracle WebLogic Server

ADF applications can use either a JDBC data source or a JDBC URL for database connections. You use the Oracle WebLogic Server Administration Console to configure a JDBC data source. For more information about database access, see Configuring Your Application Module Database Connection.

> **Note:**
>
> Fusion web applications are not compatible with data sources defined with the JDBC XA driver. When creating a data source on Oracle WebLogic Server, be sure to change the Fusion web application data source's JDBC driver from “Oracle's Driver (Thin XA)" to “Oracle's Driver (Thin)". Because XA data sources close all cursors upon commit, random JBO-27122 and closed statement errors may result when running the Fusion web application with an XA data source.

The **application module** in the data model project of a Fusion web application can be configured to use a JDBC URL connection type, a JDBC data source connection type, or a combination of both types. By default, Fusion web application modules use a JDBC URL connection. A component that uses a JDBC URL will attempt to connect directly to the database using the JDBC URL, and it will ignore any JDBC data sources (global or application level) that are available in WebLogic Server. For more information about migrating JDBC URL security information (user names and passwords) from the application to WebLogic Server, see How to Deploy Applications with ADF Security Enabled.

A Fusion web application can use a JDBC data source to connect to the database. A JDBC data source has three types: global, application level, and application level with password indirection. You generally set up a global JDBC data source in WebLogic Server. Any application that requires access to that database can use that JDBC data source. An application can also include application level JDBC data sources. When the application is packaged for deployment, if the **Auto Generate and Synchronize weblogic-jdbc.xml Descriptor During Deployment** option is selected, JDeveloper creates a *`connection_name`*`-jdbc.xml` file for each connection that was defined. Each connection's information is written to the corresponding *`connection_name`*`-jdbc.xml` file (entries are also changed in `weblogic-application.xml` and `web.xml`). When the application is deployed to WebLogic Server, the server looks for application level data source information before it looks for the global data source.

If the application is deployed with password indirection set to `true`, WebLogic Server will look for the *`connection_name`*`-jdbc.xml` file for user name information and it will then attempt to locate application level credential maps for these user names to obtain the password. If you are using JDeveloper to directly deploy the application to WebLogic Server, JDeveloper automatically creates the credential map and populates the map to the server using an MBean call.

However, if you are deploying to an EAR file, JDeveloper will not be able to make the MBean call to WebLogic Server. You must set up the credential maps using the Oracle WebLogic Administration Console. Even if you have a global JDBC data source set up, if you do not also have credential mapping set up, WebLogic Server will not be able to map the credentials with passwords and the connection will fail.

Once the data source has been created in Oracle WebLogic Server, it can be used by an application module. See Preparing the Standalone Application Server for Deployment in *Administering Oracle ADF Applications*.

Deploying the Application

Once the target environment is set up and the ADF application is prepared for deployment, the final step is to deploy the application to the target environment.

You can use JDeveloper to deploy ADF applications directly to the standalone application server or you can create an archive file and use other tools to deploy to the application server.

> **Note:**
>
> Before you begin to deploy applications that use Oracle ADF to the standalone application server, you need to prepare the application server environment by performing tasks such as installing the ADF runtime and creating and extending domains or cells. See Preparing the Standalone Application Server for Deployment.

Figure 55-5 show the process flow for deploying an application and also for deploying customizations to the target standalone application server.

Figure 55-5 Application Deployment Flow Diagram

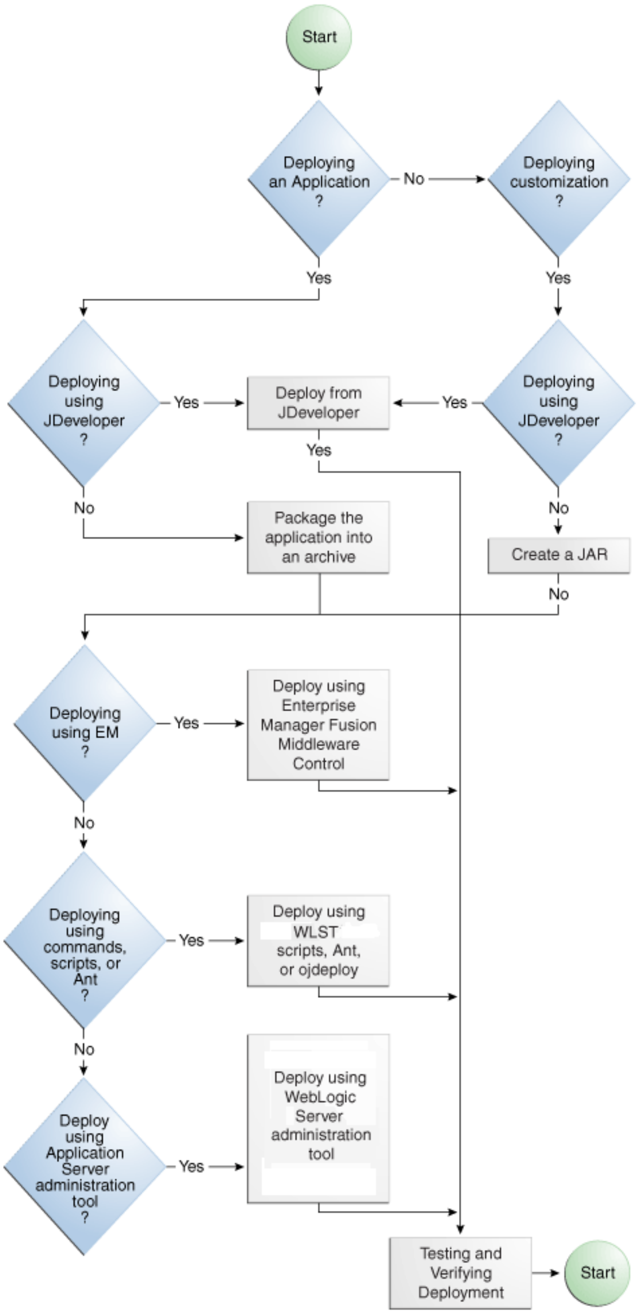

Start
Deploying an Application ?
No
Deploying customization ?
Yes
Yes
Deploying using JDeveloper ?
Yes
Deploy from JDeveloper
Yes
Deploying using JDeveloper ?
Yes
No
Package the application into an archive
No
Create a JAR
No
Deploying using EM ?
Yes
Deploy using Enterprise Manager Fusion Middleware Control
No
Deploying using commands, scripts, or Ant ?
Yes
Deploy using WLST scripts, Ant, or ojdeploy
No
Deploy using Application Server administration tool ?
Yes
Deploy using WebLogic Server administration tool
Testing and Verifying Deployment
Start

Table 55-1 describes some common deployment techniques that you can use during the application development and deployment cycle. The deployment techniques are listed in order from deploying on development environments to deploying on production environments. It is likely that in the production environment, the system administrators deploy applications by using Enterprise Manager Fusion Middleware Control or scripts.

Table 55-1 Deployment Techniques for Development or Production Environments

Deployment Technique	Environment	When to Use
Run directly from JDeveloper	Test or Development	When you are developing your application. You want deployment to be quick because you will be repeating the editing and deploying process many times. JDeveloper contains Integrated WebLogic Server, on which you can run and test your application.
Use JDeveloper to directly deploy to the target application server	Test or Development	When you are ready to deploy and test your application on an application server in a test environment. On the test server, you can test features (such as LDAP and Oracle Single Sign-On) that are not available on the development server. You can also use the test environment to develop your deployment scripts, for example, using Ant.
Use JDeveloper to deploy to an EAR file, then use the target application server's tools for deployment	Test or Development	When you are ready to deploy and test your application on an application server in a test environment. As an alternative to deploying directly from JDeveloper, you can deploy to an EAR file and then use other tools to deploy to the application server. On the test server, you can test features (such as LDAP and Oracle Single Sign-On) that are not available on the development server. You can also use the test environment to develop your deployment scripts, for example, using Ant.
Use Enterprise Manager or WLST to deploy applications	Production	When your application is in a test and production environment. In production environments, system administrators usually use Enterprise Manager Fusion Middleware Control or run WLST to deploy applications.

Any necessary MDS repositories must be registered with the application server. If the MDS repository is a database, the repository maps to a data source with MDS-specific requirements.

If you are deploying the application to Oracle WebLogic Server, make sure to target this data source to the WebLogic Administration Server and to all Managed Servers to which you are deploying the application. For information about registering MDS, see Registering and Deregistering a Database-Based MDS Repository in *Administering Oracle Fusion Middleware*.

If you are using the application server's administrative console or scripts to deploy an application packaged as an EAR file that requires MDS repository configuration in `adf-config.xml`, you must run the `getMDSArchiveConfig` command to configure MDS before deploying the EAR file. MDS configuration is required if the EAR file contains a MAR file or if the application is enabled for DT@RT (Design Time At RunTime).

For information about WLST commands, see the *WLST Command Reference for Infrastructure Components*.

If you plan to configure ADF connection information, **ADF Business Components** information, or `adf-config.xml` using ADF MBeans after the application has been deployed, make sure that the application is configured with MDS and have the MBean listeners enabled in the `web.xml` file. See How to Enable the Application for ADF MBeans.

> **Note:**
>
> If your Fusion web application has business services that you want to deploy to WebLogic Server, see How to Deploy Web Services to Oracle WebLogic Server.

How to Deploy to the Application Server from JDeveloper

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you deploy an application. See Deploying the Application.

You will need to complete this task:

Create an application level deployment profile that deploys to an EAR file as described in Creating an Application Level EAR Deployment Profile.

> **Note:**
>
> When you are deploying to Oracle WebLogic Server from JDeveloper, ensure that the HTTP Tunneling property is enabled in the Oracle WebLogic Server Administration Console. This property is located under **Servers > *ServerName* > Protocols**. ***ServerName*** refers to the name of Oracle WebLogic Server.

> **Note:**
>
> JDeveloper does not support deploying applications to individual Managed Servers that are members of a cluster. You may be able to target one or more Managed Servers within a cluster using the Oracle WebLogic Server Administration Console or other Oracle WebLogic tools; however, the cluster can be negatively affected. For more information about deploying to Oracle WebLogic Server clusters, see Deploying an Application to Managed Servers in *Administering Oracle Fusion Middleware*.

To deploy to the target application server from JDeveloper:

1. In the Applications window, right-click the application and choose **Deploy > *deployment profile***.
2. In the Deploy wizard, on the Deployment Action page, select **Deploy to Application Server** and click **Next**.
3. On the Select Server page, select the application server connection, and click **Next**.
4. If you are deploying to a WebLogic Server instance, the WebLogic Options page appears. Select a deployment option and click **Next**.

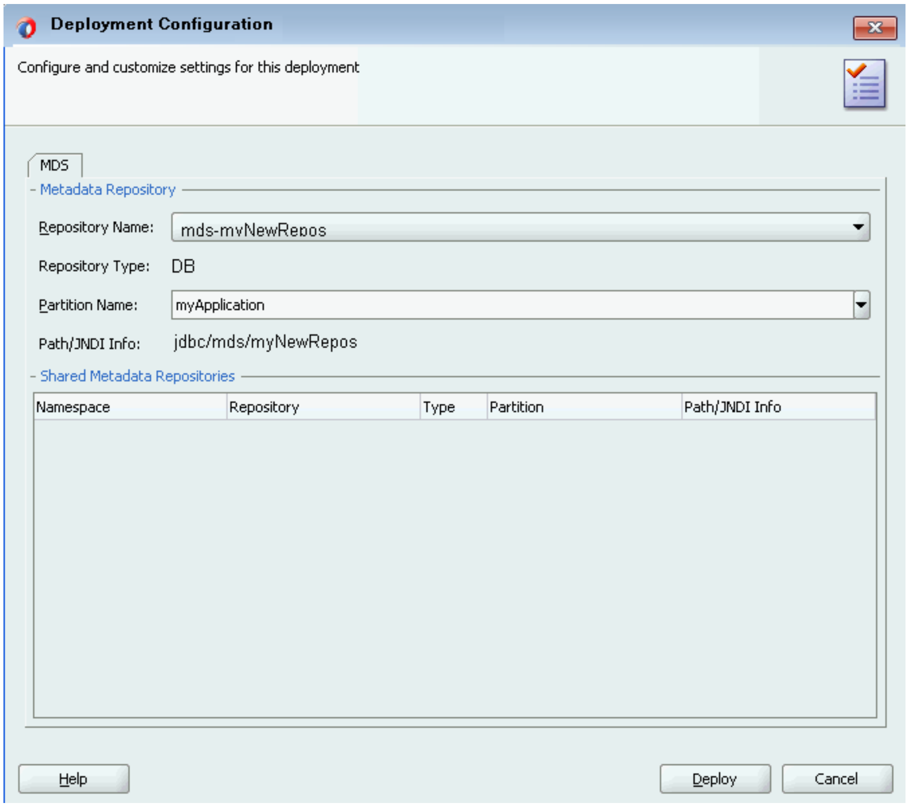
> **Note:**
>
> If you are deploying an ADF application, do not use the **Deploy to all instances in the domain** option.

5. Click **Finish**.

   During deployment, you can see the process steps displayed in the deployment Log window. You can inspect the contents of the modules (archives or exploded EAR) being created by clicking on the links that are provided in the log window. The archive or exploded EAR file will open in the appropriate editor or directory window for inspection.

   If the `adf-config.xml` file in the EAR file requires MDS repository configuration, the Deployment Configuration dialog appears for you to choose the target metadata repository or shared metadata repositories, as shown in Figure 55-6. The **Repository Name** dropdown list allows you to choose a target metadata repository from a list of metadata repositories registered with the Administration Server. The **Partition Name** dropdown list allows you to choose the metadata repository partition to which the application's metadata will be imported during deployment. You can use Oracle WebLogic Scripting Tool (WLST) and Oracle WebLogic Server Administration Tool to configure and register MDS. For more information about managing the MDS Repository, see Managing the MDS Repository in *Administering Oracle Fusion Middleware*.

Figure 55-6 MDS Configuration and Customization for Deployment



Deployment Configuration

Configure and customize settings for this deployment

MDS

Metadata Repository

Repository Name: mds-myNewRepos

Repository Type: DB

Partition Name: myApplication

Path/JNDI Info: jdbc/mds/myNewRepos

Shared Metadata Repositories

Namespace	Repository	Type	Partition	Path/JNDI Info

> **Note:**
>
> If you are deploying a Java EE application, click the application menu next to the Java EE application in the Applications window.

For more information on creating application server connections, see How to Create a Connection to the Target Application Server.

How to Create an EAR File for Deployment

You can also use the deployment profile to create an archive file (EAR file). You can then deploy the archive file using Enterprise Manager, Oracle WebLogic Scripting Tool (WLST), or Oracle WebLogic Server Administration Console.

Although a Fusion web application is encapsulated in an EAR file (which usually includes WAR, MAR, and JAR components), it may have parts that are not deployed with the EAR. For instance, ADF Business Services can be deployed as a JAR. For more information about business services, see How to Deploy Web Services to Oracle WebLogic Server.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create an EAR file for deployment. For more information, see Deploying the Application.

You will need to complete this task:

Create an application level deployment profile that deploys to an EAR file as described in Creating an Application Level EAR Deployment Profile.

To create an EAR archive file:

1. In the Applications window, right-click the application containing the deployment profile, and choose **Deploy > *deployment profile***.
2. In the Deploy wizard, on the Deployment Action page, select **Deploy to EAR** and click **Finish**.

   If an EAR file is deployed at the application level, and it has dependencies on a JAR file in the data model project and dependencies on a WAR file in the user interface project, then the files will be located in the following directories by default:

   - `*ApplicationDirectory*/deploy/*EARdeploymentprofile*.EAR`
   - `*ApplicationDirectory*/*ModelProject*/deploy/*JARdeploymentprofile*.JAR`
   - `*ApplicationDirectory*/*ViewControllerProject*/deploy/*WARdeploymentprofile*.WAR`

> **Tip:**
>
> Choose **View** >**Log** to see messages generated during creation of the archive file.

How to Deploy New Customizations Applied to ADF Library

If you have created new customizations for an ADF Library, you can use the MAR profile to deploy these customizations to any deployed application that consumes that ADF Library. For instance, suppose `applicationA`, which consumes `ADFLibraryB`, is deployed to a standalone application server. Later on, when new customizations are added to `ADFLibraryB`, you only need to deploy the updated customizations into `applicationA`. You do not need to repackage and redeploy the whole application nor do you need to manually patch the MDS repository.

> **Note:**
>
> This procedure is for applying ADF Library customizations changes to an application that has already been deployed to a standalone application server. It is not for the initial packaging of customizations into a MAR that will eventually be a part of an EAR. For information about initial packaging of the customization using a MAR, see Creating a MAR Deployment Profile.

To deploy ADF Library customizations, create a new MAR profile and only include the customizations to be deployed and then use JDeveloper to:

- Deploy the customizations directly into the MDS repository in the standalone application server.
- Deploy the customizations to a JAR. And then import the JAR into the MDS repository using tools such as the Fusion Middleware Control.

Exporting Customization to a Deployed Application

You can export the customizations directly from JDeveloper into the MDS repository for the deployed application on the standalone application server.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you export customizations directly into the application server. For more information, see Deploying the Application.

You will need to complete this task:

Create new customizations to the ADF Library. For more information on customizations, see How to Customize ADF Library Artifacts in JDeveloper.

To export the customizations directly into the application server:

1. In the Applications window, right-click the application and choose **Deploy > metadata**.
2. In the Deploy metadata dialog, on the Deployment Action page, select **Export to a Deployed Application** and click **Next**.

   If the MAR profile is included in the EAR profile of any application, **Export to a Deployed Application** will be dimmed and disabled.

3. On the Application Server page, select the application server connection and click **Next**.
4. For WebLogic Server, the Server Instance page appears. In this page, select the server instance where the deployed application is located and click **Next**.
5. On the Deployed Application page, select the application you want to apply the customizations to and click **Next**.
6. On the Sandbox Instance page, if you want to deploy to a sandbox, select **Deploy to an associated sandbox**, choose the sandbox instance, and click **Next**.
7. On the Summary page, verify the information and click **Finish**.

Deploying Customizations to a JAR

When you deploy the ADF Library customizations to a JAR, you are packaging the contents as defined by the MAR profile.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you deploy customizations as a JAR. For more information, see Deploying the Application.

You will need to complete this task:

> Create new customizations to the ADF Library. For more information on customizations, see How to Customize ADF Library Artifacts in JDeveloper.

To deploy the customizations as a JAR:

1. In the Applications window, right-click the application and choose **Deploy > metadata**.
2. In the Deploy Metadata dialog, on the Deployment Action page, select **Deploy to MAR**.
3. On the Summary page, click **Finish**.
4. Use Enterprise Manager Fusion Middleware Control or the application server's administration tool to import the JAR into the MDS repository.

For information about Fusion Middleware Control, see Getting Started Using Oracle Enterprise Manager Fusion Middleware Control in *Administering Oracle Fusion Middleware*

What You May Need to Know About ADF Libraries

An ADF Library is a JAR file that contains JAR services registered for ADF components such as ADF task flows, pages, or application modules. If you want the ADF components in a project to be reusable, you create an ADF Library deployment profile for the project and then create an ADF Library JAR based on that profile.

An application or project can consume the ADF Library JAR when you add it using the Resources window or manually by adding it to the library classpath. When the ADF Library JAR is added to a project, it will be included in the project's WAR file if the **Deployed by Default** option is selected.

For more information, see Reusing Application Components.

What You May Need to Know About EAR Files and Packaging

When you package a Fusion web application into an EAR file, it can contain the following:

- WAR files: Each web-based view controller project should be packaged into a WAR file.
- MAR file: If the application has customizations that are deployed with the application, it should be packaged into a MAR.
- ADF Library JAR files: If the application consumes ADF Library JARs, these JAR files may be packaged within the EAR.
- Other JAR files: The application may have other dependent JAR files that are required. They can be packaged within the EAR.

How to Deploy the Application Using Scripts and Ant

You can deploy the application using commands and automate the process by putting those commands in scripts. The `ojdeploy` command can be used to deploy an application without JDeveloper. You can also use Ant scripts to deploy the application. JDeveloper has a feature to help you build Ant scripts. Depending on your requirements, you may be able to integrate regular scripts with Ant scripts.

For more information about commands, scripts, and Ant, see Deploying Using Scripts and Ants in *Administering Oracle ADF Applications*.

How to Deploy ADF Faces Library JARs with the Application

You can develop applications that only use ADF Faces without using ADF Model. In that case, you can deploy the ADF Faces JARs with the application into the application server.

Before you begin:

Create an application with a web-based project. If you used the Fusion Web Application (ADF) template, you should already have a default WAR deployment profile.

To deploy ADF Faces JARs as part of the application:

1. In the Applications window, right-click the web project that you want to deploy and choose **Project Properties**.
2. In the Project Properties dialog, select **Deployment** and then the deployment profile and click **Edit**.

   If your project does not have a WAR deployment profile, follow Steps 1 to 3 in Creating a WAR Deployment Profile, to create a new WAR deployment profile.
3. Navigate to **File Groups > WEB-INF/lib > Contributors** and select **ADF Faces Runtime 11**.
4. Click **OK** to exit the Deployment Profile Properties dialog.
5. Click **OK** again to exit the Project Properties dialog.

6. Deploy your application as described in this chapter either directly using JDeveloper or by creating a EAR and deploying it using the Enterprise Manager Fusion Middleware Control, the Oracle WebLogic Server Administration Console, or other methods.

   Make sure your application-level EAR contains the WAR file as described in Creating an Application Level EAR Deployment Profile.

What You May Need to Know About JDeveloper Runtime Libraries

When an application is deployed, it includes some of its required libraries with the application. The application may also require shared libraries that have already been loaded to WebLogic Server as JDeveloper runtime libraries. It may be useful to know which JDeveloper libraries are packaged within which WebLogic Server shared library. For a listing of the contents of the JDeveloper runtime libraries, see ADF Runtime Libraries in *Administering Oracle ADF Applications*.

Postdeployment Configuration

After deployment, you can perform configuration tasks such migrate a Fusion web application from one WebLogic Server to another and enable an application for ADF MBeans.

After you have deployed your application to WebLogic Server, you can perform configuration tasks.

How to Migrate an Application

If you want to migrate a Fusion web application from one WebLogic Server to another WebLogic Server, you may need to perform some of the same steps you did for a first time deployment.

In general, to migrate an application, you would:

- Load the ADF runtime (if it is not already installed) to the target application server. See Preparing the Standalone Application Server for Deployment in *Administering Oracle ADF Applications*.
- Configure the target application server with the correct database or URL connection information.
- Migrate security information from the source to the target. For instructions, see How to Deploy Applications with ADF Security Enabled.
- Deploy the application using Enterprise Manager, administration console, or scripts. See Deploying Using Oracle Enterprise Manager Fusion Middleware Control in *Administering Oracle ADF Applications*.

How to Configure the Application Using ADF MBeans

If ADF MBeans were enabled and packaged with the deployed application, you can configure ADF properties using the Enterprise Manager MBean Browser. For instructions on how to enable an application for MBeans, see How to Enable the Application for ADF MBeans.

For information on how to configure ADF applications using ADF MBeans, see Configuring Application Properties Using the MBean Browser in *Administering Oracle ADF Applications*.

Testing the Application and Verifying Deployment

In order to ensure the deployment of the ADF application has been successful, you should test-run the application.

After you deploy the application, you can test it from the application server. To test-run your ADF application, open a browser window and enter a URL:

- For non-Faces pages: `http://<host>:port/<context root>/<page>`
- For Faces pages: `http://<host>:port/<context root>/faces/<view_id>`

  where `<view_id>` is the view ID of the ADF task flow view activity.

> **Tip:**
>
> The context root for an application is specified in the user interface project settings by default as *`ApplicationName`*/*`ProjectName`*/`context-root`. You can shorten this name by specifying a name that is unique across the target application server. Right-click the user interface project, and choose **Project Properties**. In the Project Properties dialog, select **Java EE Application** and enter a unique name for the context root.

> **Note:**
>
> `/faces` has to be in the URL for Faces pages. This is because JDeveloper configures your `web.xml` file to use the URL pattern of `/faces` in order to be associated with the Faces Servlet. The Faces Servlet does its per-request processing, strips out `/faces` part in the URL, then forwards the URL to the JSP. If you do not include the `/faces` in the URL, then the Faces Servlet is not engaged (since the URL pattern doesn't match). Your JSP is run without the necessary JSF per-request processing.

56
Using State Management in a Fusion Web Application

This chapter describes the Fusion web application state management facilities for ADF application modules to support stateful applications on the web.
This chapter includes the following sections:

About Fusion Web Application State Management

State management in the ADF application is the process by which you maintain state and page information over multiple requests for the same or different pages.

Most real-world business applications need to support multi-step user tasks. Transactional web sites tend to use a step-by-step style user interface to guide the end user through a logical sequence of pages to complete these tasks. When the task is done, the user can save or cancel everything as a unit. However, the HTTP protocol is built on the concept of individual, stateless requests, so the responsibility of grouping a sequence of user actions into a logical, stateful unit of work falls to the application developer.

The limitations of the HTTP protocol have led developers to invent solutions involving a "shadow" set of database tables with no constraints and with all of the column types defined as character-based. Using such a solution becomes very complex very quickly. Ultimately, some kind of generic application state management facility is needed to address these issues in a more generic and workable way. The solution comes in the form of **ADF Business Components**, which implements this for you out of the box.

State management enables you to easily create web applications that support multi-step use cases without falling prey to the memory, reliability, or implementation complexity problems. Oracle ADF simplifies the process of implementing and maintaining session state through the use of application module pools, which are configurable resource managers that release application module caches safely. By default, the ADF Business Components project enables one application module pool for each application module definition in your application.

In Fusion web applications, state management is provided at these levels:

- Save For Later feature in the ADF Controller layer implemented in task flows, as described in Using Save Points in Task Flows.

  Save For Later is activated at the ADF Controller layer and automatically saves a "snapshot" of the current UI and controller states, and delegates to the ADF Model layer to save its state as well. Save for Later saves an incomplete transaction without enforcing validation rules or submitting the data. You can use it to save data and state information about a **region**, view port, or portlet. Later, you can restore application state and data associated with a save point that supports both explicit saves (initiated by the user) or implicit saves (without user action). The end user can resume working on the same transaction with the same data that was originally saved when the application was exited.

- Application state management facility in the ADF Model layer implemented in application modules.

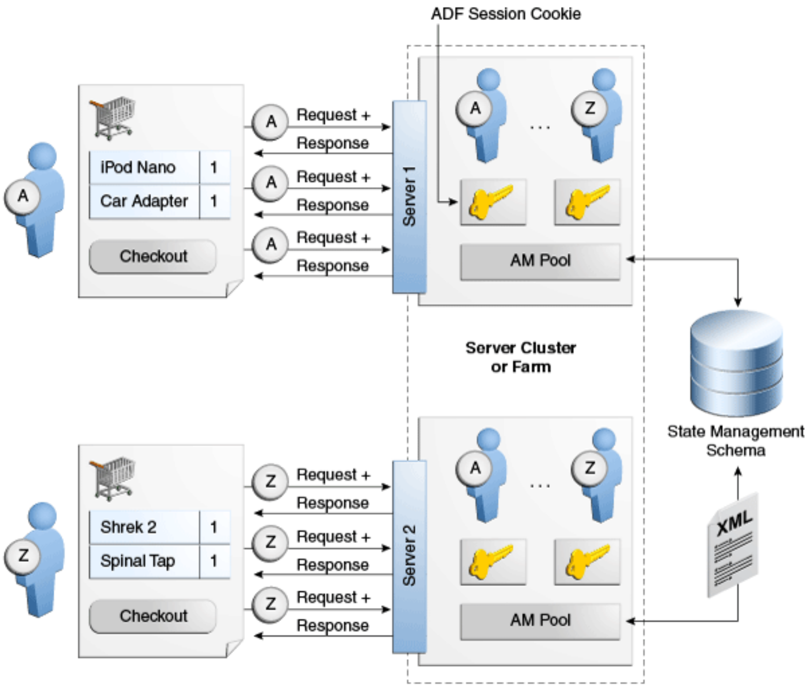
  Through the application state management facility, the ADF runtime automatically manages the application state of each user session and supports highly scalable applications. An application module supports *passivating* (storing) its pending transaction state to an XML document, which is stored either in-memory or in the database in a single, generic table, keyed by a unique passivation snapshot ID. It also supports the reverse operation of **activating** pending transaction state from one of these saved XML snapshots. This passivation and activation is performed automatically by the **application module pool** when needed, and is controlled by configuration properties that you can specify. Understanding what happens behind the scenes is essential to make the most efficient use of this important feature, as explained in Managing When Passivation and Activation Occurs.

- Application state management support for a server farm, or clustered server environment with failover.

  If you deploy in a multiple application server environment with failover enabled, then subsequent end-user requests may be handled by any server in your server farm or cluster. To support this scenario, the application state management facility passivates the application module at the end of every request. In the event failover occurs, where the user session is redirected to another server node, the session cookie can reactivate the pending application state from the database-backed XML snapshot, regardless of which server handles the request. Thus application module passivation and activation provides built-in support for cluster failover without additional configuration. For more information about ADF support for failover, see Configuring High Availability for Fusion Web Applications.

The ADF Business Components application module lets you implement completely stateless applications or support a unit of work that spans multiple browser pages. Figure 56-1 illustrates the basic architecture of the state management facility to support these multi-step scenarios.

Figure 56-1 ADF Provides Generic, Database-Backed State Management



The ADF **binding context** is the one object that lives in the `HttpSession` for each end user. It holds references to lightweight **application module data control** objects that manage acquiring an application module instance from the pool during the request (when the data control is accessed) and releasing it to the pool at the end of each request. The data control holds a reference to the application module handle that identifies the user session. In particular, serialized session objects of the application module state created or modified in the pending transaction are *not* saved in the `HttpSession` using this approach. This minimizes both the session memory required per user and eliminates the network traffic related to session replication if the servers are configured in a cluster.

What You May Need to Know About Multi-Step Tasks

In a typical search-then-edit scenario, the end user searches to find an appropriate row to update, then may open several different pages of related master-detail information to make edits before deciding to save or cancel work. Consider another scenario where the end user wants to book a vacation online. The process may involve the end user's entering details about:

- One or more flight segments that comprise the journey
- One or more passengers taking the trip
- Seat selections and meal preferences
- One or more hotel rooms in different cities
- Car they will rent

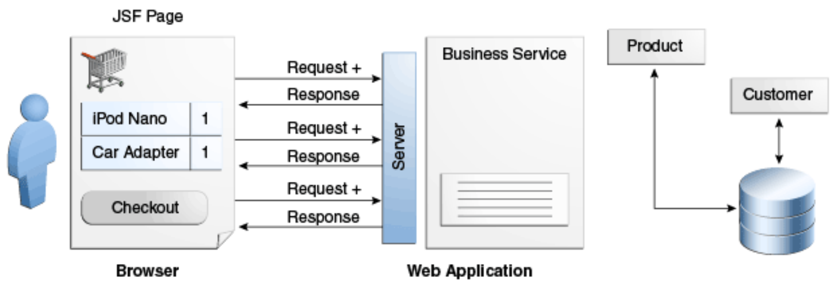
Along the way, the user might decide to complete the transaction, save the reservation for finishing later, or abandon the transaction before saving.

It's clear these scenarios involve a logical unit of work that spans multiple web pages. You've seen in previous chapters how to use JDeveloper's JSF page navigation diagram to design the page flow for these use cases, but that is only part of the puzzle. The pending changes the end user makes to business domain objects along the way — `Trip`, `Flight`, `Passenger`, `Seat`, `HotelRoom`, `Auto`, etc. — represent the in-progress state of the application for each end user. Along with this, other types of "bookkeeping" information about selections made in previous steps comprise the complete picture of the application state.

What You May Need to Know About the Stateless HTTP Protocol

While it may be easy to imagine the multi-step application scenarios, *implementing* them in web applications is complicated by the stateless nature of HTTP, the hypertext transfer protocol. Figure 56-2 illustrates how an end user's visit to a site comprises a series of HTTP request/response pairs. However, HTTP affords a web server no way to distinguish one user's request from another user's, or to differentiate between a single user's first request and any subsequent requests that user makes while interacting with the site. The server gets each request from any user always as if it were the first (and only) one they make.

Figure 56-2 Web Applications Use the Stateless HTTP Protocol



What You May Need to Know About Session Cookies

As shown in Figure 56-3, the technique used to recognize an ongoing sequence of requests from the same end user over the stateless HTTP protocol involves a unique identifier called a **cookie**. A cookie is a name/value pair that is sent in the header information of each HTTP request the user makes to a site. On the initial request made by a user, the cookie is not part of the request. The server uses the *absence* of the cookie to detect the start of a user's session of interactions with the site, and it returns a unique identifier to the browser that represents this session for this user. In practice, the cookie value is a long string of letters and numbers, but for the simplicity of the illustration, assume that the unique identifier is a letter like "A" or "Z" that corresponds to different users using the site.

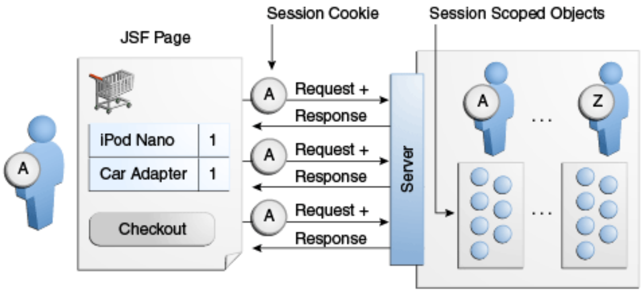
Web browsers support a standard way of recognizing the cookie returned by the server that allows the browser to identify the following:

- the site that sent the cookie
- how long it should remember the cookie value

On each subsequent request made by that user, until the cookie expires, the browser sends the cookie along in the header of the request. The server uses the value of the cookie to distinguish between requests made by different users.

Cookies can be set to live beyond a single browser session so that they might expire in a week, a month, or a year from when they were first created, while a session cookie expires when the user closes the browser.

Figure 56-3 Tracking State Using a Session Cookies and Server-Side Session



Java EE-compliant web servers provide a standard server-side facility called the `HttpSession` that allows a web application to store Java objects related to a particular user's session as named attribute/value pairs. An object placed in this session `Map` on one request can be retrieved by the application while handling a subsequent request during the same session.

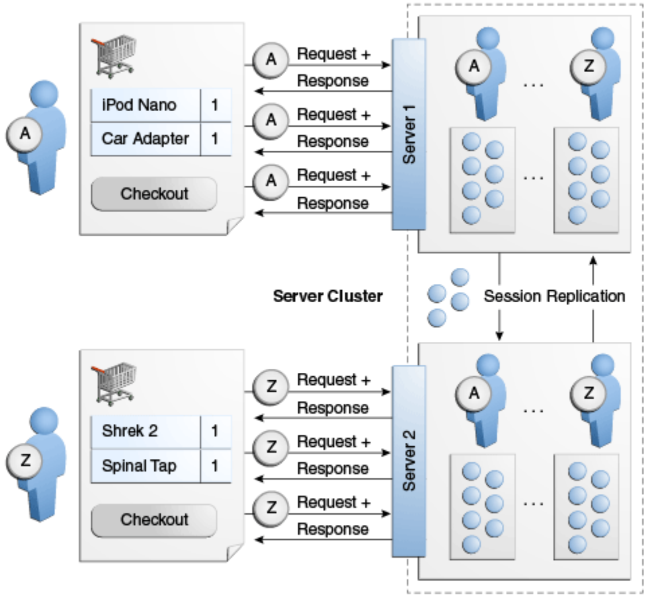
The session remains active while the user continues to send new requests within the timeframe specified by the `<session-timeout>` element in the `web.xml` file. The default session length is typically around 30 minutes, depending on the container.

What You May Need to Know About Using HttpSession

The `HttpSession` facility is an ingredient in most **application state management** strategies, but it can present performance and reliability problems if not used judiciously. First, because the session-scope Java objects the application creates are held in the memory of the Java EE web server, the objects in the HTTP session are lost if the server should fail.

As shown in Figure 56-4, one way to improve the reliability is to configure multiple Java EE servers in a cluster. By doing this, the Java EE application server replicates the objects in the HTTP session for each user across multiple servers in the cluster so that if one server goes down, the objects exist in the memory of the other servers in the cluster that can continue to handle the users requests. Since the cluster comprises separate servers, replicating the HTTP session contents among them involves broadcasting the changes to HTTP session objects over the network.

Figure 56-4 Session Replication in a Server Cluster



You can begin to see some of the performance implications of overusing the HTTP session:

- The more active users, the more HTTP sessions will be created on the server.
- The more objects stored in each HTTP session, the more memory you will need. Note that the memory is not reclaimed when the user becomes inactive; this only happens with a session timeout or an explicit session invalidation. Session invalidations don't always happen because users don't always logout.
- In a cluster, the more objects in each HTTP session that *change*, the more network traffic will be generated to replicate the changed objects to other servers in the cluster.

At the outset, it would seem that keeping the number of objects stored in the session to a minimum addresses the problem. However, this implies leveraging an alternative mechanism for temporary storage for each user's pending application state. The solution comes in the form of **ADF Business Components**, which implements this for you out of the box.

Managing When Passivation and Activation Occurs

Passivation and activation, in the ADF application is designed to keep temporary user data across requests, when there are more online users than the application pool can handle.

The passivation/activation cycle supports application module pooling and was designed originally to maintain a reasonable number of caches in memory, as well as to support failover in high-availability, clustered server environments. This process however requires the application module state to be written to and read from the database, and is itself an expensive operation. In legacy system, comprised of 32-bit platforms, where generally smaller amounts of memory were available than on today's middle-tier servers, the passivation of application module instances helped compensate for memory limitations.

In today's systems, where RAM generally is not a limitation, passivation/activation provides little benefit by reducing middle-tier server memory usage. Today the best use of the passivation/activation cycle built into application module pooling is to support failure conditions in a clustered server environment.

The following scenario depicts when the automatic passivation and activation of application module state occurs. This background may help you to understand later how application module pooling can be tuned to limit the use of passivation/activation.

1. At the beginning of an HTTP request, the application module data control handles the `beginrequest` event by checking out an application module instance from the pool.

   The application module pool returns an *unreferenced* instance. An unreferenced application module is one that is not currently managing the pending state for any other user session.

2. At the end of the request, the application module data control handles the `endrequest` event by checking the application module instance back into the pool in "managed state" mode.

   That application module instance is now *referenced* by the data control that just used it. And the application module instance is an object that still contains pending transaction state made by the data control (that is, **entity object** and **view object** caches; updates made but not committed; and cursor states), stored in memory. As you'll see below, the instance is not *dedicated* to this data control, just referenced by it.

3. On a subsequent request, the same data control — identified by its application module handle — checks out an application module instance again.

   Due to the "stateless with user affinity" algorithm the pool uses, the pool returns the exact same application module instance, with the state still there in memory.

The default configuration of application module pooling dictates, at peak usage times, with a high number of users simultaneously accessing the site, application module instances must be sequentially reused by different user sessions. In this case, the application pool will *recycle* a currently referenced application module instance for use by another session, as follows:

1. The application module data control for User A's session checks an application module instance into the application pool at the end of a request. Assume this instance is named `AM1`.

2. The application module data control for User Z's new session requests an application module instance from the pool for the first time, but there are no unreferenced instances available. The application module pool then:

   - Passivates the state of instance `AM1` to the database.
   - Resets the state of `AM1` in preparation to be used by another session.
   - Returns the `AM1` instance to User Z's data control.

3. On a subsequent request, the application module data control for User A's session requests an application module instance from the pool. The application module pool then:

   - Obtains an unreference instance.

This could be instance AM1, obtained by following the same tasks as in Step 2, or another AM2 instance if it had become unreferenced in the meantime.

- Activates the appropriate pending state for User A from the database.
- Returns the application module instance to User A's data control.

In summary, the process of passivation, activation, and recycling preserves the state referenced by the data control across requests without requiring a dedicated application module instance for each data control. Both browser users in the above scenario are carrying on an application transaction that spans multiple HTTP requests, but the end users are unaware whether the passivation and activation is occurring in the background. They just continue to see the pending changes. While the pending changes never need to be saved into the underlying application database tables until the end user is ready to commit the logical unit of work. The application module pool makes a best effort to keep an application module instance "sticky" to the current data control whose pending state it is managing. This is known as maintaining user session affinity.

The best performance is achieved by minimizing the need for passivation/activation cycle in the deployed Fusion web application. You can control this by changing the default configuration parameters that specify the size of the application module pool, control the resource cleanup behavior of the pool manager, and tune the JDBC connection disconnect behavior used by application module instance at the end of a user request. For more information about design time configuration parameters that let you tune application module pools to reduce passivation, see What You May Need to Know About Application Module Pooling Configuration Parameters.

How to Confirm That Fusion Web Applications Use Optimistic Locking

Oracle recommends using optimistic locking, the default mode for web applications. Pessimistic locking should not be used for web applications as it creates pending transactional state in the database in the form of row-level locks. If pessimistic locking is set, state management will work, but the locking mode will not perform as expected. Behind the scenes, every time an application module is recycled, a rollback is issued in the JDBC connection. This releases all the locks that pessimistic locking had created.

Note that under normal circumstances with optimistic locking, `RowInconsistentException` is thrown when inconsistent data is detected while locking a row during the commit operation. However, ADF can also raise `RowInconsistentException` outside of the commit operation, such as during application module activation, even though the framework is configured for optimistic lock.

> **Performance Tip:**
>
> Always use the default mode optimistic locking for web applications. Only optimistic locking is compatible with the application module unmanaged release level mode, which allows the application module instance to be immediately released when a web page terminates. This provides the best level of performance for web applications that expect many users to access the application simultaneously.

To ensure your configuration uses optimistic locking, open the Business Components page of the adf-config.xml overview editor and confirm that **Locking Mode** (corresponding to the jbo.locking.mode property) is set to optimistic or optupdate. To open the adf-config.xml editor, in the Application Resources window, expand **Descriptors** and **ADF META-INF** folders and double-click the file node.

Optimistic locking (optimistic) issues a SELECT FOR UPDATE statement to lock the row, then detects whether the row has been changed by another user by comparing the change indicator attribute — or, if no change indicator is specified, the values of all the persistent attributes of the current entity as they existed when the entity object was fetched into the cache.

Optimistic update locking (optupdate) does not perform any locking. The UPDATE statement determines whether the row was updated by another user by including a WHERE clause that will match the existing row to update only if the attribute values are unchanged since the current entity object was fetched.

What You May Need to Know About Pending Changes Across HTTP Requests

The ADF state management mechanism relies on passivation and activation to manage the state of an application module instance. Implementing this feature in a robust way is only possible if all pending changes are managed by the application module transaction in the middle tier. The most scalable strategy is to keep pending changes in middle-tier objects and not perform operations that cause pending database state to exist across HTTP requests. This allows the highest leverage of the performance optimizations offered by the application module pool and the most robust runtime behavior for your application.

> **Caution:**
>
> When the jbo.doconnectionpooling configuration parameter is set to true — typically in order to share a common pool of database connections across multiple application module pools — upon releasing your application module to the application module pool, its JDBC connection is released back to the database connection pool and a ROLLBACK will be issued on that connection. This implies that all changes which were posted but *not committed* will be lost. On the next request, when the application module is used, it will receive a JDBC connection from the pool, which may be a different JDBC connection instance from the one it used previously. Those changes that were posted to the database but not committed during the previous request are lost.
>
> The jbo.doconnectionpooling configuration parameter is set by checking the **Disconnect Application Module Upon Release** property on the **Database and Scalability** tab of the overview editor for application module configurations (on the bc4j.xcfg file).

What You May Need to Know About Release Levels and postChanges() Method

The transaction-level `postChanges()` method exists to force the transaction to post unvalidated changes without committing them. This method is not recommended for use in web applications unless you can guarantee that the transaction will definitely be committed or rolled back during the same HTTP request. Failure to heed this advice can lead to strange results in an environment where both application modules and database connections can be pooled and shared serially by multiple different clients.

If for some reason you need to create a transactional state in the database in a particular request by invoking the `postChanges()` method or by calling a PL/SQL stored procedure, but you cannot issue a commit or rollback by the end of that same request, then you *must* release the application module instance with the **reserved** level from that request until a subsequent request when you either commit or rollback.

> **Note:**
>
> Use as short a period of time as possible between creation of transactional state in the database and performing the concluding commit or rollback. This ensures that reserved level doesn't have to be used for a long time, as it has adverse effects on application's scalability and reliability.

Once an application module has been released with reserved level, it remains at that release level for all subsequent requests until release level is explicitly changed back to managed or unmanaged level. So, it is your responsibility to set release level back to managed level once commit or rollback has been issued.

Testing to Ensure Your Application Module is Activation-Safe

It is important during the development process that you test your ADF application modules for being activation-safe. This is done by disabling ADF application module pooling in the application module configuration.

If you have not *explicitly* tested that your application module functions when its pending state gets activated from a passivation snapshot, then you may encounter an unpleasant surprise in your production environment when heavy system load tests this aspect of your system for the first time.

How To Disable Application Module Pooling to Test Activation

As part of your overall testing plan, you should adopt the practice of testing your application modules with the `jbo.ampool.doampooling` configuration parameter set to `false`. This setting completely disables application module pooling and forces the system to activate your application module's pending state from a passivation snapshot on *each* page request. It is an excellent way to detect problems that might occur in your production environment due to assumptions made in your custom application code.

> **Note:**
>
> It is important to reenable application module pooling after you conclude testing and are ready to deploy the application to a production environment. The configuration property `jbo.ampool.doampooling` set to `false` is not a supported configuration for production applications and must be set to `true` before deploying the application.

For example, if you have transient view object attributes you believe *should* be getting passivated, this technique allows you to test that they are working as you expect. In addition, consider situations where you might have introduced:

- Private member fields in application modules, view objects, or entity objects
- Custom user session state in the `Session` user data hashtable

Your custom code likely assumes that this custom state will be maintained across HTTP requests. As long as you test with a single user on the JDeveloper Integrated WebLogic Server, or test with a small number of users, things will appear to work fine. This is due to the "stateless with affinity" optimization of the ADF application module pool. If system load allows, the pool will continue to return the same application module instance to a user on subsequent requests. However, under heavier load, during real-world use, it may not be able to achieve this optimization and will need to resort to grabbing any available application module instance and reactivating its pending state from a passivation snapshot. If you have not correctly overridden `passivateState()` and `activateState()` (as described in Managing Custom User-Specific Information During Passivation) to save and reload your custom component state to the passivation snapshot, then your custom state will be missing (i.e. `null` or back to your default values) after this reactivation step. Testing with `jbo.ampool.doampooling` set to false allows you to quickly isolate these kinds of situations in your code.

For more information about configuring Business Component configuration properties, How to Set Configuration Properties Declaratively.

What You May Need to Know About the jbo.ampool.doampooling Configuration Parameter

The `jbo.ampool.doampooling` configuration property corresponds to the **Enable Application Module Pooling** option in the **Database and Scalability** tab of the overview editor for application module configurations (on the `bc4j.xcfg` file). By default, this checkbox is checked so that application module pooling is enabled. Whenever you deploy your application in a production environment the default setting of `jbo.ampool.doampooling=true` is the way you will run your application. But, as long as you run your application in a test environment, setting the property to `false` can play an important role in your testing. When this property is false, there is effectively no application pool. When the application module instance is released at the end of a request it is immediately removed. On subsequent requests made by the same user session, a new application module instance must be created to handle it and the pending state of the application module must be reactivated from the passivation store.

For more information about configuring Business Component properties, How to Set Configuration Properties Declaratively.

What You May Need to Know About State Management and Data Consistency

When an entity object is updated concurrently by more than one user and one of those users loses affinity with their application module during the update, it is possible to experience data corruption if a change indicator has not been defined for the entity object.

Consider a scenario where two users (User A and User B) access the same entity object at the same time.

- After querying, User B changes the value of an attribute (for example, changing the `Dept` attribute value from `Accounting` to `Research`) and then commits.
- Meanwhile, User A has also queried and sees the same record where the value of the `Dept` attribute is `Accounting`.
- User A loses affinity with the application module that was used to query, resulting in passivation.
- User A then changes the `Dept` attribute value from `Accounting` to `Sales` and commits, resulting in activation.
- When activation occurs, the new value of the `Dept` attribute is activated but the browser is unaware of the data change, and the commit proceeds without issue.

To avoid this type of scenario, define a change indicator attribute for all entity objects. For more information, see How to Protect Against Losing Simultaneously Updated Data.

What You May Need to Know About State Management and Subclassed Entity Objects

If your application employs subclassed entity objects, the key attribute of new entities must be prepopulated. If the key is not prepopulated, passivation and activation will fail. You can prepopulate the key attribute by overriding the `create()` method or by using the `DBSequence` type that will assign a temporary negative value to the key before the real value is fetched from the database after commit. For more information, see How to Get Trigger-Assigned Primary Key Values from a Database Sequence.

Controlling Where Model State Is Saved

You can control where you want to save passivation snapshot by configuring an option in the ADF application module configuration. You can either save it in the database (default option) or in a file stored on the local file system.

By default, passivation snapshots are saved in the database, but you can configure it to use the file system as an alternative.

When saving to the database, the passivated XML snapshot is written to a `BLOB` column in a table named `PS_TXN`, using a connection specified by the `jbo.server.internal_connection` property. Each time a passivation record is saved, it is assigned a unique passivation snapshot ID based on the sequence number taken from the `PS_TXN_SEQ` sequence. The application module handle held by the application module data control in the ADF binding context remembers the latest passivation snapshot ID that was created on its behalf and remembers the previous ID that was used.

How to Control the Schema Where the State Management Table Resides

The ADF runtime recognizes a configuration property named `jbo.server.internal_connection` that controls which database connection and schema should be used for the creation of the `PS_TXN` table and the `PS_TXN_SEQ` sequence. If you don't set the value of this configuration parameter explicitly, then the state management facility creates the temporary tables using the credentials of the current application database connection.

To keep the temporary information separate, the state management facility uses a different connection *instance* from the database connection pool, but the database credentials are the same as the current user. Since the framework creates temporary tables, and possibly a sequence if they don't already exist, the implication of not setting a value for the `jbo.server.internal_connection` is that the current database user must have `CREATE TABLE`, `CREATE INDEX` and `CREATE SEQUENCE` privileges. Since this is often not desirable, Oracle recommends always supplying an appropriate value for the `jbo.server.internal_connection` property, providing the credentials for a state management schema where table and schema be created. Valid values for the `jbo.server.internal_connection` property in your configuration are:

- A fully qualified JDBC connection URL like:

  `jdbc:oracle:thin:`*`username/password@host`*`:`*`port`*`:`*`SID`*

- A JDBC datasource name like:

  `java:/comp/env/jdbc/YourJavaEEDataSourceName`

Performance Tip:

When creating the `PS_TXN` table, use securefiles to store LOB data (the content column), and create a primary column index on the `PS_TXN` table as global, partitioned reverse key index. The securefile configuration delivers superior performance over the basicfile configuration when working with LOB data. The reverse key index helps by reducing contention that can happen when the rate of inserts is high.

How to Configure the Passivation Store

Passivated information can be stored in several places. You can control it by configuring an option in the application module configuration. The choices are database or a file stored on local file system:

- **File**

This choice may be the fastest available, because access to the file is faster then access to the database. This choice is good if the entire middle tier is either installed on the same machine or has access to a commonly shared file system, so passivated information is accessible to all. Usually, this choice may be good for a small middle tier where one Oracle WebLogic Server domain is used. In other words this is a very suitable choice for small middle tier such as one Oracle WebLogic Server instance with all its components installed on one physical machine. The location and name of the persistent snapshot files are determined by `jbo.tmpdir` property if specified. It follows usual rules of ADF property precedence for a configuration property. If nothing else is specified, then the location is determined by `user.dir` if specified. This is a default property and the property is OS specific.

- **Database**

  This is the *default* choice. While it may be a little slower than passivating to file, it is by far the most reliable choice. With passivation to file, the common problem might be that it is not accessible to Oracle WebLogic Server instances that are remotely installed. In this case, in a cluster environment, if one node goes down the other may not be able to access passivated information and then failover will not work. Another possible problem is that even if file is accessible to the remote node, the access time for the local and remote node may be very different and performance will be inconsistent. With database access, time should be about the same for all nodes.

To set the value of your choice in design time, set the property `jbo.passivationstore` to `database` or `file`. The value `null` will indicate that a connection-type-specific default should be used. This will use database passivation for Oracle or DB2, and file serialization for any others.

What Happens at Runtime: What Model State Is Saved and When It Is Cleaned Up

The information saved by application model passivation is divided in two parts: transactional and nontransactional state. Transactional state is the set of updates made to entity object data – performed either directly on entity objects or on entities through view object rows – that are intended to be saved into the database. Nontransactional state comprises view object runtime settings, such as the current row index, `WHERE` clause, and `ORDER BY` clause.

The information saved as part of the application module passivation snapshot includes the following.

Transactional State

- New, modified, and deleted entities in the entity caches of the **root application module** for this user session's (including old/new values for modified ones).

Nontransactional State

- For each active view object (both statically and dynamically created):
  - Current row indicator for each row set (typically one)

- New rows and their positions. (New rows are treated differently then updated ones. Their index in the view object is traced as well.)
- ViewCriteria and all related parameters such as **view criteria** row, etc.
- Flag indicating whether or not a row set has been executed
- Range start and Range size
- Access mode
- Fetch mode and fetch size
- Any view object-level custom data

  > **Note:**
  >
  > Transient view object attributes can be saved if they are selected for passivation at design time. However, use this feature judiciously because this results in a snapshot that will grow in size with the number of rows that have been retrieved. See How to Manage the State of View Objects for information on enable the passivation of transient attributes.

- `SELECT`, `FROM`, `WHERE`, and `ORDER BY` clause if created dynamically or changed from the View definition

> **Note:**
>
> If you enable ADF Business Components runtime diagnostics, the contents of each XML state snapshot are also saved. See How to Enable ADF Business Components Debug Diagnostics for information on how to enable diagnostics.

Other information, such as user-specific information or instant variables in ADF Business Components objects, is not saved as part of the passivation snapshot. To save this type of information, use the techniques described in Managing Custom User-Specific Information During Passivation.

Cleaning Up the Model State

The adfbc_purge_statesnapshots.sql script in JDeveloper helps you to clean up the ADF application module passivation snapshot records time to time.

Under normal circumstances, the ADF state management facility provides automatic cleanup of the passivation snapshot records.

How to Clean Up Temporary Storage Tables Resulting from Passivation

JDeveloper supplies the `adfbc_purge_statesnapshots.sql` script to help with periodically cleaning up the application module state management table. You can find this file in the `oracle_common` subdirectory of your Oracle Middleware installation directory (for example, *`ORACLE_HOME`*`\oracle_common\common\sql`).

Persistent snapshot records can accumulate over time if the server has been shutdown in an abnormal way, such as might occur during development or due to a server failure. Running the script in SQL*Plus will create the `BC4J_CLEANUP` PL/SQL package. The two relevant procedures in this package are:

- `PROCEDURE Session_State(olderThan DATE)`

  This procedure cleans up application module session state storage for sessions older than a given date.

- `PROCEDURE Session_State(olderThan_minutes INTEGER)`

  This procedure cleans up application module session state storage for sessions older than a given number of minutes.

You can schedule periodic cleanup of your ADF temporary persistence storage by submitting an invocation of the appropriate procedure in this package as a database job.

You can use an anonymous PL/SQL block like the one shown in the following example to schedule the execution of `bc4j_cleanup.session_state()` to run starting tomorrow at 2:00 am and each day thereafter to cleanup sessions whose state is over 1 day (1440 minutes) old.

```
SET SERVEROUTPUT ON
DECLARE
  jobId    BINARY_INTEGER;
  firstRun DATE;
BEGIN
  -- Start the job tomorrow at 2am
  firstRun := TO_DATE(TO_CHAR(SYSDATE+1,'DD-MON-YYYY')||' 02:00',
              'DD-MON-YYYY HH24:MI');
   -- Submit the job, indicating it should repeat once a day
  dbms_job.submit(job       => jobId,
                  -- Run the ADF Purge Script for Session State
                  -- to cleanup sessions older than 1 day (1440 minutes)
                  what      => 'bc4j_cleanup.session_state(1440);',
                  next_date => firstRun,
                  -- When completed, automatically reschedule
                  -- for 1 day later
                  interval  => 'SYSDATE + 1'
                 );
  dbms_output.put_line('Successfully submitted job. Job Id is '||jobId);
END;
.
/
```

What Happens at Runtime: When a Passivation Record Is Updated

When a passivation record is saved to the database on behalf of a session cookie, this passivation record gets a new, unique snapshot ID. The passivation record with the previous snapshot ID used by that same session cookie is deleted as part of the same transaction. In this way, assuming no server failures, there will only ever be a single passivation snapshot record per active end-user session.

What Happens at Runtime: When a Passivation Snapshot is Removed on Unmanaged Release

The passivation snapshot record related to a session cookie is removed when the application module is checked into the pool with the unmanaged state level. This can occur when:

- Your code specifically calls `resetState()` on the application module data control.
- Your code explicitly invalidates the `HttpSession`, for example, as part of implementing an explicit "Logout" functionality.
- The `HttpSession` times out due to exceeding the session timeout threshold for idle time and failover mode is disabled (which is the default).

In each of these cases, the application module pool also resets the application module referenced by the session cookie to be "unreferenced" again. Since no changes were ever saved into the underlying database tables, once the pending session state snapshots are removed, there remains no trace of the unfinished work the user session had completed up to that point.

What Happens at Runtime: Passivation Snapshot Retained in Failover Mode

When the failover mode is enabled, if the `HttpSession` times out due to session inactivity, then the passivation snapshot is retained so that the end user can resume work upon returning to the browser.

After a break in the action, when the end user returns to the browser and continues to use the application, it continues working as if nothing had changed. The session cookie is used to reactivate any available application module instance with the user's last pending state snapshot before handling the request. So, even though the users next request will be processed in the context of a new `HttpSession` (perhaps even in a different application server instance), the user is unaware that this has occurred.

> **Note:**
>
> If an application module was released with Reserved level then when the `HttpSession` times out, the user will have to go through an authentication process, and all unsaved changes are lost.

Managing the HttpSession in Fusion Web Applications

The <session-timeout> tag in the web.xml file allows you to configure the timeout threshold so that the HTTP session is freed in Fusion web applications when the user is away or inactive.

Since HTTP is a stateless protocol, the server receives no implicit notice that a client has closed the browser or gone away for the weekend. Therefore any Java EE-compliant server provides a standard, configurable session timeout mechanism to allow resources tied to the HTTP session to be freed when the user has stopped performing requests. You can also programmatically force a timeout.

How to Configure the Implicit Timeout Due to User Inactivity

You configure the session timeout threshold using the `<session-timeout>` tag in the `web.xml` file. The default value is typically around 30 minutes, depending on the container.

How to Code an Explicit HttpSession Timeout

To end a user's session before the session timeout expires, you can call the `invalidate()` method on the `HttpSession` object from a backing bean in response to the user's click on a Logout button or link. This cleans up the `HttpSession` in the same way as if the session time had expired. Using JSF and ADF, after invalidating the session, you *must* perform a redirect to the next page you want to display, rather than just doing a forward. The following example shows sample code to perform this task from a Logout button.

```
public String logoutButton_action() throws IOException{
  ExternalContext ectx = FacesContext.getCurrentInstance().getExternalContext();
  HttpServletResponse response = (HttpServletResponse)ectx.getResponse();
  HttpSession session = (HttpSession)ectx.getSession(false);
  session.invalidate();
  response.sendRedirect("Welcome.jspx");
  return null;
}
```

As with the implicit timeouts, when the HTTP session is cleaned up this way, it ends up causing any referenced application modules to be marked unreferenced.

What Happens at Runtime: How Application Modules are Handled When the HTTP Times Out

When the `HttpSession` times out, the `BindingContext` goes out of scope, and along with it, any data controls that might have referenced application modules released to the pool in the managed state level. The application module pool resets any of these referenced application modules and marks the instances unreferenced again.

Managing Custom User-Specific Information During Passivation

You can add custom user-defined information in the ADF application module by overriding the passivateState() and activateState() methods in the ApplicationModuleImpl class.

It is fairly common practice to add custom user-defined information in the application module in the form of member variables or some custom information stored in `oracle.jbo.Session` user data hashtable. The ADF state management facility provides a mechanism to save this custom information to the passivation snapshot as well, by overriding the `passivateState()` method and the `activateState()` methods in the `ApplicationModuleImpl` class.

> **Note:**
>
> Similar methods are available on the `ViewObjectImpl` class and the `EntityObjectImpl` class to save custom state for those objects to the passivation snapshot as well.

How to Passivate Custom User-Specific Information

You can override `passivateState()` and `activateState()` to ensure that custom application module state information is included in the passivation/activation cycle. The following example shows how this is done.

In the example, `jbo.counter` contains custom values you want to preserve across passivation and activation of the application module state. Each application module has an `oracle.jbo.Session` object associated with it that stores application module-specific session-level state. The session contains a user data hashtable where you can store transient information. For the user-specific data to "survive" across application module passivation and reactivation, you need to write code to save and restore this custom value into the application module state passivation snapshot.

```
/**
 * Overridden framework method to passivate custom XML elements
 * into the pending state snapshot document
 */
public void passivateState(Document doc, Element parent) {
  // 1. Retrieve the value of the value to save
  int counterValue = getCounterValue();
  // 2. Create an XML element to contain the value
  Node node = doc.createElement(COUNTER);
  // 3. Create an XML text node to represent the value
  Node cNode = doc.createTextNode(Integer.toString(counterValue));
  // 4. Append the text node as a child of the element
  node.appendChild(cNode);
  // 5. Append the element to the parent element passed in
  parent.appendChild(node);
}
/**
 * Overridden framework method to activate  custom XML elements
 * into the pending state snapshot document
 */
public void activateState(Element elem) {
  super.activateState(elem);
  if (elem != null) {
    // 1. Search the element for any <jbo.counter> elements
    NodeList nl = elem.getElementsByTagName(COUNTER);
    if (nl != null) {
      // 2. If any found, loop over the nodes found
      for (int i=0, length = nl.getLength(); i < length; i++) {
        // 3. Get first child node of the <jbo.counter> element
        Node child = nl.item(i).getFirstChild();
        if (child != null) {
          // 4. Set the counter value to the activated value
          setCounterValue(new Integer(child.getNodeValue()).intValue()+1);
          break;
        }
      }
    }
  }
}
/*
 * Helper Methods
 */
private int getCounterValue() {
  String counterValue = (String)getSession().getUserData().get(COUNTER);
  return counterValue == null ? 0 : Integer.parseInt(counterValue);
}
private void setCounterValue(int i) {
  getSession().getUserData().put(COUNTER,Integer.toString(i));
}
private static final String COUNTER = "jbo.counter";
```

What Happens When You Passivate Custom Information

As the example in How to Passivate Custom User-Specific Information illustrates, when `activateState()` is overridden, the following steps are performed:

1. Search the element for any `jbo.counter` elements.
2. If any are found, loop over the nodes found in the node list.
3. Get first child node of the `jbo.counter` element.

   It should be a DOM Text node whose value is the string you saved when your `passivateState()` method above got called, representing the value of the `jbo.counter` attribute.
4. Set the counter value to the activated value from the snapshot.

When `passivateState()` is overridden, it performs the reverse job by doing the following:

1. Retrieve the value of the value to save.
2. Create an XML element to contain the value.
3. Create an XML text node to represent the value.
4. Append the text node as a child of the element.
5. Append the element to the parent element passed in.

> **Note:**
>
> The API's used to manipulate nodes in an XML document are provided by the Document Object Model (DOM) interfaces in the `org.w3c.dom` package. These are part of the Java API for XML Processing (JAXP). See the Javadoc for the `Node`, `Element`, `Text`, `Document`, and `NodeList` interfaces in this package for more details.

What You May Need to Know About Activating Custom Information

The `activateState()` method is called at the end of the activation process after the view objects have been activated. Most of the time this is where you want to place the application module state activation logic. However, if your application module activation logic needs to set up custom state information before the ADF state management facility activates the view objects (for example, you might need to write custom code to allow the view objects to internally reference custom values at execution time), then the `prepareForActivation()` method in the `ApplicationModuleImpl` class would be the right place because it fires at the beginning of the activation process.

Managing the State of View Objects During Passivation

ADF Business Components view objects are marked as passivated by default. But a transient view object (containing only transient attributes) only passivates the updates related to the current row and other nontransactional state.

By default, all view objects are marked as passivation-enabled, so their state will be saved. However, view objects that have transient attributes do not have those attributes passivated by default. You can change how a view object is passivated, and even which attributes are passivated, using the Tuning page of the view object overview editor.

How to Manage the State of View Objects

Each view object can be declaratively configured to be passivation-enabled or not. If a view object is not passivation enabled, then no information about it gets written in the application module passivation snapshot.

> **Performance Tip:**
>
> There is no need to passivate read-only view objects that are not used to display values in the UI but are simply used in re-entrant methods for calculation or derivation of values. This eliminates the performance overhead associated with passivation and activation and reduces the CPU usage needed to maintain the application module pool.

Before you begin:

It may be helpful to have an understanding of passivation in view objects. For more information, see Managing the State of View Objects During Passivation.

To set the passivation state of a view object:

1. In the Applications window, double-click the view object for which you want to set the passivation state.
2. In the overview editor, click the **General** navigation tab.
3. On the General page, expand the **Tuning** section and select **Passivate State** to make sure the view object data is saved.
4. Optionally, you can select **Including All Transient Attributes** to passivate all transient attributes at this time, but see What You May Need to Know About Passivating Transient View Objects for additional information.

What You May Need to Know About Passivating View Objects

The activation mechanism is designed to return your view object to the state it was in when the last passivation occurred. To ensure that, **Oracle ADF** stores in the state snapshot the values of any bind variables that were used for the last query execution. These bind variables are in addition to those that are set on the row set at the time of passivation. The passivated state also stores the user-supplied WHERE clause on the view object related to the row set at the time of passivation.

How to Manage the State of Transient View Objects and Attributes

Because view objects are marked as passivated by default, a transient view object — one that contains only transient attributes — is marked to be passivation enabled, but only passivates its information related to the current row and other nontransactional state. Note that to enable a transient view object to passivate or activate correctly, it needs a non-null key attribute value.

> **Performance Tip:**
>
> Transient view object attributes are not passivated by default. Due to their nature, they are usually intended to be read-only and are easily re-created. So, it often doesn't make sense to passivate their values as part of the XML snapshot. This also avoids the performance overhead associated with passivation and activation and reduces the CPU usage needed to maintain the application module pool.

Before you begin:

It may be helpful to have an understanding of passivation in view objects. For more information, see Managing the State of View Objects During Passivation.

You will also need to perform the following task:

- Open the application in JDeveloper.

To individually set the passivation state for transient view object attributes:

1. In the Applications window, double-click the view object that contains the transient attributes for which you want to specify the passivation state.
2. In the overview editor, click the **Attributes** navigation tab.
3. On the Attributes page, select the transient attribute you want to passivate and click the **Details** tab.
4. In the Details page, select the **Passivate** checkbox.

What You May Need to Know About Passivating Transient View Objects

Passivating transient view object attributes is more costly resource-wise and performance- wise, because transactional functionality is usually managed on the entity object level. Since transient view objects are not based on an entity object, this means that all updates are managed in the view object row cache and not in the entity cache. Therefore, passivating transient view objects or transient view object attributes requires special runtime handling. For a transient VO to passivate or activate correctly, it needs a non-null key attribute value.

Usually passivation only saves the values that have been changed, but with transient view objects passivation has to save entire row. The row will include only the view object attributes marked for passivation.

> **Note:**
>
> The ADF passivation and activation logic will take care of restoring transient attribute values in the entity object and view object *only* as long as the entity object is in an unposted state. Once the user Saves their changes to the database, the entity object is posted to the database and in the next activation cycle, the view row is restored using the database query. Since the attribute is transient, its value will remain null. If you want your entity object or view object-based transient attribute values to survive failover, you need to define the transient attribute value using an expression and the expression should point to a getter method that can supply the value.

How to Use Transient View Objects to Store Session-level Global Variables

Using passivation, you can use a view object to store one or more global variables, each on a different transient attribute. When you mark a transient attribute as passivated, the ADF Business Components framework will remember the transient values across passivation and activation in high-throughput and failover scenarios. Therefore, it is an easy way to implement a session-level global value that is backed up by the state management mechanism, instead of the less-efficient HTTP Session replication. This also makes it easy to bind to controls in the UI if necessary.

There are two basic approaches to store values between invocations of different screens, one is controller-centric, and the other is model-centric.

To implement this task in the **ADF Controller** layer using the controller-centric approach involves storing and referencing values using attributes in the pageFlow scope. This approach might be appropriate if the global values do not need to be referenced internally by any implementations of ADF Business Components. For more information about pageFlow scope, see What You May Need to Know About Memory Scope for Task Flows.

The model-centric approach involves creating a transient view object, which is conceptually equivalent to a nondatabase block in Oracle Forms.

Before you begin:

It may be helpful to have an understanding of passivation in view objects. For more information, see Managing the State of View Objects During Passivation.

You will also need to perform the following tasks:

1. Open the application in JDeveloper.

2. Create a new view object using the View Object wizard, as described in How to Create an Entity-Based View Object.
   a. On Page 1 of the wizard, select the option for **Data Source: Programmatic**.
   b. On Page 2, click **New** to define the transient attribute names and types the view object should contain.
   c. On Page 3, set the **Updatable** option to **Always** for each attribute.
   d. Click **Finish** and the newly created view object appears in the overview editor.
3. Add an instance of the view object with transient attributes to your application module's data model, as described in Adding Master-Detail View Object Instances to an Application Module.

To store session-level global variables using the model-centric approach:

1. Disable any entries from being performed in the view object:
   a. In the overview editor, on the General page, expand the **Tuning** section.
   b. In the **Retrieve from the Database** group, select the **No Rows** option.
2. Make sure data in the view object is not cleared out during a rollback operation. To implement this, you enable a custom Java class for the view object and override two rollback methods.
   a. In the overview editor, click the **Java** navigation tab, and click the **Edit** icon for the **Java Classes** section.
   b. In the Select Java Options dialog, select **Generate View Object Class** and click **OK**.
   c. In the overview editor, click the link next to View Object Class in the **Java Classes** section.
   d. From the main menu, choose **Source > Override Methods**.
   e. In the Override Methods dialog, select the `beforeRollback()` and `afterRollback()` methods to override, and click then **OK**.
   f. In both the `beforeRollback()` and `afterRollback()` methods, comment out the call to `super` in the Java code.
3. Create an empty row in the view object when a new user begins using the application module:
   a. In the Applications window, double-click the application module.
   b. Generate an application module Java class if you don't have one yet, as described in How to Generate a Custom Class for an Application Module.
   c. Override the `prepareSession()` method of the application module, as described in How to Override a Built-in Framework Method.
   d. After the call to `super.prepareSession()`, add code to create a new row in the transient view object and insert it into the view object.

Now you can bind read-only and updatable UI elements to the "global" view object attributes just as with any other view object using the **Data Controls panel**.

57
Tuning Application Module Pools

This chapter describes how ADF Business Components application module pools work and how you can tune application module pools to optimize application performance.
This chapter includes the following sections:

About Application Module Pooling

An ADF application module pool is a collection of instances of a single application module type which are shared by multiple application clients. The amount of time between submitting web pages enables a smaller number of application module components to serve a larger number of active users. This reduces memory usage and improves performance.

An application module pool is a collection of application module runtime instances of the same type. To provision a number of users visiting it, the Fusion web application can be configured to provide one or more application module instances from the pool to service users at runtimee.

Each application module instance in a pool is shared by multiple browser clients whose typical "think time" between submitting web pages allows optimizing the number of application module components to be effectively smaller than the total number of active users working on the system. For example, twenty users visiting the website from their browser might be able to be serviced by 5 or 10 application module instances instead of having as many application module instances as you have browser users. Therefore, not only can the pool service more users than the number of application modules available, but in addition the middle tier requires less memory to service this smaller set of application modules allowing ADF applications to scale further on limited hardware resources.

Through your design-time configuration, application modules can be used to support Fusion web application scenarios that are completely stateless, or they can be used to support a unit of work that spans multiple browser pages. As a performance optimization, when an instance of an application module is returned to the pool in "managed state" mode, the pool tracks session references to the application module. The application module instance is still in the pool and available for use, but it would *prefer* to be used by the same session that was using it the last time because maintaining this "session affinity" improves performance.

So, at any one moment in time, the instances of application modules in the pool are logically partitioned into three groups, reflecting their state:

- Checked into the pool and unconditionally *available* for use

- Checked into the pool and available for use, but *referenced* for session affinity reuse by an active user session
- Checked out of the pool and *un*available, inasmuch as it's currently in use (at that very moment) by some thread in the web container

Understanding what happens behind the scenes is essential to make the most efficient use of application module pooling. For details about session state management of application modules, see Managing When Passivation and Activation Occurs.

For details about the configuration parameters that can be set at design time to affect the behavior of the application module pool, see What You May Need to Know About Application Module Pool Configuration Parameters.

Application Module Pools

Application Module components can be used at runtime in two ways:

- As an application module the client accesses directly
- As a reusable component aggregated (or "nested") inside of another application module instance

When a client accesses it directly, an application module is called a **root application module**. Clients access nested application modules *indirectly* as a part of their containing application module instance. It's possible, but not common, to use the same application module at runtime in both ways. The important point is that **ADF Business Components** only creates an application module pool for a *root* application module.

The basic rule is that one application module pool is created for each *root* application module used by a Fusion web application in *each* Java VM where a root application module of that type is used by the **ADF Controller** layer. For advanced topics related to the number of application module pools your application may need, see Deployment Environment Scenarios and Pooling.

Deployment Environment Scenarios and Pooling

The number of pools and the type of pools that your application will utilize will depend upon how the target platform is configured. For example, will there be more than one Java Virtual Machine (JVM) available to service the web requests coming from your application users and will there be more than one Oracle WebLogic Server domain? To understand how many pools of which kinds are created for an application in both a single-JVM scenario and a multiple-JVM runtime scenario, review the following assumptions:

- Your Fusion web application makes use of two application modules `HRModule` and `PayablesModule`.
- You have a `CommonLOVModule` containing a set of commonly used view objects to support list of values in your application, and that both `HRModule` and `PayablesModule` aggregate a nested instance of `CommonLOVModule` to access the common LOV view objects it contains.
- You have configured both `HRModule` and `PayablesModule` to use the same JDeveloper connection definition named `appuser`.

Single Oracle WebLogic Server Domain, Single Oracle WebLogic Server Instance, Single JVM

If you deploy this application to a single Oracle WebLogic Server domain, configured with a single Oracle WebLogic Server instance, there is only a single Java VM available to service the web requests coming from your application users.

Assuming that all the users are making use of web pages that access both the `HRModule` and the `PayablesModule`, this will give:

- One application module pool for the `HRModule` root application module
- One application module pool for the `PayablesModule` root application module

This gives a total of two application module pools in this single Java VM.

> **Note:**
>
> There is no separate application module pool for the nested instances of the reusable `CommonLOVModule`. Instances of `CommonLOVModule` are wrapped by instances of `HRModule` and `PayablesModule` in their respective application module pools.

Multiple Oracle WebLogic Server Domains, Multiple Oracle WebLogic Server Instance, Multiple JVMs

Next consider a deployment environment involving multiple Java VMs. Assume that you have configured four different physical machines as two Oracle WebLogic Server domains, with a hardware load-balancer in front. On these four machines, each Oracle WebLogic Server instance will have a single JVM. As users of your application access the application, their requests are shared across these two Oracle WebLogic Server domains, and within each domain, across the two JVMs that its Oracle WebLogic Server instances have available.

Again assuming that all the users are making use of web pages that access both the `HRModule` and the `PayablesModule`, this will give:

- Four application module pools for `HRModule`, one in each of four JVMs.

  (1 `HRModule` root application module) x (2 Oracle WebLogic Server domains) x (2 Oracle WebLogic Server JVMs each)

- Four application module pools for `PayablesModule`, one in each of four JVMs.

  (1 `PayablesModule` root application module) x (2 Oracle WebLogic Server domains) x (2 Oracle WebLogic Server JVMs each)

This gives a total of eight application module pools spread across four JVMs.

As you begin to explore the configuration parameters for the application module pools in What You May Need to Know About Application Module Pool Configuration Parameters, keep in mind that the parameters apply to a given application module pool for a given application module in a single JVM.

As the load balancing spreads user requests across the multiple JVMs where Oracle ADF is running, each individual application module pool in each JVM will have to support one *n* th of the user load — where *n* is the number of JVMs available to service those user requests. The appropriate values of the application module pools need to be set with the number of Java VMs in mind. The basic approach is to base sizing parameters on load testing and the results of the application module pooling statistics, then divide that total number by the *n* number of pools you will have based on your use of multiple application server domains and multiple Oracle WebLogic Server instances. For example, if you decide to set the minimum number of application modules in the pool to ten and you end up with five pools due to having five Oracle WebLogic Server instances servicing this application, then you would want to configure the parameter to 2 (ten divided by five), not 10 (which would only serve a given application module in a single JVM).

For details about available sizing parameters, see Pool Sizing Parameters.

Setting Pool Configuration Parameters

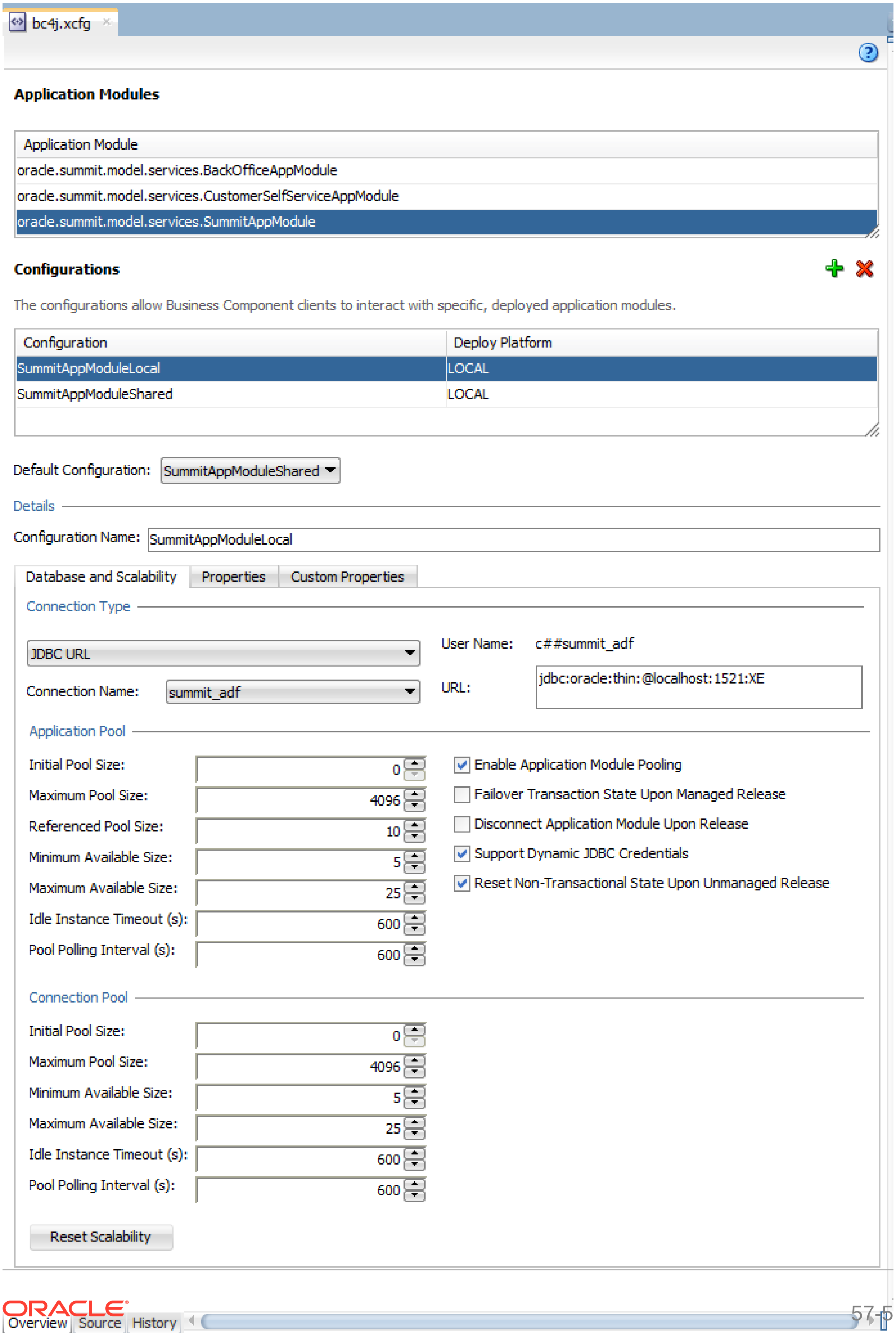
Configuration parameters set in the ADF application allow you to control the runtime behavior of an application module pool. There are three logical categories of the application module pool configuration parameters: Pool behavior, Pool Sizing, and Pool Cleanup Behavior.

You control the runtime behavior of an application module pool by setting appropriate configuration parameters. You can set these declaratively in an application module configuration, supply them as Java System parameters, or set them programmatically at runtime.

How to Set Configuration Properties Declaratively

You use the **Database and Scalability** tab of the overview editor for application module configurations shown in Figure 57-1 for viewing and setting runtime configuration properties of individual application modules.

Figure 57-1 Database and Scalability Tab of the Edit Configuration Dialog



Application Modules

Application Module
oracle.summit.model.services.BackOfficeAppModule
oracle.summit.model.services.CustomerSelfServiceAppModule
oracle.summit.model.services.SummitAppModule

Configurations

The configurations allow Business Component clients to interact with specific, deployed application modules.

Configuration	Deploy Platform
SummitAppModuleLocal	LOCAL
SummitAppModuleShared	LOCAL

Default Configuration: SummitAppModuleShared

Details

Configuration Name: SummitAppModuleLocal

Database and Scalability | Properties | Custom Properties

Connection Type

JDBC URL

Connection Name: summit_adf

User Name: c##summit_adf

URL: jdbc:oracle:thin:@localhost:1521:XE

Application Pool

Initial Pool Size:	0
Maximum Pool Size:	4096
Referenced Pool Size:	10
Minimum Available Size:	5
Maximum Available Size:	25
Idle Instance Timeout (s):	600
Pool Polling Interval (s):	600

☑ Enable Application Module Pooling
□ Failover Transaction State Upon Managed Release
□ Disconnect Application Module Upon Release
☑ Support Dynamic JDBC Credentials
☑ Reset Non-Transactional State Upon Unmanaged Release

Connection Pool

Initial Pool Size:	0
Maximum Pool Size:	4096
Minimum Available Size:	5
Maximum Available Size:	25
Idle Instance Timeout (s):	600
Pool Polling Interval (s):	600

Reset Scalability

Before you begin:

It may be helpful to have an understanding of application module pooling. For more information, see Setting Pool Configuration Parameters.

Familiarize yourself with the application module configuration parameters. For more information about configuration parameters for application module pool tuning, see What You May Need to Know About Application Module Pool Configuration Parameters.

For best practice guidelines that may help you to tune the Fusion web application for an expected level of usage, see the Tuning Oracle Application Development Framework chapter of *Tuning Performance*.

Note that if your application defines a JDBC URL connection type (legacy only), refer to the 11*g* release version of this guide to obtain documentation on the ADF database connection pool configuration parameters.

To edit your application module's pooling configuration:

1. In the Applications window, double-click the application module.
2. In the overview editor, click the **Configurations** navigation tab.
3. In the Configurations page, click the configuration hyperlink for the configuration you want to edit.
4. In the overview editor for application module configurations, click the **Database and Scalability** tab and edit the desired runtime properties and click **OK** to save the changes for your configuration.

What Happens When You Set Configuration Properties Declaratively

The values that you supply through the overview editor for application module configurations are saved in an XML file named `bc4j.xcfg` in the `./common` subdirectory relative to the application module's XML definition. All of the configurations for all of the application modules in a single Java package are saved in that same file. Typically, you do not modify the `bc4j.xcfg` file directly and use the overview editor to change configuration settings for specific application modules. To display the application module configuration overview editor, double-click the application module in the Applications window and, in the overview editor, select the **Configurations** navigation tab. Then, in the Configurations page of the overview editor, click the configuration hyperlink.

For example, if you look at the overview editor for application module configurations for the `BackOfficeAppModule` application module in the `SummitADF` application (on the `bc4j.xcfg` file in the `./src/oracle/summit/model/services/common` directory), you will see the two named configurations for the `BackOfficeAppModule` application module. In this case, as the following example shows, the `BackOfficeAppModuleLocal` and the `BackOfficeAppModuleShared` configurations specify JDBC URL connections for use by the Oracle ADF Model Tester. The connection details for the JDBC connections appear in the `connections.xml` file located in the `./.adf/META-INF` subdirectory relative to the project directory.

```
<BC4JConfig version="11.1" xmlns="http://xmlns.oracle.com/bc4j/configuration">
...
   <AppModuleConfigBag
      ApplicationName="oracle.summit.model.services.BackOfficeAppModule">
      <AppModuleConfig
```

```
            name="BackOfficeAppModuleLocal"
            jbo.project="oracle.summit.model.Model"
            ApplicationName="oracle.summit.model.services.BackOfficeAppModule"
            DeployPlatform="LOCAL" JDBCName="summit_adf">
         <Database jbo.TypeMapEntries="Java"/>
         <Security

AppModuleJndiName="oracle.summit.model.services.BackOfficeAppModule"/>
      </AppModuleConfig>
      <AppModuleConfig
            name="BackOfficeAppModuleShared"
            jbo.project="oracle.summit.model.Model"
            ApplicationName="oracle.summit.model.services.BackOfficeAppModule"
            DeployPlatform="LOCAL" JDBCName="summit_adf">
         <AM-Pooling jbo.ampool.maxpoolsize="1"
                     jbo.ampool.isuseexclusive="false"/>
         <Database jbo.TypeMapEntries="Java"/>
         <Security

AppModuleJndiName="oracle.summit.model.services.BackOfficeAppModule"/>
      </AppModuleConfig>
   </AppModuleConfigBag>
</BC4JConfig>
```

Note that attributes of child elements of the `<AppModuleConfig>` tag have names beginning with `jbo` that match the names of their ADF Business Components properties (for example, the `<AM-Pooling>` tag defines the attribute `jbo.ampool.maxpoolsize` that corresponds to the property `jbo.ampool.maxpoolsize`). It's also important to understand that if a property is currently set to its runtime *default* value in the application module configurations editor, then JDeveloper does *not* write the entry to the `bc4j.xcfg` file.

How to Programmatically Set Configuration Properties

When your application requires control over application module configuration properties at runtime, you can dynamically set these properties using ADF Business Components API. For example, in a typical ADF application, your application module may configure a static JDBC data source. However, when the application needs to display data, depending upon the user, from various databases, you might set the data source name and application module pool parameters at runtime.

You can set configuration properties programmatically by creating a Java class that implements the `EnvInfoProvider` interface in the `oracle.jbo.common.ampool` package. In your class, you override the `getInfo()` method and call `put()` to put values into the environment Hashtable passed in as shown in the following example.

```
package devguide.advanced.customenv.view;
import java.util.Hashtable;
import oracle.jbo.common.ampool.EnvInfoProvider;
/**
 * Custom EnvInfoProvider implementation to set
 * environment properties programmatically
 */
public class CustomEnvInfoProvider implements EnvInfoProvider {
  /**
   * Overridden framework method to set custom values in the
   * environment hashtable.
   *
   * @param string - ignore
   * @param environment Hashtable of config parameters
   * @return null - not used
   */
  public Object getInfo(String string, Object environment) {
    Hashtable envHashtable = (Hashtable)environment;
    envHashtable.put("some.property.name","some value");
    return null;
  }
  /* Required to implement EnvInfoProvider */
  public void modifyInitialContext(Object object) {}
  /* Required to implement EnvInfoProvider */
  public int getNumOfRetries() {return 0;}
}
```

When creating an application module for a stateless or command-line-client, with the `createRootApplicationModule()` method of the `Configuration` class, you can pass the custom `EnvInfoProvider` as the optional second argument. In order to use a custom `EnvInfoProvider` in an ADF web-based application, you need to implement a custom session cookie factory class as shown in the following example. To use your custom session cookie factory, set the `jbo.ampool.sessioncookiefactoryclass` configuration property to the fully qualified name of your custom session cookie factory class.

```
package devguide.advanced.customenv.view;
import java.util.Properties;
import oracle.jbo.common.ampool.ApplicationPool;
import oracle.jbo.common.ampool.EnvInfoProvider;
import oracle.jbo.common.ampool.SessionCookie;
import oracle.jbo.http.HttpSessionCookieFactory;
/**
 * Example of custom http session cookie factory
 * to install a custom EnvInfoProvider implementation
 * for an ADF web-based application.
 */
public class CustomHttpSessionCookieFactory
       extends HttpSessionCookieFactory {
  public SessionCookie createSessionCookie(String appId,
                                           String sessionId,
                                           ApplicationPool pool,
                                           Properties props) {
    SessionCookie cookie =
      super.createSessionCookie(appId, sessionId,pool, props);
    EnvInfoProvider envInfoProv = new CustomEnvInfoProvider();
    cookie.setEnvInfoProvider(envInfoProv);
    return cookie;
  }
}
```

Then add the JAR file containing the `oracle.jbo.http.HttpSessionCookieFactory` class to the project file. The JAR file `adfmweb.jar` can be found in the `[JDEV_INSTALL]\Oracle_common\modules\oracle.adf.model\` directory. This file is also called ADF Web Runtime in the libraries. To add the JAR file follow these steps:

1. Right-click the project and select **Project Properties**.
2. Select **Libraries and Classpath**.
3. Click **Add Library**.
4. Select **ADF Web Runtime** and click **OK**.

> **Note:**
>
> If you do not add the JAR file, the IDE displays an error when you set `jbo.envinfoprovider` to set the Data Source name and Application Module Pool parameters dynamically at runtime

What You May Need to Know About Configuration Property Scopes

Each runtime configuration property used by ADF Business Components has a scope. The scope of each property indicates when the property's value is evaluated and whether its value is effectively shared (i.e. static) in a single Java VM, or not. The ADF Business Components `PropertyManager` class is the registry of all supported properties. It defines the property names, their default values, and their scope. This class contains a `main()` method so that you can run the class from the command line to see a list of all the configuration property information.

Assuming JDEVHOME is the JDeveloper installation directory, to see this list of settings for reference, do the following:

```
$ java -cp JDEVHOME/BC4J/lib/bc4jmt.jar oracle.jbo.common.PropertyManager
```

Issuing this command will send all of the ADF Business Components configuration properties to the console. It also lists a handy reference about the different levels at which you can set configuration property values and remind you of the precedence order these levels have:

```
---------------------------------------------------------------
Properties loaded from following sources, in order:
1. Client environment [Provided programmatically
                       or declaratively in bc4j.xcfg]
2. Applet tags
3. -D flags (appear in System.properties)
4. bc4j.properties file (in current directory)
5. /oracle/jbo/BC4J.properties resource
6. /oracle/jbo/commom.jboserver.properties resource
7. /oracle/jbo/common.Diagnostic.properties resource
8. System defined default
---------------------------------------------------------------
```

You'll see each property is listed with one of the following scopes:

- MetaObjectManager

  Properties at this scope are initialized once per Java VM when the ADF PropertyManager is first initialized.

- SessionImpl

  Properties at this scope are initialized once per invocation of `ApplicationModule.prepareSession()`.

- Configuration

  Properties at this scope are initialized when the application module pool is first created and the application module's configuration is read the first time.

- Diagnostic

Properties at this scope are specific to the built-in ADF Business Components diagnostic facility.

At each of these scopes, the layered value resolution described above is performed when the properties are initialized. Whenever property values are initialized, if you have specified them in the *Client Environment* (level 1 in the resolution order) the values will take precedence over values specified as System parameters (level 3 in the resolution order).

The Client Environment is a hashtable of name/value pairs that you can either programmatically populate, or which will be automatically populated for you by the `Configuration` object when loaded, with the name/value pairs it contains in its entry in the `bc4j.xcfg` file. The implication of this is that for any properties scoped at MetaObjectManager level, the most reliable way to ensure that all of your application modules use the same default value for those properties is to do both of the following:

1. Make sure the property value does not appear in any of your application module's `bc4j.xcfg` file configuration name/value pair entries.
2. Set the property value using a Java system property in your runtime environment.

If, instead, you leave any MetaObjectManager-scoped properties in your `bc4j.xcfg` files, you will have the undesirable behavior that they will take on the value specified in the configuration of the *first* application module whose pool gets created after the Java VM starts up.

What You May Need to Know About Application Module Pool Configuration Parameters

The application module pool configuration parameters fall into three logical categories relating to pool behavior, pool sizing, and pool cleanup behavior. By default Fusion web applications enable application module pooling.

Pool Behavior Parameters

The application module defines configuration parameters that affect the behavior of the application module pool and whether application module instances holds onto JDBC connection that they obtain from the connection pool after an instance is removed from the application module pool.

How ADF application module pools use the database connection pool depends on the setting of the **Disconnect Application Module Upon Release** configuration parameter `jbo.doconnectionpooling`. As shown in Figure 57-1, by default the setting is disabled and the parameter has the setting `jbo.doconnectionpooling=false`.

.

> **Note:**
>
> The setting `jbo.doconnectionpooling=false` does not mean that there is no database connection pooling happening. What it means is that the application module is not disconnected from its JDBC connection upon check in back to the application module pool.

When an application module instance is created in the application module pool it acquires a JDBC connection. By default, ADF favors maintaining session affinity so that across the request of a single user, ADF will try and reuse the same application module instance (including its associated memory and JDBC connection and state), with the goal of speeding up performance for such users. The default setting of `jbo.doconnectionpooling=false` enables this behavior which enforces application module instances keep their JDBC `PreparedStatement` objects cached and reusable across subsequent requests by clients.

However, as the number of users who access the Fusion web application increases, so does the memory requirement on the server to hold onto the JDBC connection, prepare statement, and result sets. When the application module pool is configured to support several thousand users, the demands of not releasing JDBC connections after each request can place demands on server load with a fraction of the supported managed user sessions. This situation is particularly notable in a deployment environment configured to use database drivers that optimize performance over memory consumption, as has been the case for Oracle JDBC drivers since Oracle Database 10g.

To avoid the situation where server memory can be rapidly consumed by the JDBC driver, you can override the default application module instance connection behavior by setting `jbo.doconnectionpooling=true`, then each time a user session finishes using an application module (typically at the end of each HTTP request), the application module instance disassociates itself with the JDBC connection it was using on that request and it returns the connection to the JDBC connection pool. The next time that application module instance is used by a user session, it will reacquire a JDBC connection from the JDBC connection pool and use it for the span of time that application module is checked out of the application module pool (again, typically the span of one HTTP request). Since the application module instance "unplugs" itself from the JDBC connection object used to create each `PreparedStatement`, in this situation it follows that the `prepareSession()` method will fire each time the application module is checked out of the pool to reinitialize the database state. So, when releasing JDBC connections is enabled for sessions, the trade-off is slightly more JDBC overhead setup each time, in return for holding onto a smaller number of overall database connections.

The key difference is seen when many application module pools are all using the same underlying database user for their application connection.

- If 50 different application module pools each have even just a single application module instance in them, with `jbo.doconnectionpooling=false` there will be 50 JDBC application connections in use. If the application module pooling size parameters are set such that the application module pools are allowed to shrink to 0 instances after an appropriate instance idle timeout by setting the **Minimum Available Size** configuration parameter `jbo.ampool.minavailablesize=0`, then when the application module is removed from its pool, it will put back the connection its holding onto. This situation is optimal when the number of managed application module instances remains low, outside of peak demand times or when the application needs to support a low number of simultaneous users.
- In contrast, if 50 different application module pools each have a single application module instance and `jbo.doconnectionpooling=true`, then the amount of JDBC connections in use will depend on how many of those application modules are *simultaneously* being used by different clients. If an application module instance is in the pool and is not currently being used by a user session, then with `jbo.doconnectionpooling=true` it will have released its JDBC connection back to the connection pool and while the application module instance is sitting there

waiting for either another user to need it again, or to eventually be cleaned up by the application module pool monitor, it will not be "hanging on" to a JDBC connection. This situation is optimal in ensuring the scalability of the Fusion web application during peak demand times or when the number of simultaneous users is expected to increase.

When you set `jbo.doconnectionpooling` to `true`, the default setting 1 for the **Transaction Disconnect Level** configuration parameter `jbo.txn.disconnect_level` ensures that all application modules, view objects and row sets remain in memory and stay valid after their corresponding references to JDBC connections are dropped. This configuration reduces the memory requirement associated with this situation where application module passivation is normally performed upon release. Instead, upon activation, the framework only needs to reexecute and synchronize the cursor positions.

Alternatively, you may be able to achieve a balance between conserving JDBC connections and holding onto connections by configuring the **Connection Threshold** configuration parameter `jbo.ampool.connection_threshold`. This parameter specifies the maximum number of connections that application modules in all application module pools combined may hold onto without releasing back to the connection pool during application module pool cleanup. Note that setting the `jbo.ampool.connection_threshold` parameter to a maximum number of connections (greater than 0) implies disconnecting application modules during pool cleanup is desired and therefore the configuration parameter `jbo.doconnectionpooling` must not be enabled. For more details about how the `jbo.ampool.connection_threshold` parameter affects application module pooling, see What You May Need to Know About Optimizing Application Module Pooling.

Pool Sizing Parameters

The application module defines configuration parameters that affect the sizing of the application module pool. When the pool monitor performs one of its resource cleanup passes, it will try to bring the number of application module instances into the range specified by the maximum and minimum available size parameters.

When the pool monitor cannot reduce the pool size below the maximum size by removing unused and idle application module instance, it will then remove active instances that have not timed out and passivate the state of these active application modules. The **Referenced Pool Size** configuration parameter `jbo.recyclethreshold` determines how many active application modules will remain in the pool monitor when passivation occurs. By default, the threshold over which the pool monitor will begin passivating is 10 application module instances.

You can use the **Maximum Available Pool Size** configuration parameter `jbo.ampool.maxavailablesize` to specify the upper range of the pool size after resource cleanup by the pool monitor. Typically, you override the default maximum available size 25 based on the number of concurrent users and how frequently the users need to access the services of the application module. However, when server memory is not the limiting factor for application performance, the maximum available size may be set to an arbitrarily large value to minimize the affect of passivation and activation of application module instances that are still tied to active user sessions.

> **Note:**
>
> Specifying a sufficiently large maximum pool size is especially important when configuring pool cleanup not to reclaim unused application module instances and to allow idle times of 30 minutes for a typical HTTP session timeout. For more information about configuring pool cleanup, see Pool Behavior Parameters.

Similarly, you can use the **Minimum Available Pool Size** configuration parameter `jbo.ampool.minavailablesize` to specify the lower range of the pool size, when the default of 5 instances is not sufficient to ensure a baseline of available application module instance. Typically, consider setting this value equal to the number of concurrent users at non-peak usage. Setting this value too low results in needless instantiation of application module instances that can affect performance.

Pool Cleanup Parameters

The application module defines configuration parameters that affect how resources are reclaimed when the pool monitor does one of its resource cleanup passes.

A single application module pool monitor per Java VM runs in a background thread and wakes up to scan through the pool to perform cleanup at the interval of 600000 milliseconds (10 minutes) by default. To change the default Pool Polling Interval, you can configure the **Pool Polling Interval** parameter `jbo.ampool.monitorsleepinterval`.

The pool monitor uses the following two, independent strategies to identify which application module instances are candidates to be removed from the pool and reclaimed as a potential new resource.

1. The application module pool monitor removes application module instances from the pool that have not been used for more than 3600000 milliseconds (which is the default value and is exactly one hour). These unused application module instances will be reclaimed regardless of the minimum available size configured for the pool. To override the maximum time to live for an application module, you can set the **Maximum Instance Time to Live** configuration parameter `jbo.ampool.timetolive`. However, for most Fusion web applications, you will not need to reclaim unused application module instance and can set this parameter value to `-1`.
2. The application module pool monitor removes application module instances that have remained idle for 600000 milliseconds (which is the default value and is exactly 10 minutes). This cleanup stops when the number of instances in the pool reaches the minimum available size. You can override the idle timeout for application module instances using the **Idle Instance Timeout** configuration parameter `jbo.ampool.maxinactiveage` when you want to allow idle user sessions of more than 10 minutes.

> **Best Practice:**
>
> When you specify the length of time between application module pool cleanup passes, set the same value for **Pool Polling Interval** configuration parameter `jbo.ampool.monitorsleepinterval` for all application modules. Since there is only a single application monitor pool monitor per Java VM, the value that will effectively be used for the application module pool monitor polling interval will be the value found in the application module configuration read by the *first* application module pool that gets created. Setting all application modules to use the same value ensures that this value is set in a predictable way.

What You May Need to Know About Optimizing Application Module Pooling

You may be able to achieve a balance between conserving JDBC connections and holding onto connections by configuring the **Connection Threshold** configuration parameter `jbo.ampool.connection_threshold`. This parameter specifies the maximum number of connections that application modules in all application module pools combined may hold onto without releasing back to the connection pool upon the next pool cleanup cycle.

When the **Connection Threshold** parameter `jbo.ampool.connection_threshold` is enabled (value greater than `0`), application module connections are released during pool cleanup, not during checkin. The cleanup monitor makes a best effort to disconnect *least recently-used* application modules until the total number of JDBC connections falls below the specified threshold limit. Since enabling the `jbo.ampool.connection_threshold` parameter changes when application module pool cleanup is performed, this implies setting the parameter `jbo.ampool.connection_threshold=true` is not compatible.

> **Note:**
>
> When configuring a threshold value for disconnecting application modules during the pool cleanup cycle, be sure to disable `jbo.doconnectionpooling` so the value is `false` (default). The **Connection Threshold** parameter is not compatible with the setting `jbo.doconnectionpooling=true` since this setting causes JDBC connections to release back to the connection pool upon *checkin* of the application module, and the setting works in combination with the `jbo.txn.disconnect_level` parameter to ensure business components remain in memory. For more information about the `jbo.doconnectionpooling` parameter, see Pool Behavior Parameters.

For example, assume the **Connection Threshold** parameter is set to 10 for two application module pools, where application module pool 1 has seven application modules and pool 2 has eight application modules for a total of 15 JDBC connections. When the application module pool cleanup monitor runs, the monitor will make a best effort to reclaim five connections (15 connection resources - 10 connection threshold). The monitor disconnects application modules proportional to the total number of connected application modules for each pool. In this case, one third of the application modules connections (5/15) from each pool will be disconnected. So pool 1 will loose two connections (7/3) and pool 2 will release three connections (8/3), where the fraction of the pool size will be rounded up.

The cleanup monitor makes a best effort to disconnect application modules until the connected resource number falls below the specified limit but note that it may not be able to do so if there are not enough application modules in an available state. To ensure JDBC connections are released efficiently, you can configure the **Pool Polling Interval** parameter `jbo.ampool.monitorsleepinterval` to shorten the monitor sleep period.

Initializing Database State and Pooling Considerations

You can set database state on a per-user basis for your Fusion web application. This is of great benefit as there may be a need to call upon stored procedures to initialize database state related to a particular user's session.

Sometimes you may need to invoke stored procedures to initialize database state related to the current user's session. The correct place to perform this initialization is in an overridden `prepareSession()` method of your application module.

How to Set Database State Per User

The Fusion web application can set database state on a per-user basis. You typically create a database `CONTEXT` namespace, associate a PL/SQL procedure with it, and then use the `SYS_CONTEXT()` SQL function to reference values from the context.

For example, you can use the PL/SQL package to set and get a package-level variable that holds the name of the currently authenticated Fusion web application user as shown in the following example.

```
create or replace package context_pkg as
  procedure set_app_user_name(username varchar2);
  function app_user_name return varchar2;
end context_pkg;
```

Then your application can define the `WHERE` clause of a view object to reference the `context_pkg.app_user_name` function and query the per-user state.

To set the database state, the application module framework extension class (*AppModuleImpl*`.java`) defines a `callStoredProcedure()` helper method similar to the ones in How to Invoke Stored Procedure with Only IN Arguments. The custom application module class then extends this framework extension class and defines the `setCurrentUserInPLSQLPackage()` helper method shown in the following example. The helper method uses the `callStoredProcedure()` method to invoke the `context_pkg.set_app_user_name()` stored procedure, passing the value of the currently authenticated user as a parameter value.

```
// In CustomAppModuleImpl.java
public void setCurrentUserInPLSQLPackage() {
  String user = getUserPrincipalName();
  callStoredProcedure("context_pkg.set_app_user_name(?)",new Object[]{user});
}
```

With this helper method in place, the custom application module class then overrides the `prepareSession()` method as shown in the following example.

```
// In CustomAppModuleImpl.java
  protected void prepareSession(Session session) {
    super.prepareSession(session);
    getLoggedInUser().retrieveUserInfoForAuthenticatedUser();
    setUserIdIntoUserDataHashtable();
    setCurrentUserInPLSQLPackage();
  }
```

Part VII

Appendixes

This part provides information that developers may need to know when building Fusion web applications with Oracle ADF.

Specifically, it contains the following appendixes:

- Oracle ADF XML Files
- Oracle ADF Binding Properties
- ADF Security Permission Grants
- Most Commonly Used ADF Business Components Methods
- ADF Business Components Java EE Design Pattern Catalog
- ADF Equivalents of Common Oracle Forms Triggers
- Performing Common Oracle Forms Tasks in Oracle ADF
- Deploying ADF Applications to GlassFish

A

Oracle ADF XML Files

This appendix provides a reference for the Oracle ADF metadata files that you create in your data model and user interface projects. You may use this information when you want to edit the contents of the metadata these files define.
This appendix includes the following sections:

About ADF Metadata Files

An Oracle Fusion web application uses various metadata XML files that summarizes basic information about data, which can make finding and working with particular instances of data easier. These files are used to configure the runtime behavior of the system.

Metadata files in a Fusion web application are structured XML files used by the application to:

- Specify the parameters, methods, and return values available to your application's Oracle ADF **data control** usages
- Create objects in the Oracle ADF **binding context** and define the runtime behavior of those objects
- Define configuration information about the UI components in JSF and **ADF Faces**
- Define application configuration information for the Java EE application server

In the case of **ADF bindings**, you can use the binding-specific editors to customize the runtime properties of the binding objects. You can open a binding editor when you display the Structure window for a page definition file and choose **Go to Properties** from the context menu.

Additionally, you can view and edit the contents of any metadata file in JDeveloper's XML editor. The easiest way to work with these files is through the Structure window and Properties window. In the Structure window, you can choose an element and in the Properties window, you can define attribute values for the element, often by choosing among dropdown menu choices. Use this appendix to learn the choices you can choose in the case of the Oracle ADF-specific elements.

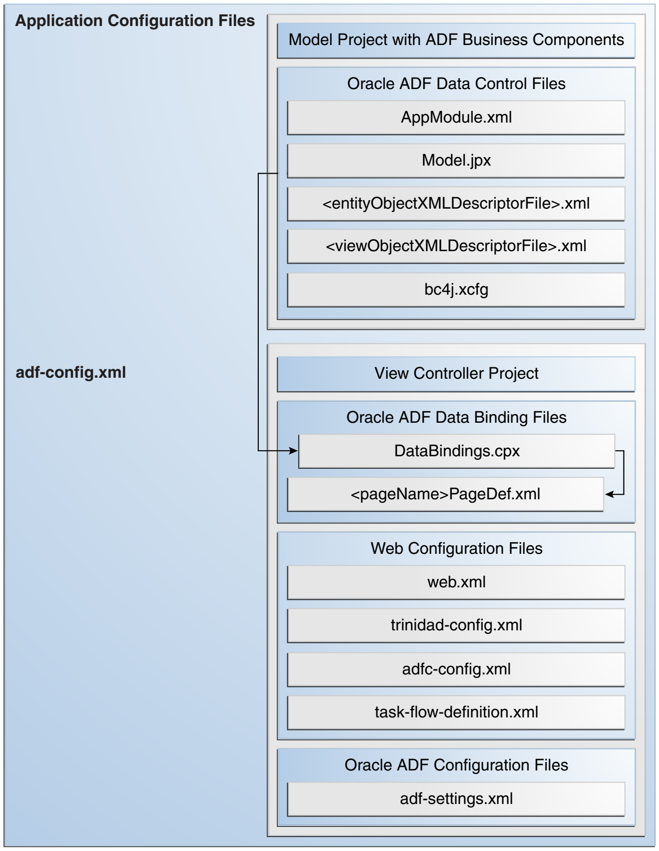
ADF File Overview Diagram

The Oracle ADF File Hierarchy diagram helps you to understand the interrelationship of the metadata files used to work with the ADF Business Components data control in a view layer project.

The relationship between the Oracle ADF metadata files defines dependencies between the model data and the user interface projects. The dependencies are defined as file references within XML elements of the files, where these design time definitions in the Model project reference the DataBindings.jpx file in the ViewController project, which in turn references the page definition files.

Figure A-1 illustrates the hierarchical relationship of the XML metadata files that you might work with in a Fusion web application that uses an ADF Business Components **application module** as a service interface to JSF web pages.

Figure A-1 Oracle ADF File Hierarchy Overview for the Fusion Web Application


Application Configuration Files
adf-config.xml
Model Project with ADF Business Components
Oracle ADF Data Control Files
AppModule.xml
Model.jpx
<entityObjectXMLDescriptorFile>.xml
<viewObjectXMLDescriptorFile>.xml
bc4j.xcfg
View Controller Project
Oracle ADF Data Binding Files
DataBindings.cpx
<pageName>PageDef.xml
Web Configuration Files
web.xml
trinidad-config.xml
adfc-config.xml
task-flow-definition.xml
Oracle ADF Configuration Files
adf-settings.xml

Oracle ADF Data Control Files

In a Fusion web application containing **ADF Business Components**, the data control implementation files are contained within the application. The application module and **view object** XML component descriptor files provide the references for the data control. These files, with the `bc4j.xcfg` file, provide the necessary information for the data control.

An application that uses ADF Business Components in one project and a non-ADF Business Components data control in another project may have a `DataControls.dcx` file, as well as supporting `<`*`sessionbeanname`*`>.xml` and `<`*`beanname`*`>.xml` files.

Oracle ADF Data Binding Files

These standard XML configuration files for a Fusion web application appear in your user interface project:

- `adfm.xml`: This file lists the `DataBindings.cpx` file that is available in the current project.

  See adfm.xml for more information.

- `DataBindings.cpx` : This file contains the page map, page definitions references, and data control references. The file is created the first time you create a data binding for a UI component (either from the Structure window or from the **Data Controls panel**). The `DataBindings.cpx` file defines the Oracle ADF binding context for the entire application. The binding context provides access to the bindings and data controls across the entire application. The `DataBindings.cpx` file also contains references to the `<`*`pagename`*`>PageDef.xml` files that define the metadata for the Oracle ADF bindings in each web page.

  For more information, see DataBindings.cpx.

- `<`*`pagename`*`>PageDef.xml`: This is the page definition XML file. It associates web page UI components with data, or data controls. JDeveloper creates this file each time you design a new web page using the Data Controls panel or Structure window. These XML files contain the metadata used to create the bindings that populate the data in the web page's UI components. For every web page that refers to an ADF binding, there must be a corresponding page definition file with binding definitions.

  For more information, see pageNamePageDef.xml.

Web Configuration Files

These standard XML configuration files required for a JSF application appear in your user interface project:

- `web.xml`: Part of the application's configuration is determined by the contents of its Java EE application deployment descriptor, `web.xml`. The `web.xml` file defines everything about your application that a server needs to know. The file plays a role in configuring the Oracle ADF data binding by setting up the `ADFBindingFilter`. Additional runtime settings include servlet runtime and initialization parameters, custom tag library location, and security settings.

For more information about ADF data binding and JSF configuration options, see web.xml.

An ADF Faces application typically uses its own set of configuration files in addition to `web.xml`. For more information, see Configuration in trinidad-config.xml in *Developing Web User Interfaces with Oracle ADF Faces*.

- `adfc-config.xml:` The configuration file for an ADF unbounded task flow. The configuration file contains metadata about the activities and control flows contained in the unbounded task flow. The default name for this file is `adfc-config.xml`, but an end user can change the name.

  For more information, see adfc-config.xml.

- `task-flow-definition.xml`: The configuration file for an ADF bounded task flow. The configuration file contains metadata about the activities and control flows contained in the bounded task flow. The default name for this file can be `task-flow-defintion.xml` or whatever an end user specifies in the Create ADF Task Flow dialog. The same application can contain multiple task flow definition files.

  For more information, see task-flow-definition.xml.

ADF File Syntax Diagram

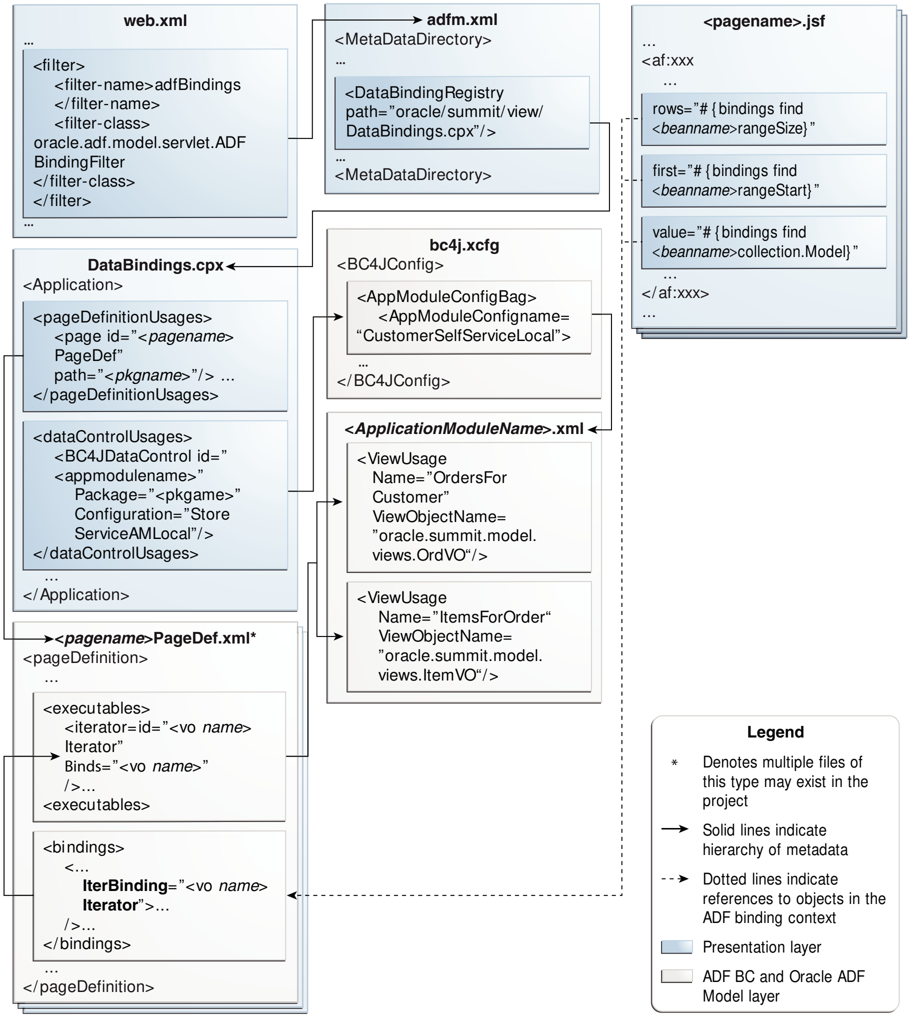
Oracle ADF File Syntax Diagram helps you to understand the hierarchical interrelationship of the XML metadata files in a web application that uses an ADF application module as a service interface to ADF Business Components.

Figure A-2 illustrates the hierarchical relationship of the XML metadata files that you might work with in a web application that uses an ADF application module as a service interface to ADF Business Components. At runtime, the objects created from these files interact in this sequence:

1. When the first request for an ADF databound web page occurs, the servlet registers the Oracle ADF servlet filter `ADFBindingFilter` named in the `web.xml` file.
2. The binding filter creates an empty binding context.
3. When a page is rendered, the binding filter asks the binding context to load a corresponding `PageDef.xml` for the page.
4. The binding context creates the **binding container** by loading the `<page>` file as referenced by the `<pagemap>` element in the `DataBindings.cpx` file.
5. The `adfm.xml` file loads the `DataBindings.cpx` contents and finds the right `PageDef.xml` based on the `<pagemap>` element reference to the `<pageDefinitionUsage>` element.
6. The binding container's `prepareModel` phase prepares and refreshes all relevant executables (most are marked `deferred` by default).
7. An **iterator binding** gets executed by referencing the named method on the data control found through the data control factory named in the case of ADF Business Components in the `bc4j.xcfg` file.
8. The binding container also creates the bindings defined in the `<bindings>` section of the *`pagename`*`PageDef.xml` file for the mapped web page.
9. The web page references to ADF bindings through EL using the expression `#{bindings}` are resolved by accessing the binding container of the page.

10. The page pulls the available data from the bindings in the binding container.

Figure A-2 Oracle ADF File Hierarchy and Syntax Diagram for an ADF BC-based Web Application



adfm.xml

The adfm.xml file is a registry of registries used by the ADF framework to quickly find all metadata files used at runtime. This registry XML file records four types of metadata files —.cpx, .dcx, .xcfg, .jpx.

The `adfm.xml` file contains the classpath-relative paths for the `.cpx`, `.dcx`, `.jpx`, and `.xcfg` files in each design time project that is included in the runtime deployed application. The `adfm.xml` file operates as a dynamically maintained "Registry of Registries" that is used to quickly find all `.cpx`, `.dcx`, `.jpx`, and `.xcfg` files (which are themselves registries of metadata).

The file registry is used extensively by the ADF Library resource catalog browsing implementations, by **ADF Model** layer design time, and at runtime during merge and discovery.

When a developer creates a binding on a page, JDeveloper adds metadata files (for example, page definitions) in the project source tree. The adfm.xml file then notes the location of each.

When a project is built, the adfm.xml file is put in *project-root*/adfmsrc/META-INF/adfm.xml. The project-level archive deployment profiles locate the file at META-INF/adfm.xml.

At runtime, the application classpath is scanned to build the list of .cpx files that comprise the application. The application then loads each.cpx as needed to create the binding context. For details about ADF Model layer usage, see What Happens When You Use the Data Controls Panel.

Four types of sub registries are recorded by the adfm.xml file:

- DataBindingRegistry (.cpx)
- DataControlRegistry (.dcx)
- BusinessComponentServiceRegistry (.xcfg)
- BusinessComponentProjectRegistry (.jpx)

modelProjectName.jpx

JDeveloper uses the modelProjectName.jpx file as a registry to track the contents of the model packages. It also contains metadata for shared ADF application modules present in the project, which is used by both design-time and runtime components.

The.jpx file contains configuration information that JDeveloper uses in the design time to allow you to create the data model project with ADF Business Components. It also contains metadata that defines how a **shared application module** is used at runtime. Because the shared application module can be accessed by any data model project in the same Fusion web application, JDeveloper maintains the scope of the shared application module in the ADF Business Components project configuration file.

This file is saved in the src directory of the project. For example, the Summit sample application for Oracle ADF stores the Model.jpx file in the .src/oracle/summit/model subdirectory of the Model project.

The following example displays a sample default.jpx file.

```
<JboProject
   xmlns="http://xmlns.oracle.com/bc4j"
   Name="Model"
   Version="12.1.2.66.11"
   SeparateXMLFiles="true"
   PackageName="oracle.summit.model">
   <DesignTime>
      <Attr Name="_appModuleNames2"
         Value="oracle.summit.model.services.BackOfficeAppModule"/>
      <Attr Name="_jprName" Value="../../../../Model.jpr"/>
      <Attr Name="_appModuleNames1"
         Value="oracle.summit.model.services.CustomerSelfServiceAppModule"/>
      <Attr Name="_appModuleNames0"
         Value="oracle.summit.model.services.SummitAppModule"/>
      <Attr Name="_jbo.TypeMapEntries" Value="Java"/>
      <Attr Name="_NamedConnection" Value="summit_adf"/>
   </DesignTime>
   <Containee
```

```
      Name="assoc"
      PackageName="oracle.summit.model.entities.assoc"
      ObjectType="JboPackage">
      <DesignTime>
         <Attr Name="_AS" Value="true"/>
      </DesignTime>
   </Containee>
   <Containee
      Name="entities"
      PackageName="oracle.summit.model.entities"
      ObjectType="JboPackage">
      <DesignTime>
         <Attr Name="_EO" Value="true"/>
         <Attr Name="_VO" Value="true"/>
      </DesignTime>
   </Containee>
   <Containee
      Name="services"
      PackageName="oracle.summit.model.services"
      ObjectType="JboPackage">
      <DesignTime>
         <Attr Name="_AM" Value="true"/>
      </DesignTime>
   </Containee>
   <Containee
      Name="links"
      PackageName="oracle.summit.model.views.links"
      ObjectType="JboPackage">
      <DesignTime>
         <Attr Name="_VL" Value="true"/>
      </DesignTime>
   </Containee>
   <Containee
      Name="readonly"
      PackageName="oracle.summit.model.views.readonly"
      ObjectType="JboPackage">
      <DesignTime>
         <Attr Name="_VO" Value="true"/>
      </DesignTime>
   </Containee>
   <Containee
      Name="views"
      PackageName="oracle.summit.model.views"
      ObjectType="JboPackage">
      <DesignTime>
         <Attr Name="_VO" Value="true"/>
      </DesignTime>
   </Containee>
</JboProject>
```

bc4j.xcfg

The bc4j.xcfg file contains metadata information about the ADF application module and runtime parameters that have been configured to manage the runtime behavior of each application module instance.

The `bc4j.xcfg` file contains metadata information about application module names, the database connection used by the application module, and the runtime parameters the user has configured for the application module.

The bc4j.xcfg file is located in the ./common subdirectory relative to the application module's XML component definition. All of the configurations for all of the application modules in a single Java package are saved in that same file. For example, if you look at the bc4j.xcfg file in the /Model/src/oracle/summit/model/services/common directory of the Summit ADF sample application's Model project, you see the configurations for its application modules. For details about editing the configurations, see How to Change Your Application Module's Runtime Configuration and Setting Pool Configuration Parameters.

Example A-1 displays a sample bc4j.xcfg file from the Summit ADF sample application.

Example A-1   Sample bc4j.xcfg File

```
<?xml version = '1.0' encoding = 'UTF-8'?>
<BC4JConfig version="11.1" xmlns="http://xmlns.oracle.com/bc4j/configuration">
   <AppModuleConfigBag
ApplicationName="oracle.summit.model.services.SummitAppModule">
      <AppModuleConfig name="SummitAppModuleLocal" DeployPlatform="LOCAL"
JDBCName="summit_adf" jbo.project="oracle.summit.model.Model"
java.naming.factory.initial="oracle.jbo.common.
       JboInitialContextFactory"
ApplicationName="oracle.summit.model.services.SummitAppModule">
         <Database jbo.TypeMapEntries="Java"/>
         <Security
AppModuleJndiName="oracle.summit.model.services.SummitAppModule"/>
      </AppModuleConfig>
      <AppModuleConfig name="SummitAppModuleShared" DeployPlatform="LOCAL"
JDBCName="summit_adf" jbo.project="oracle.summit.model.Model"
java.naming.factory.initial="oracle.jbo.common.JboInitialContextFactory"

ApplicationName="oracle.summit.model.services.SummitAppModule">
         <AM-Pooling jbo.ampool.isuseexclusive="false"
jbo.ampool.maxpoolsize="1"/>
         <Database jbo.TypeMapEntries="Java"/>
         <Security
AppModuleJndiName="oracle.summit.model.services.SummitAppModule"/>
      </AppModuleConfig>
   </AppModuleConfigBag>
   <AppModuleConfigBag
ApplicationName="oracle.summit.model.services.CustomerSelfServiceAppModule">
      <AppModuleConfig
name="CustomerSelfServiceLocal" jbo.project="oracle.summit.model.Model"
ApplicationName="oracle.summit.model.services.CustomerSelfServiceAppModule"
                                          DeployPlatform="LOCAL"
JDBCName="summit_adf">
         <Database jbo.TypeMapEntries="Java"/>
         <Security
AppModuleJndiName="oracle.summit.model.services.CustomerSelfServiceAppModule"/>
      </AppModuleConfig>
      <AppModuleConfig
name="CustomerSelfServiceShared" jbo.project="oracle.summit.model.Model"
ApplicationName="oracle.summit.model.services.CustomerSelfServiceAppModule"
                                         DeployPlatform="LOCAL"
JDBCName="summit_adf">
         <AM-Pooling jbo.ampool.maxpoolsize="1"
jbo.ampool.isuseexclusive="false"/>
         <Database jbo.TypeMapEntries="Java"/>
         <Security
AppModuleJndiName="oracle.summit.model.services.CustomerSelfServiceAppModule"/>
      </AppModuleConfig>
   </AppModuleConfigBag>
   <AppModuleConfigBag
ApplicationName="oracle.summit.model.services.BackOfficeAppModule">
      <AppModuleConfig
name="BackOfficeAppModuleLocal" jbo.project="oracle.summit.model.Model"
ApplicationName="oracle.summit.model.services.BackOfficeAppModule"
DeployPlatform="LOCAL" JDBCName="summit_adf">
         <Database jbo.TypeMapEntries="Java"/>
         <Security
AppModuleJndiName="oracle.summit.model.services.BackOfficeAppModule"/>
      </AppModuleConfig>
      <AppModuleConfig
name="BackOfficeAppModuleShared" jbo.project="oracle.summit.model.Model"
ApplicationName="oracle.summit.model.services.BackOfficeAppModule"
DeployPlatform="LOCAL" JDBCName="summit_adf">
         <AM-Pooling jbo.ampool.maxpoolsize="1"
jbo.ampool.isuseexclusive="false"/>
         <Database jbo.TypeMapEntries="Java"/>
         <Security
AppModuleJndiName="oracle.summit.model.services.BackOfficeAppModule"/>
      </AppModuleConfig>
   </AppModuleConfigBag>
</BC4JConfig>
```

DataBindings.cpx

In an ADF web application, as soon as you drop a databound component on your page, a DataBindings.cpx file gets created. The DataBindings.cpx keeps track of the individual page definition files used within the project, as well as the DataControls mapping to things such as ADF Business Components Application Modules.

The `DataBindings.cpx` file is created in the user interface project the first time you drop a data control usage onto a web page in the visual editor. The `DataBindings.cpx` file defines the Oracle ADF binding context for the entire application and provides the metadata from which the Oracle ADF binding objects are created at runtime. It is used extensively by the ADF Library resource catalog browsing implementations, and also by the `.cpx` and `.dcx` design time and runtime merge and discovery. When you insert a databound UI component into your document, the page will contain binding expressions that access the Oracle ADF binding objects at runtime.

The `DataBindings.cpx` file appears in the `/src` directory of the user interface project. When you double-click the file node, the binding context description appears in the XML source editor. (To edit the binding context parameters, use the Properties window and choose the desired parameter in the Structure window.)

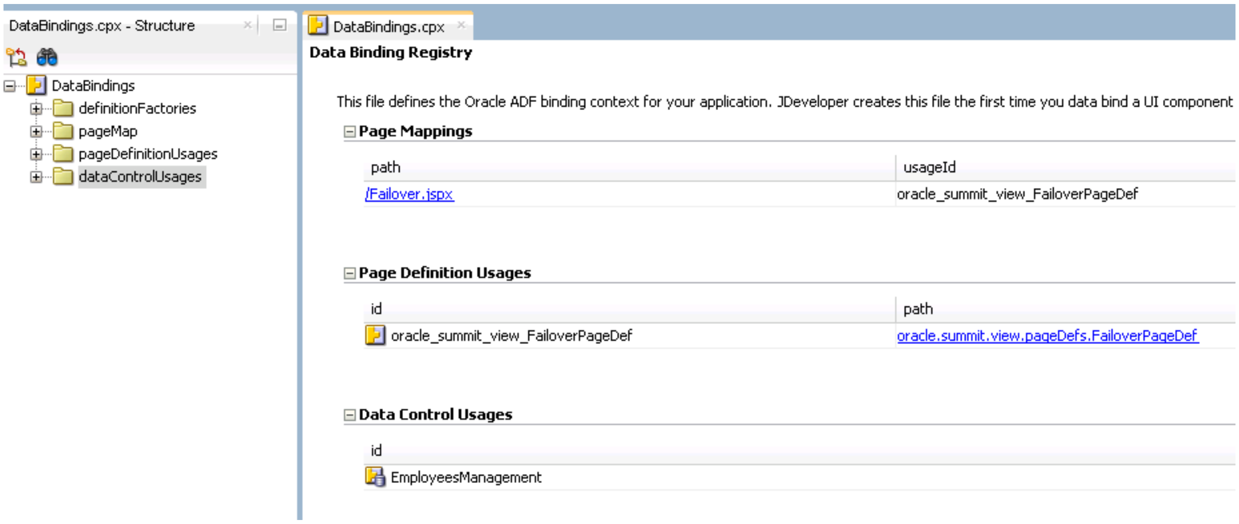
DataBindings.cpx Syntax

The top level element of the `DataBindings.cpx` file is `<Application>`. For example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<Application xmlns="http://xmlns.oracle.com/adfm/application"
             version="11.1.1.56.60" id="DataBindings" SeparateXMLFiles="false"
             Package="oracle.summit.view" ClientType="Generic">
```

Figure A-3 shows the structure definition of the `DataBindings.cpx` file in the Summit standalone sample applications for Oracle ADF.

Figure A-3 Structure of the DataBindings.cpx File



Data Binding Registry

This file defines the Oracle ADF binding context for your application. JDeveloper creates this file the first time you data bind a UI component

Page Mappings

path	usageId
/Failover.jspx	oracle_summit_view_FailoverPageDef

Page Definition Usages

id	path
oracle_summit_view_FailoverPageDef	oracle.summit.view.pageDefs.FailoverPageDef

Data Control Usages

id
EmployeesManagement

The child elements have the following usages:

- `<definitionFactories>` registers a factory class to create the ADF binding objects associated with a particular namespace at runtime. The factory class is specific to the namespace associated with the type of ADF binding (for instance, a task flow binding).
- `<pageMap>` maps all user interface URLs and the corresponding page definition usage name. This map is used at runtime to map a URL to its page definition.
- `<pageDefinitionUsages>` maps a page definition usage (`BindingContainer` instance) name to the corresponding page definition. The `id` attribute represents the usage ID. The path attribute represents the full path to the page definition.
- `<dataControlUsages>` declares a list of data control usages (shortnames) and corresponding path to the data control definition entries in the `.dcx` or `.xcfg` file.

Table A-1 describes the attributes of the `DataBindings.cpx` elements.

Table A-1 Attributes of the DataBindings.cpx File Elements

Element Syntax	Attributes	Attribute Description
`<definitionFactories> <factory/></ definitionFactories>`	`nameSpace`	A URI. Identifies the location of the executable elements in the page definition usage.
	`className`	The fully qualified class name. Identifies the location of the factory class that creates the page definition usage objects.
`<pageMap> <page /></ pageMap>`	`path`	The full directory path. Identifies the location of the user interface page.
	`usageId`	A unique qualifier. Names the page definition ID that appears in the ADF **page definition file**. The ADF binding servlet looks at the incoming URL requests and checks that the `bindings` variable is pointing to the ADF page definition associated with the URL of the incoming HTTP request.

Table A-1 (Cont.) Attributes of the DataBindings.cpx File Elements

Element Syntax	Attributes	Attribute Description
`<pageDefinitionUsages><page/></pageDefinitionUsages>`	`id`	A unique qualifier. References the page definition ID that appears in the ADF page definition file.
	`path`	The fully qualified package name. Identifies the location of the user interface page's ADF page definition file.
`<dataControlUsages><dc.../></dataControlUsages>`	`id`	A unique qualifier. Identifies the data control usage as is defined in the `DataControls.dcx` file.
	`path`	The fully qualified package name. Identifies the location of the data control.

DataBindings.cpx Sample

Example A-2 shows the syntax for the `DataBindings.cpx` file in the Summit ADF sample application.

The ADF executable definition factory (`factory` element) is named by a `className` attribute and is associated with a `namespace`. At runtime, the factory class creates the executable definition objects that leads to the creation of the binding objects for the ADF binding container associated with a particular page definition. The factory locates the page definition through two `DataBindings.cpx` file elements: the `pageMap` element that maps the page URL to the page definition ID (`usageId` attribute) assigned at design time and the `pageDefinitionUsages` element that maps the ID to the location of the page definition from the project or project classpath.

Additionally, the ADF Business Components data control (`BC4JDataControl` element) is named by the `id` attribute. The combination of the `Package` attribute and the `Configuration` attribute is used to locate the `bc4j.xcfg` file in the `./common` subdirectory of the indicated package. The configuration contains the information of the application module name and all the runtime parameters the user has configured.

Example A-2 Sample DataBindings.cpx File

```
<?xml version="1.0" encoding="UTF-8" ?>
<Application xmlns="http://xmlns.oracle.com/adfm/application"
             version="11.1.1.56.60" id="DataBindings" SeparateXMLFiles="false"
             Package="oracle.summit.view" ClientType="Generic">
  <definitionFactories>
    <factory nameSpace="http://xmlns.oracle.com/adf/controller/binding"

className="oracle.adf.controller.internal.binding.TaskFlowBindingDefFactoryImpl"/
>
    <dtfactory
className="oracle.adf.controller.internal.dtrt.binding.BindingDTObjectFactory"/>
    <factory nameSpace="http://xmlns.oracle.com/adfm/dvt"

className="oracle.adfinternal.view.faces.dvt.model.binding.FacesBindingFactory"/>
  </definitionFactories>
  <pageMap>
    <page path="/index.jsf" usageId="oracle_summit_view_indexPageDef"/>
    <page path="/Customers.jsff" usageId="oracle_summit_view_CustomersPageDef"/>
    <page path="/orders/Orders.jsff"
          usageId="oracle_summit_view_OrdersPageDef"/>
    <page path="/carousel/InventoryControl.jsff"
usageId="oracle_summit_view_InventoryControlTestPageDef"/>
  </pageMap>
  <pageDefinitionUsages>
    <page id="oracle_summit_view_CustomersPageDef"
          path="oracle.summit.view.pageDefs.CustomersPageDef"/>
    <page id="oracle_summit_view_indexPageDef"
          path="oracle.summit.view.pageDefs.indexPageDef"/>
    <page id="oracle_summit_view_OrdersPageDef"
          path="oracle.summit.view.pageDefs.OrdersPageDef"/>
    <page id="oracle_summit_view_InventoryControlTestPageDef"
          path="oracle.summit.view.pageDefs.InventoryControlPageDef"/>
  </pageDefinitionUsages>
  <dataControlUsages>
    <BC4JDataControl id="BackOfficeAppModuleDataControl"
                     Package="oracle.summit.model.services"
                     FactoryClass="oracle.adf.model.bc4j.DataControlFactoryImpl"
                     SupportsTransactions="true" SupportsFindMode="true"
                     SupportsRangesize="true" SupportsResetState="true"
                     SupportsSortCollection="true"
                     Configuration="BackOfficeAppModuleLocal"
                     syncMode="Immediate"
                     xmlns="http://xmlns.oracle.com/adfm/datacontrol"/>
  </dataControlUsages>
</Application>
```

*pageName*PageDef.xml

In an ADF web application, as soon as you draga component from Data Control panel to a JSF page, a page definition file gets created with the name pageNamePageDef.xml. The pageNamePageDef.xml file contains binding metadata for each data bound UI element present on the page.

The *pageName*`PageDef.xml` files are created each time you insert a databound component into a web page using the Data Controls panel or Structure window. These XML files define the Oracle ADF binding container for each web page in the application. The binding container provides access to the bindings within the page. You will have one XML file for each databound web page.

> **Caution:**
>
> The `DataBindings.cpx` file maps JSF pages to their corresponding page definition files. If you change the name of a page definition file or a JSF page, JDeveloper does *not* automatically refactor the `DataBindings.cpx` file. You must manually update the page mapping in the `DataBindings.cpx` file.

The `PageDef.xml` file appears in the `/src/view` directory of the user interface project. The Applications window displays the file in the view package of the Application Sources node. When you double-click the file node, the page description appears in the XML source editor. To edit the page description parameters, use the Properties window and choose the desired parameter in the Structure window. See Working with Page Definition Files.

There are important differences in how the page definitions are generated for methods that return a single value and a collection.

PageDef.xml Syntax

The top-level element of the `PageDef.xml` file is `<pageDefinition>`:

```
<pageDefinition xmlns="http://xmlns.oracle.com/adfm/uimodel"
                version="11.1.1.59.23" id="<pagename>PageDef"
                Package="oracle.summit.view.pageDefs">
```

where the XML namespace attribute (`xmlns`) specifies the URI to which the ADF binding container binds at runtime. Only the package name is editable; all other attributes should have the values shown.

The following example displays the child element hierarchy of the `<pageDefinition>` element. Note that each business service for which you have created a data control will have its own `<AdapterDataControl>` definition.

```
<?xml version="1.0" encoding="UTF-8" ?>
<pageDefinition>
    <parameters>
        ...
    </parameters>
    <executables>
        ...
    </executables>
    <bindings>
        ...
    </bindings>
</pageDefinition>
```

The child elements have the following usages:

- `<parameters>` defines page-level parameters that are EL accessible. These parameters store information local to the web page request and may be accessed in the binding expressions.
- `<executables>` defines the list of items (methods, view objects, and accessors) to execute during the `prepareModel` phase of the ADF page lifecycle. Methods to be executed are defined by `<methodIterator>`. The lifecycle performs the execute in the sequence listed in the `<executables>` section. Whether or not the method or operation is executed depends on its `Refresh` or `RefreshCondition` attribute value. Built-in operations on the data control are defined by:
  - `<page>`: definition for a nested page definition (binding container)
  - `<iterator>`: definition to a named collection in `DataControls`
  - `<accessorIterator>`: definition to get an accessor in a data control hierarchy
  - `<methodIterator>`: definition to get to an iterator returned by an invoked method defined by a `methodAction` in the same file
  - `<variableIterator>`: internal iterator that contains variables declared for the binding container
  - `<invokeAction>`: definition of which method to invoke as an executable

> **Note:**
>
> Only use `invokeAction` when your application does not use the ADF Controller module from the Fusion web application technology stack. If your application uses the ADF Controller module, Oracle recommends that you use a task flow method call activity, as described in Using Method Call Activities.

- `<bindings>` refers to an entry in `<executables>` to get to the collection from which bindings extract/submit attribute level data.

Table A-2 describes the attributes of the top-level `<pageDefinition>` element.

Table A-2 Attributes of the PageDef.xml File <pageDefinition> Element

Element Syntax	Attributes	Attribute Description
`<pageDefinition>`	`ControllerClass`	Fully qualified class name to create when controller requests a `PageController` object for this `bindingContainer`.
	`EnableTokenValidation`	Enables currency validation for this bindingContainer when a postback occurs. This is to confirm that the web tier state matches the state that particular page was rendered with.
	`FindMode`	FindMode is for legacy (10.1.2) use only and indicates whether this bindingContainer should start out in findMode when initially prepared.
	`MsgBundleClass`	Fully qualified package name. Identifies the class which contains translation strings for any bindings.
	`SkipValidation`	Determines if data validation occurs. The supported values are: • `true`: skips data validation. Note that attribute validation on the client side at the binding level still occurs. For example, validates non-null and type conversion errors. • `false`: validates all rows for all data controls referenced in the current page. This is the default value. • `skipDataControls`: validates the current rows of iterator bindings modified in the current page. • `custom`: set to `custom` if your application implements an instance of the `oracle.binding.BindingContainerValidator` interface and references it through an EL expression entry named `CustomValidator` in the binding container. Setting a value for this attribute can be useful if you want to skip data validation on, for example, a train component. For more information, see How to Create the Train Model in *Developing Web User Interfaces with Oracle ADF Faces*.
	`Viewable`	An EL expression that should resolve at runtime to whether this binding and the associated component should be rendered or not.

Table A-3 describes the attributes of the child element of `<parameters>`.

Table A-3 Attributes of the PageDef.xml File <parameters> Element

Element Syntax	Attributes	Attribute Description
`<parameter>`	`evaluate`	Specifies when the parameter should be evaluated: `eachUse`, `firstUse`, or `inPrepareModel`.
	`id`	Unique identifier. May be referenced by ADF bindings.
	`option`	Indicates the usage of the variable within the binding container: • `Final` indicates that this parameter cannot be passed in by a usage of this binding container. It must use the default value in the definition. • `Optional` indicates that the variable value need not be provided. • `Mandatory` specifies that the parameter value must be provided at runtime using Java or an EL expression. Failure to provide a value for a mandatory parameter or binding leads to an exception being thrown during binding validation.
	`readonly`	Indicates whether the parameter value may be modified or not. Set to `true` when you do not want the application to modify the parameter value.
	`value`	A default value, which can be an EL expression.

Table A-4 describes the attributes of the `PageDef.xml` `<executables>` elements.

Table A-4 Attributes of the PageDef.xml File <executables> Element

Element Syntax	Attributes	Attribute Description
`<accessorIterator>`	`Accessor`	Specifies any other accessor defined by this binding.
	`Binds`	Specifies the view or action to which the iterator is bound.
	`BeanClass`	Identifies the Java type of beans in the associated iterator or collection.
	`CacheResults`	If `true`, manages the data collection between requests.
	`ChangeEventRate`	Specifies the rate of events when a component is wired to data via this iterator and is in polling event mode.
	`DataControl`	Interprets and returns the collection referred to by this iterator binding.
	`id`	Unique identifier. May be referenced by any ADF value binding.
	`MasterBinding`	Reference to the `methodIterator` (or iterator) that binds the data collection that serves as the master to the accessor iterator's detail collection.
	`ObjectType`	Used for ADF Business Components only. A boolean value determines whether the collection is an object type or not.

Table A-4 (Cont.) Attributes of the PageDef.xml File <executables> Element

Element Syntax	Attributes	Attribute Description
	`RangeSize`	Specifies the number of data objects in a range to fetch from the bound collection. The range defines a window you can use to access a subset of the data objects in the collection. Typically, when you create a component by dragging an object from the Data Controls panel, the range size is set to 25. Use `RangeSize` when you want to work with an entire set or when you want to limit the number of data objects to display in the page. Note that the values -1 and 0 have specific meaning: the value -1 returns all available objects from the collection, while the value 0 will return the same number of objects as the collection uses to retrieve from its data source. If the `RangeSize` attribute is not specified at all, all available objects are returned.

Table A-4 (Cont.) Attributes of the PageDef.xml File <executables> Element

Element Syntax	Attributes	Attribute Description
	`Refresh`	Determines when to invoke the executable. Oracle recommends that you do not change from the default value (`deferred`) for the `Refresh` attribute and that you do use an EL expression for the `RefreshCondition` attribute to determine if an executable should be refreshed. For completeness' sake, the following is the list of valid attribute values: • `always`: Invokes the executable each time the binding container is prepared. This occurs when the page displays and when the user submits changes, or when the application posts back to the page. • `deferred`: Refresh occurs when another binding requires or refers to this executable. Since refreshing an executable may impact performance, you can set the refresh to occur only if `deferred` is used in a binding that is being rendered. This is the default value for ADF Faces applications. • `<default>`: Always set to `deferred` by default. • `ifNeeded`: Whenever the framework needs to refresh the executable because it has not been refreshed to this point. For example, when you have an accessor hierarchy such that a detail is listed first in the page definition, the master could be refreshed twice (once for the detail and again for the master's iterator). Using `ifNeeded` avoids duplicate refreshes. This is the default value for all view technologies other than ADF Faces. • `never`: The application itself calls refresh on the executable during one of the controller phases and does not want the framework to refresh the executable at all. • `prepareModel`: Invokes the executable each time the page's binding container is prepared. • `prepareModelIfNeeded`: Invokes the executable during the Prepare Model phase if this executable has not been refreshed to this point. See also `ifNeeded`. • `renderModel`: Invokes the executable each time the page is rendered. • `renderModelIfNeeded`: Invokes the executable during the page's Render Model phase on the condition that it is needed. See also `ifNeeded`.
	`RefreshCondition`	An EL expression that determines if the executable should be refreshed. Use with the executable's `Refresh` attribute to make the refresh of the executable predictable. The `Refresh` attribute is evaluated if the EL expression that you set for `RefreshCondition` returns `true`.
	`RefreshAfter`	Use to handle dependencies between executables. For example, you can set a condition so that this executable refreshes after another executable.

Table A-4 (Cont.) Attributes of the PageDef.xml File <executables> Element

Element Syntax	Attributes	Attribute Description
	`RowCountThreshold`	Specify a value to determine if a result set returns the number of rows you specify as a value. If you set `RowCountThreshold` to `0`, the iterator returns the estimated row count in the result set by executing the count query. If you set `RowCountThreshold` to less than `0`, the iterator does not execute the count query. Set `RowCountThreshold` to a value greater than `0` if you want the iterator to execute the count query with the maximum value equal to the value you specify for `RowCountThreshold`. If the estimated row count is less than the value of `RowCountThreshold`, return the number of rows in the estimated row count. If the estimated row count is greater than the value of RowCountThreshold, return `-1`.
	`Sortable`	Specifies whether the iterator is sortable or not.
`<invokeAction>`		Only use `<invokeAction>` when your application does not use the ADF Controller module from the Fusion web application technology stack. Oracle recommends that you use a task flow method call activity, as described in Using Method Call Activities, if your application uses the ADF Controller module.
	`Binds`	Determines the action to invoke. This may be on any `actionBinding`. Additionally, in the case, of the EJB session facade data control, you may bind to the finder method exposed by the data control. Built-in actions supported by the EJB session facade data control include: • `Execute` executes the bound action defined by the data collection. • `Find` retrieves a data object from a collection. • `First` navigates to the first data object in the data collection range. • `Last` navigates to the first data object in the data collection range. • `Next` navigates to the first data object in the data collection range. If the current range position is already on the last data object, then no action is performed. • `Previous` navigates to the first data object in the data collection range. If the current position is already on the first data object, then no action is performed. • `setCurrentRowWithKey` passes the row key as a `String` converted from the value specified by the input field. The row key is used to set the currency of the data object in the bound data collection. When passing the key, the URL for the form will not display the row key value. You may use this operation when the data collection defines a multipart attribute key. • `setCurrentRowWithKeyValue` is used as above, but when you want to use a primary key value instead of the stringified key.
	`id`	Unique identifier. May be referenced by any ADF action binding.

Table A-4 (Cont.) Attributes of the PageDef.xml File <executables> Element

Element Syntax	Attributes	Attribute Description
	Refresh	See Refresh for <accessorIterator>.
	RefreshCondition	See RefreshCondition for <accessorIterator>.
<iterator> and <methodIterator>	BeanClass	Identifies the Java type of beans in the associated iterator or collection.
	BindingClass	This is for backward compatibility to indicate which class implements the runtime for this binding definition. Not used in current JDeveloper release.
	Binds	See Binds for <invokeAction>.
	CacheResults	See CacheResults for <accessorIterator>.
	ChangeEventRate	Specifies the rate of events when a component is wired to data via this iterator and is in polling event mode.
	DataControl	Name of the DataControl usage in the bindingContext (.cpx) which this iterator is associated with.
	DefClass	Used internally by ADF.
	id	Unique identifier. May be referenced by any ADF value binding.
	ObjectType	Not used by EJB session facade data control (used by ADF Business Components only).
	RangeSize	See RangeSize for <accessorIterator>.
	Refresh	See Refresh for <accessorIterator>.
	RefreshAfter	Specifies the condition after which the page should be refreshed.
	RefreshCondition	See RefreshCondition for <accessorIterator>.
	RowCountThreshold	See RowCountThreshold for <accessorIterator>.
<page> and <variableIterator>	id	Unique identifier. In the case of <page>, refers to nested page or **region** that is included in this page. In the case of the <variableIterator> executable, the identifier may be referenced by any ADF value binding.
	ChangeEventRate	Specifies the rate of events when a component is wired to data via this iterator and is in polling event mode.
	path	Used by <page> executable only. Advanced, a fully qualified path that may reference another page's binding container.
	Refresh	See Refresh for <accessorIterator>.
	RefreshAfter	Specifies the condition after which the page should be refreshed.
	RefreshCondition	See RefreshCondition for <accessorIterator>.

Table A-5 describes the attributes of the PageDef.xml <bindings> element.

Table A-5   Attributes of the PageDef.xml File <bindings> Element

Element Syntax	Attributes	Attribute Description
`<action>`	`Action`	Fully qualified package name. Identifies the class for which the data control is created. In the case of the EJB session facade, this is the session bean.
	`BindingClass`	This is for backward compatibility to indicate which class implements the runtime for this binding definition. This is used by earlier versions of JDeveloper.
	`DataControl`	Name of the `DataControl` usage in the `bindingContext (.cpx)` which this `iteratorBinding` or `actionBinding` is associated with.
	`Execute`	Used by default when you drop an operation from the Data Controls panel in the automatically configured `ActionListener` property. It results in executing the action binding's operation at runtime.
	`InstanceName`	Specifies the instance name for the action.
	`IterBinding`	Specifies the `iteratorBinding` instance in this `bindingContainer` to which this binding is associated.
	`Outcome`	Use if you want to use the result of a method action binding (once converted to a `String`) as a JSF navigation outcome name.
`<attributeValues>`	`ApplyValidation`	Set to `true` by default. When `true`, `controlBinding` executes validators defined on the binding. You can set to `false` in the case of ADF Business Components, when running in local mode and the same validators are already defined on the corresponding attribute.
	`BindingClass`	This is for backward compatibility to indicate which class implements the runtime for this binding definition. This is used by earlier versions of JDeveloper.
	`ChangeEventPolicy`	Specifies the event strategy for the component when run with ADS (Active Data Services). Can be specified as push poll ppr none
	`ControlClass`	Used internally by ADF.
	`CustomInputHandler`	This is the class name for a `oracle.jbo.uicli.binding.JUCtrlValueHandler` implementation that is used to process the `inputValue` for a given value binding.
	`DefClass`	Used internally by ADF.
	`id`	Unique identifier. May be referenced by any ADF action binding.
	`IterBinding`	Refers to the `iteratorBinding` instance in this `bindingContainer` to which this binding is associated.

Table A-5 (Cont.) Attributes of the PageDef.xml File <bindings> Element

Element Syntax	Attributes	Attribute Description
	NullValueId	Refers to the entry in the message bundle for this bindingContainer that contains the String to indicate the null value in a list display.
<button>	ApplyValidation	Set to true by default. When true, controlBinding executes validators defined on the binding. You can set to false in the case of ADF Business Components, when running in local mode and when the same validators are already defined on the corresponding attribute.
	BindingClass	This is for backward compatibility to indicate which class implements the runtime for this binding definition. This is used by earlier versions of JDeveloper.
	BoolVal	Identifies whether the value at the zero index in the static value list in this boolean list binding represents true or false.
	ControlClass	Used internally by ADF.
	CustomInputHandler	This is the class name for a oracle.jbo.uicli.binding.JUCtrlValueHandler implementation that is used to process the inputValue for a given value binding.
	DefClass	Used internally by ADF.
	id	Unique identifier. May be referenced by any ADF action binding.
	IterBinding	Refers to the iteratorBinding instance in this bindingContainer to which this binding is associated.
	ListIter	Refers to the iteratorBinding that is associated with the source list of this listBinding.
	ListOperMode	Determines whether this list binding is for navigation, contains a static list of values or is an LOV type list.
	NullValueFlag	Describes whether this list binding has a null value and, if so, whether it should be displayed at the beginning or the end of the list.
	NullValueId	Refers to the entry in the message bundle for this bindingContainer that contains the String to indicate the null value in a list display.
<gantt>	ChangeEventPolicy	Identifies the update policy for the component. Valid values are none, ppr, and push. The default is ppr.
	xmlns	Used internally by ADF.
	CustomInputHandler	This is the class name for a oracle.jbo.uicli.binding.JUCtrlValueHandler implementation that is used to process the inputValue for a given value binding.
	id	Unique identifier. May be referenced by any ADF action binding.
	IterBinding	Refers to the iteratorBinding instance in this bindingContainer to which this binding is associated.

Table A-5 (Cont.) Attributes of the PageDef.xml File <bindings> Element

Element Syntax	Attributes	Attribute Description
	`NullValueId`	Refers to the entry in the message bundle for this `bindingContainer` that contains the `String` to indicate the null value in a list display.
	`ganttDataMap`	Wraps the data binding XML for the ADF Faces Gantt chart component.
`<gauge>`	`type`	Identifies the gauge type.
	`ledStyle`	Identifies the LED style for the gauge.
	`ChangeEventPolicy`	Identifies the update policy for the component. Valid values are `none`, `ppr`, and `push`. The default is `ppr`.
	`xmlns`	Used internally by ADF.
	`CustomInputHandler`	This is the class name for a `oracle.jbo.uicli.binding.JUCtrlValueHandler` implementation that is used to process the `inputValue` for a given value binding.
	`id`	Unique identifier. May be referenced by any ADF action binding.
	`IterBinding`	Refers to the `iteratorBinding` instance in this `bindingContainer` to which this binding is associated.
	`NullValueId`	Refers to the entry in the message bundle for this `bindingContainer` that contains the `String` to indicate the null value in a list display.
	`gaugeDataMap`	Wraps the data binding XML for the ADF Faces `gauge` component.
`<graph>`	`type`	Identifies the graph type.
	`ChangeEventPolicy`	Identifies the update policy for the component. Valid values are `none`, `ppr`, and `push`. The default is `ppr`.
	`xmlns`	Used internally by ADF.
	`CustomInputHandler`	This is the class name for a `oracle.jbo.uicli.binding.JUCtrlValueHandler` implementation that is used to process the `inputValue` for a given value binding.
	`id`	Unique identifier. May be referenced by any ADF action binding.
	`IterBinding`	Refers to the `iteratorBinding` instance in this `bindingContainer` to which this binding is associated.
	`NullValueId`	Refers to the entry in the message bundle for this `bindingContainer` that contains the `String` to indicate the null value in a list display.
	`graphDataMap`	Wraps the data binding XML for the ADF Faces `graph` component.

Table A-5 (Cont.) Attributes of the PageDef.xml File <bindings> Element

Element Syntax	Attributes	Attribute Description
`<list>`	`ApplyValidation`	Set to `true` by default. When `true`, `controlBinding` executes validators defined on the binding. You can set to `false` in the case of ADF Business Components, when running in local mode and when the same validators are already defined on the corresponding attribute.
	`BindingClass`	This is for backward compatibility to indicate which class implements the runtime for this binding definition. This is used by earlier versions of JDeveloper.
	`ControlClass`	Used internally by ADF.
	`CustomInputHandler`	This is the class name for a `oracle.jbo.uicli.binding.JUCtrlValueHandler` implementation that is used to process the `inputValue` for a given value binding.
	`DefClass`	Used internally by ADF.
	`id`	Unique identifier. May be referenced by any ADF action binding.
	`IterBinding`	Refers to the `iteratorBinding` instance in this `bindingContainer` to which this binding is associated.
	`ListIter`	Refers to the `iteratorBinding` that is associated with the source list of this `listBinding`.
	`ListOperMode`	Determines whether this list binding is for navigation, contains a static list of values, or is an LOV type list.
	`Mode`	The value of this attribute determines if the list binding passes the actual value (`Object`) or the location of the value in an index (`Index`). For more information about setting this attribute, see What You May Need to Know About Values in a Selection List .
	`MRUCount`	Specifies the number of items to display in a choice list when you want to provide a shortcut for the end user to display their most recent selections. For example, a form might display a choice list of supplier ID values to drive a purchase order form. In this case, you can allow users to choose from a list of their most recently view suppliers, where the number of supplier choices is determined by the count you enter. The default for the choice list is to display all values for the attribute and is specified by the count 0 (zero).
	`MRUId`	Specifies the `String` that will be the discriminator line for the MRU list.
	`NullValueFlag`	Describes whether this list binding has a null value and, if so, whether it should be displayed at the beginning of the list or the end.
	`NullValueId`	Refers to the entry in the message bundle for this `bindingContainer` that contains the `String` to indicate the null value in a list display.
	`StaticList`	Defines a static list of values that will be rendered in the bound list component.

Table A-5 (Cont.) Attributes of the PageDef.xml File <bindings> Element

Element Syntax	Attributes	Attribute Description
`<mapTheme>`	`ChangeEventPolicy`	Identifies the update policy for the component. Valid values are `none`, `ppr`, and `push`. The default is `ppr`.
	`xmlns`	Used internally by ADF.
	`CustomInputHandler`	This is the class name for a `oracle.jbo.uicli.binding.JUCtrlValueHandler` implementation that is used to process the `inputValue` for a given value binding.
	`id`	Unique identifier. May be referenced by any ADF action binding.
	`IterBinding`	Refers to the `iteratorBinding` instance in this `bindingContainer` to which this binding is associated.
	`NullValueId`	Refers to the entry in the message bundle for this `bindingContainer` that contains the `String` to indicate the null value in a list display.
	`mapThemeDataMap`	Wraps the data binding XML for the ADF Data Visualization geographic map component.
`<methodAction>`	`Action`	Fully qualified package name. Identifies the class for which the data control is created. In the case of the EJB session facade, this is the session bean.
	`BindingClass`	This is for backward compatibility to indicate which class implements the runtime for this binding definition. This is used by earlier versions of JDeveloper.
	`ClassName`	This is the class to which the method being invoked belongs.
	`DataControl`	Name of the `DataControl` usage in the `bindingContext (.cpx)` which this `iteratorBinding` or `actionBinding` is associated with.
	`DefClass`	Used internally by ADF.
	`id`	Unique identifier. May be referenced by any ADF action binding.
	`InstanceName`	A dot-separated EL path to a Java object instance on which the associated method is to be invoked.
	`IsLocalObjectReference`	Set to `true` if the `instanceName` contains an EL path relative to this `bindingContainer`.
	`IsViewObjectMethod`	Set to `true` if the `instanceName` contains an instance path relative to the associated data control's application module.
	`MethodName`	Indicates the name of the operation on the given instance or class that needs to be invoked for this `methodActionBinding`.
	`RequiresUpdateModel`	Whether this action requires that the model be updated before the action is to be invoked.
	`ReturnName`	The EL path of the result returned by the associated method.

Table A-5 (Cont.) Attributes of the PageDef.xml File <bindings> Element

Element Syntax	Attributes	Attribute Description
`<pivotTable>`	`ChangeEventPolicy`	Identifies the update policy for the component. Valid values are `none`, `ppr`, and `push`. The default is `ppr`.
	`xmlns`	Used internally by ADF.
	`CustomInputHandler`	This is the class name for a `oracle.jbo.uicli.binding.JUCtrlValueHandler` implementation that is used to process the `inputValue` for a given value binding.
	`id`	Unique identifier. May be referenced by any ADF action binding.
	`IterBinding`	Refers to the `iteratorBinding` instance in this `bindingContainer` to which this binding is associated.
	`NullValueId`	Refers to the entry in the message bundle for this `bindingContainer` that contains the `String` to indicate the null value in a list display.
	`pivotTableDataMap`	Wraps the data binding XML for the ADF Data Visualization pivot table component.
`<table>` and `<tree>`	`ApplyValidation`	Set to `true` by default. When `true`, `controlBinding` executes validators defined on the binding. You can set to `false` in the case of ADF Business Components, when running in local mode and when the same validators are already defined on the corresponding attribute.
	`BindingClass`	This is for backward compatibility to indicate which class implements the runtime for this binding definition.This is used by earlier versions of JDeveloper.
	`CollectionModel`	Accesses the `CollectionModel` object, the data model that is used by ADF table components. A table's value is bound to the `CollectionModel` attribute. The table wraps the result set from the iterator binding in a `CollectionModel` object. The `CollectionModel` attribute allows each item in the collection to be available within the table component using the `var` attribute.
	`ControlClass`	Used internally for testing purposes.
	`CustomInputHandler`	This is the class name for a `oracle.jbo.uicli.binding.JUCtrlValueHandler` implementation that is used to process the `inputValue` for a given value binding.
	`DefClass`	Used internally by ADF.
	`DiscrValue`	Indicates the discriminator value for a hierarchical type binding (type definition for a tree node). This value is used to determine whether a given row in a collection being rendered in a polymorphic tree binding should be rendered using the containing hierarchical type binding.
	`id`	Unique identifier. May be referenced by any ADF action binding.
	`IterBinding`	Refers to the `iteratorBinding` instance in this `bindingContainer` to which this binding is associated.

Table A-5 (Cont.) Attributes of the PageDef.xml File <bindings> Element

Element Syntax	Attributes	Attribute Description
	`TreeModel`	The data model used by ADF Tree components. `TreeModel` extends `CollectionModel` to add support for container rows. Rows in the `TreeModel` may (recursively) contain other rows.

adfc-config.xml

ADF Faces uses adfc-config.xml to store its configurations. It is an unbounded taskflow and the `adfc-config.xml` file contains activies, control flow rules, and managed beans interacting to allow a user to complete a task.

The `adfc-config.xml` file is the source file for the ADF unbounded task flow that JDeveloper creates by default when you create an application using the ADF Fusion Web Application template. By default, JDeveloper stores the `adfc-config.xml` file in the following directory:

`application_root``\ViewController\Web Content\WEB-INF`

If you create additional unbounded task flows, JDeveloper proposes the following file name for the source files of the additional unbounded task flows:

`adfc-config`*`N`*`.xml`

where *N* is a number that increments each time you create a new unbounded task flow. Alternatively, you choose the file name you want for an unbounded task flow's source file.

Each source file for an unbounded task flow contains the metadata for activities, control flow rules, and managed beans that you added to an unbounded task flow so that end users can interact with the Fusion web application to complete a task.

The XML schema definition file (XSD) that defines valid metadata entries for the `adfc-config.xml` file is `adfc-config_1_0.xsd`. The `adfc-config_1_0.xsd` file is stored in the `adf-controller-schema.jar` file in the following directory of your JDeveloper installation:

`jdev_install``\oracle_common\modules\oracle.adf.model_12.1.2`

Use JDeveloper's XSD Visual Editor to view the `adfc-config_1_0.xsd` file. Using this editor, you can identify the valid metadata elements and attributes for the `adfc-config.xml` file.

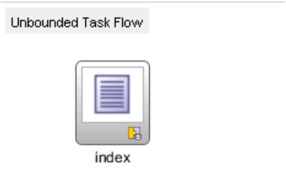
The following example shows the source file for an unbounded task flow where view activities, control flow rules, and so on have yet to be added.

```
<?xml version="1.0" encoding="windows-1252" ?>
<adfc-config xmlns="http://xmlns.oracle.com/adf/controller" version="1.2">
</adfc-config>
```

Figure A-4 shows the Diagram view of the Summit sample application for ADF task flows' `adfc-config.xml` file. Example A-3 shows the corresponding source view of the unbounded task flow that appears in Figure A-4.

For more information about unbounded task flows, see Getting Started with ADF Task Flows .

Figure A-4 Diagram View of Summit ADF Task Flow Sample Application´s adfc-config.xml File



Example A-3 Source View of Summit ADF Task Flow Sample Application´s adfc-config.xml File

```
<?xml version="1.0" encoding="windows-1252" ?>
<adfc-config xmlns="http://xmlns.oracle.com/adf/controller" version="1.2">
  <view id="index">
    <page>/index.jsf</page>
  </view>
  <managed-bean id="__1">
    <managed-bean-name>login</managed-bean-name>
    <managed-bean-class>oracle.summit.bean.LoginBean</managed-bean-class>
    <managed-bean-scope>request</managed-bean-scope>
  </managed-bean>
  <managed-bean id="__2">
    <managed-bean-name>WelcomePageBean</managed-bean-name>
    <managed-bean-class>oracle.summit.bean.WelcomePageBean</managed-bean-class>
    <managed-bean-scope>request</managed-bean-scope>
  </managed-bean>
</adfc-config>
```

task-flow-definition.xml

The task-flow-definition.xml file stores metadata definition for a bounded task flow. You can specify the task flow definition XML filename and location when you generate the task flow.

The default file name that JDeveloper proposes for the source file of the first ADF bounded task flow that you create is `task-flow-definition.xml`. This file stores the metadata for a bounded task flow. It contains entries for each view activity, method call activity, task flow call, and so on that you add to the bounded task flow.

JDeveloper proposes the following file name for subsequent bounded task flows that you create:

`task-flow-definition`*`N`*`.xml`

where *N* is a number that increments each time you create a new bounded task flow. Alternatively, you choose the file name you want for a bounded task flow's source file. By default, JDeveloper stores the `task-flow-definition.xml` file in the following directory:

`application_root``\ViewController\Web Content\WEB-INF`

The XML schema definition files (XSD) that define the valid metadata entries for the source file of a bounded task flow are stored in the `adf-controller-schema.jar` file in the following directory of your JDeveloper installation:

`jdev_install``\oracle_common\modules\oracle.adf.model_12.1.2`

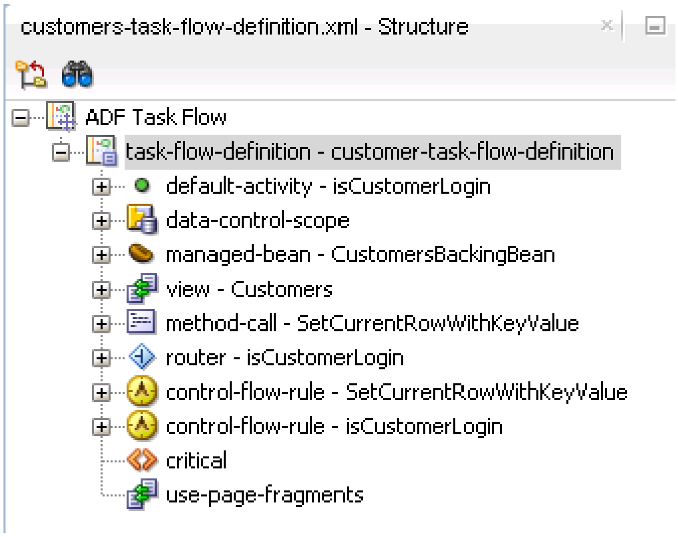
Use JDeveloper's XSD Visual Editor to view these XSD files and identify the valid metadata elements and attributes for the source file of a bounded task flow.

The following example shows the content of a source file when you create a new bounded task flow using the Create Task Flow dialog. The `<task-flow-definition>` element identifies this file as a bounded task flow. JDeveloper generates the `<task-flow-definition>` element when you select the **Create as Bounded Task Flow** checkbox in the Create Task Flow dialog. The value of the `<task-flow-definition>` element's `id` attribute (`task-flow-definition`) corresponds to the value that you enter in the **Task Flow ID** field of the Create Task Flow dialog. The `<use-page-fragments/>` element indicates that all view activities in this bounded task flow must be associated with page fragments. JDeveloper generates this entry when you select the **Create with Page Fragments** checkbox in the Create Task Flow dialog.

```
<?xml version="1.0" encoding="windows-1252" ?>
<adfc-config xmlns="http://xmlns.oracle.com/adf/controller" version="1.2">
  <task-flow-definition id="task-flow-definition">
    <use-page-fragments/>
  </task-flow-definition>
</adfc-config>
```

Example A-4 shows the Summit ADF task flow sample application's `customers-task-flow-definition.xml` file. You can view entries for the activities shown in the Structure window in Figure A-5. See Getting Started with ADF Task Flows .

Figure A-5 Structure View of the customers-task-flow-definition.xml File



customers-task-flow-definition.xml - Structure

- ADF Task Flow
  - task-flow-definition - customer-task-flow-definition
    - default-activity - isCustomerLogin
    - data-control-scope
    - managed-bean - CustomersBackingBean
    - view - Customers
    - method-call - SetCurrentRowWithKeyValue
    - router - isCustomerLogin
    - control-flow-rule - SetCurrentRowWithKeyValue
    - control-flow-rule - isCustomerLogin
    - critical
    - use-page-fragments

Example A-4 Source View of the customers-task-flow-definition.xml File

```
<?xml version="1.0" encoding="windows-1252" ?>
<adfc-config xmlns="http://xmlns.oracle.com/adf/controller" version="1.2">
  <task-flow-definition id="customer-task-flow-definition">
```

```
    <default-activity id="__1">isCustomerLogin</default-activity>
    <managed-bean id="__11">
      <managed-bean-name id="__9">CustomersBackingBean</managed-bean-name>
      <managed-bean-class id="__10">oracle.summit.backing.CustomersBackingBean</managed-bean-
class>
      <managed-bean-scope id="__8">backingBean</managed-bean-scope>
    </managed-bean>
    <view id="Customers">
      <page>/customers/Customers.jsff</page>
    </view>
    <method-call id="SetCurrentRowWithKeyValue">
      <method>#{bindings.setCurrentRowWithKeyValue.execute}</method>
      <outcome>
        <fixed-outcome>setCurrentRowWithKey</fixed-outcome>
      </outcome>
    </method-call>
    <router id="isCustomerLogin">
      <case id="__4">
        <expression>#{securityContext.userInRole['Application Customer Role']}</expression>
        <outcome>customerLogin</outcome>
      </case>
      <default-outcome>notCustomerLogin</default-outcome>
    </router>
    <control-flow-rule id="__2">
      <from-activity-id>SetCurrentRowWithKeyValue</from-activity-id>
      <control-flow-case id="__3">
        <from-outcome>setCurrentRowWithKey</from-outcome>
        <to-activity-id>Customers</to-activity-id>
      </control-flow-case>
    </control-flow-rule>
    <control-flow-rule id="__5">
      <from-activity-id>isCustomerLogin</from-activity-id>
      <control-flow-case id="__6">
        <from-outcome>notCustomerLogin</from-outcome>
        <to-activity-id>Customers</to-activity-id>
      </control-flow-case>
      <control-flow-case id="__7">
        <from-outcome>customerLogin</from-outcome>
        <to-activity-id>SetCurrentRowWithKeyValue</to-activity-id>
      </control-flow-case>
    </control-flow-rule>
    <critical/>
    <use-page-fragments/>
  </task-flow-definition>
</adfc-config>
```

adf-config.xml

The adf-config.xml file contains application-level settings, that manages the runtime infrastructure such as failover behavior for the ADF application modules, global fetch limit for all the view objects, caching of resource bundles, automated refresh of page bindings, and so on for your application.

JDeveloper generates the `adf-config.xml` file when you create an application using the ADF Fusion Web Application template. JDeveloper stores this application-level descriptor file in the following directory:

application_root\.adf\META-INF

In JDeveloper, you can locate the file in the Application Resources panel of the Applications window by expanding the **Descriptors > ADF META-INF** nodes.

The `adf-config.xml` file specifies application-level settings that you define during development. You can use a deployment profile to specify settings that are used at application deploy time. You can change some of the settings at runtime using Oracle Enterprise Manager.

Examples of tasks that you can accomplish by changing settings in the `adf-config.xml` file include the following:

- Enable or disable the validation of ADF Controller metadata
- Replicate memory scope if you deploy your Fusion web application in a clustered environment
- Configure properties to manage the caching of resource bundles where your application uses EL expressions to retrieve strings at runtime from resource bundles

  The properties that you can configure are:

  - `initial-size`

    Specifies the initial number of resource bundles that your application can cache. The default value is `100`.

  - `max-size`

    Specifies the maximum number of resource bundles that your application can cache. The default value is `100`.

  - `load-factor`

    The default value is `0.75`.

  - `expire-time`

    The default value is `43200` seconds (12 hours).

    You specify these properties as attribute values of the `<resource-bundle-cache>` element in the `adf-config.xml` file. Example A-5 demonstrates how you might configure these values for your application in the `adf-config.xml` file.

  As an alternative to configuring the caching of resource bundles in the `adf-config.xml` file, you can specify the resource bundle caching properties as parameters for the Java Virtual Machine (JVM). If you specify the properties as parameters for the JVM, the changes apply to all applications managed by the JVM. For this reason, consider configuring the resource bundle caching properties in the `adf-config.xml` file for you application. Use the following property names if you decide to specify the resource bundle caching properties as parameters for the JVM:

  - `resource-bundle-cache-initial-size`
  - `resource-bundle-cache-max-size`
  - `resource-bundle-cache-load-factor`
  - `resource-bundle-cache-expire-time`

At runtime, the Fusion web application loads the `adf-config.xml` file from the `META-INF` directory. If the Fusion web application finds more than one `adf-config.xml` file, it stops loading the file and logs a warning.

The following tasks modify or require you to modify the `adf-config.xml` file:

- In the ADF Controller layer
  - Creating task flows

    By default, the ADF Controller uses a user session level cache to store the application's unbounded task flow metadata. This enables support for user specific seeded customization or design time at runtime customization of unbounded task flow metadata in your application. Clear the **Allow Unbounded Task Flow Customizations** checkbox in the Controller page of the `adf-config.xml`'s file overview editor to disable this behavior and set the value of the `<allow-unbounded-task-flow-customizations>` element to `false`. You can still have customizations of the unbounded task flow metadata in your application. However, these customizations are shared by all users of the application. For information about customizations, see Customizing Applications with MDS . For information about task flows, see Getting Started with ADF Task Flows .
  - Configure an integer value for the `root-view-port-request-lock-timeout` property that specifies the number of seconds a request can hold a lock on the root view port before timing out and allowing other requests to acquire the lock. The default value is 600 seconds. For information about the root view port, see About View Ports and ADF Regions.
  - Minimize the number of active root view ports in an application by specifying a value for the `<max-root-view-ports>` property. See Minimizing the Number of Active Root View Ports in an Application.
  - Modify the number of events collected or stop the collection of events by ADF Controller. Both tasks involve configuring the value of the `<debug-history-size>` element in your application's `adf-config.xml` file. See Reporting Incidents to the Diagnostic Framework.
  - Enabling implicit save points

    See How to Enable Implicit Save Points.
  - Specify a bounded task flow that does not require authorization to invoke when a user attempts to invoke a bounded task flow in a region for which they do not have security permissions. See Handling Access to Secured Task Flows by Unauthorized Users.
  - Enable ADF Controller to track changes to ADF memory scopes and replicate the page flow scope and view scope within the server cluster. You must set the ADF Controller parameter `<adf-scope-ha-support>` in the application's `adf-config.xml` file to `true`. See How to Ensure Oracle ADF Receives Notifications From Managed Bean Changes .
- In the ADF Model layer
  - Disable automatic partial page rendering as the default behavior for an application. This requires you to change the value for the `changeEventPolicy` attribute from the default value of `ppr`:

    ```
    <defaults changeEventPolicy="ppr"/>
    ```

    See What You May Need to Know About Partial Page Rendering and Iterator Bindings.

  - Configure all Query components in the user interface project to render input search fields using a bind variable.

    See What You May Need to Know About Named Bind Variables in Search Forms.

  - Control the dispatch of contextual events

    See How to Control Contextual Events Dispatch.

- Persisting saved searches in MDS

  See How to Persist Saved Searches into MDS.

- Creating RESTful web services from ADF application modules and managing the release version identifiers of REST resources

  See Creating ADF REST Resources Using the Application Module and see Versioning the ADF REST Resource.

- Configuring ADF Business Components global settings

  See Using Fetch Size to Limit the Maximum Number of Records Fetched for a View Object and How to Initialize the Data Model Project With a Database Connection.

- Enabling or disabling Oracle ADF Security

  See Enabling ADF Security and Disabling ADF Security.

- Enabling seeded customizations

  See How to Enable Seeded Customizations for User Interface Projects and How to Enable Seeded Customizations in Existing Pages.

- Configuring change persistence

  See How to Enable User Customizations.

- Enabling user customizations

  See What Happens When You Enable User Customizations.

Example A-5 shows example entries for an application's `adf-config.xml` file.

Example A-5   Sample adf-config.xml File

```
<?xml version="1.0" encoding="windows-1252" ?>
<adf-config xmlns="http://xmlns.oracle.com/adf/config"
            xmlns:sec="http://xmlns.oracle.com/adf/security/config">
  <sec:adf-security-child xmlns="http://xmlns.oracle.com/adf/security/config">
    <CredentialStoreContext credentialStoreClass=

"oracle.adf.share.security.providers.jps.CSFCredentialStore"
                                      credentialStoreLocation="../../src/META-
INF/jps-config.xml"/>
    <!-- The Configure ADF Security wizard modifies JaasSecurityContext settings
-->
    <JaasSecurityContext initialContextFactoryClass=

"oracle.adf.share.security.JAASInitialContextFactory"
                         jaasProviderClass=

"oracle.adf.share.security.providers.jps.JpsSecurityContext"
```

```
                         authorizationEnforce="true"
                         authenticationRequire="true"/>
  </sec:adf-security-child>
  <adf-controller-config xmlns="http://xmlns.oracle.com/adf/controller/config">
    <savepoint-datasource>java:comp/env/jdbc/summit_adfDS</savepoint-datasource>
    <enable-implicit-savepoints>true</enable-implicit-savepoints>
    <unauthorized-region-taskflow>/WEB-INF/task-flow-definition.xml</
unauthorized-region-taskflow>
  </adf-controller-config>
  <adf-faces-config xmlns="http://xmlns.oracle.com/adf/faces/config">
    <persistent-change-manager>
      <persistent-change-manager-class>
           oracle.adf.view.rich.change.MDSDocumentChangeManager
      </persistent-change-manager-class>
    </persistent-change-manager>
    <taglib-config>
      <taglib uri="http://xmlns.oracle.com/adf/faces/rich">
        <tag name="calendar">
          <attribute name="activeDay">
            <persist-changes>true</persist-changes>
          </attribute>
        </tag>
        <!-- Additional tags omitted to make this example concise -->
        <tag name="table">
          <attribute name="filterVisible">
            <persist-changes>true</persist-changes>
          </attribute>
        </tag>
      </taglib>
    </taglib-config>
  </adf-faces-config>
  <adf-mds-config xmlns="http://xmlns.oracle.com/adf/mds/config">
    <mds-config xmlns="http://xmlns.oracle.com/mds/config" version="11.1.1.000">
      <cust-config>
        <match path="/">
          <customization-class name="oracle.adf.share.config.UserCC"/>
        </match>
      </cust-config>
    </mds-config>
  </adf-mds-config>
  <adf-adfm-config xmlns="http://xmlns.oracle.com/adfm/config">
    <defaults rowLimit="100"/>
    <startup>
      <amconfig-overrides>
        <config:Database jbo.SQLBuilder="Oracle" jbo.locking.mode="optimistic"/>
      </amconfig-overrides>
    </startup>
  </adf-adfm-config>
<!-- Properties to manage the caching of a resource bundle in your application
-->
<adf-resourcebundle-config xmlns="http://xmlns.oracle.com/adf/resourcebundle/
config">
    <applicationBundleName>
           path-to-resource-bundle/bundle-name
    </applicationBundleName>
    <resource-bundle-cache initial-size="20" max-size="100" expire-time="30000"
load-factor=".75"/>
    <bundleList>
      <bundleId override="true">
             package.BundleID
      </bundleId>
    </bundleList>
  </adf-resourcebundle-config>
</adf-config>
```

adf-settings.xml

The adf-settings.xml file is used by ADF application developers to extend and customize the ADF lifecycle with custom ADF phase listeners.

The `adf-settings.xml` file holds project-level and library-level settings such as ADF Faces help providers and ADF Controller phase listeners.

The configuration settings for `adf-settings.xml` are fixed and cannot be changed during or after application deployment. There can be multiple `adf-settings.xml` files in an application. The users of `adf-settings.xml` files are responsible for merging the contents of their configuration.

JDeveloper creates an `adf-settings.xml` file when you:

- Create an application using the ADF Fusion Web Application template
- Add ADF Page Flow (ADF Controller) as a feature to an existing project

By default, JDeveloper stores the `adf-settings.xml` file in the following directory:

```
application_root\ViewController\src\META-INF
```

The following tasks modify or require you to modify the `adf-settings.xml` file:

- Registering a phase listener

  See How to Register a Listener Globally.
- Creating help for ADF Faces components

  See Displaying Help for Components in *Developing Web User Interfaces with Oracle ADF Faces*.

For information about how to edit the `adf-settings.xml` file, see How to Configure for ADF Faces in adf-settings.xml in *Developing Web User Interfaces with Oracle ADF Faces*.

Example A-6 shows a sample `adf-setting.xml` file with settings configured for a phase listener and a help provider.

Example A-6 Sample adf-settings.xml File

```
<?xml version="1.0" encoding="windows-1252" ?>
<adf-settings xmlns="http://xmlns.oracle.com/adf/config">
  <adfc-controller-config xmlns="http://xmlns.oracle.com/adf/controller/config">
    <lifecycle>
      <phase-listener>
        <listener-id>SummitPhaseListener</listener-id>
        <class>oracle.summit.listeners.SummitPhaseListener</class>
      </phase-listener>
    </lifecycle>
  </adfc-controller-config>
  <adf-faces-config>
    <help-provider prefix="MYAPP">
      <help-provider-class>oracle.summit.MyHelpProvider</help-provider-class>
      <property>
        <property-name>myCustomProperty</property-name>
      <value>someValue</value>
    </property>
  </help-provider>
</adf-faces-config>
</adf-settings>
```

web.xml

The web.xml file is used as a deployment descriptor file by Java web applications to determine how URLs map to servlets, which URLs require authentication, and other information. This file is an XML document that defines everything about your ADF application that a server needs to know.

Oracle ADF has specific configuration settings for the standard `web.xml` deployment descriptor file.

When you create a project in JDeveloper that uses JSF technology, a starter `web.xml` file with default settings is created for you in the `/WEB-INF` folder. To edit the file, double-click **web.xml** in the Applications window to open it in the XML editor.

The following must be configured in `web.xml` for all applications that use JSF and ADF Faces:

- ADF Library resource servlet and mapping: Serves up web application resources (images, style sheets, JavaScript libraries) from ADF Library JAR files on the application class path.

  By default, static web application resources (including resources in ADF Libraries), such as images, have a staleness period setting of 364 days. This staleness period setting requests that the client not make a request to the server to validate the static web application resources until one of the following events occur:

  - Staleness period expires
  - Browser cache no longer contains the static resource
  - User executes a page refresh

  If the client does make a request to the server and the resources have not changed, the server returns a HTTP 304 response (Not Modified) with no body.

  You can override the staleness period setting of 364 days for all static web application resources in ADF Libraries by adding initialization parameters to the `web.xml` file. The following example demonstrates how to set these initialization parameters using a number of examples.

  ```
  <!-- Expires all static resources in 30 days -->
   <init-param>
     <param-name>expires</param-name>
     <param-value>60*60*24*30</param-value>
   </init-param>

  <!-- Expires static resources with the .js file extension in 1 day -->
   <init-param>
     <param-name>expires.js</param-name>
     <param-value>60*60*24</param-value>
   </init-param>

  <!-- Turns off staleness setting. Use for testing -->
   <init-param>
     <param-name>expires</param-name>
  ```

```
    <param-value>OFF</param-value>
  </init-param>
```

The following example shows the default file extensions. You can add additional file extensions by configuring the `extensions` parameter value to include the file extensions that you want. Note that you must prepend and append "`.`" to the file extension. The example also shows how you specify visibility for cache control.

```
<!-- Specify file extensions for a number of different file formats -->
  <init-param>
    <param-name>extensions</param-name>
    <param-value>.png.jpg.jpeg.gif.js.css.</param-value>
  </init-param>

<!-- Specify visibility for cache control, for example -->
  <init-param>
    <param-name>visibility</param-name>
    <param-value>Public</param-value>
  </init-param>
```

- JSF servlet and mapping: The servlet `javax.faces.webapp.FacesServlet` that manages the request-processing lifecycle for web applications utilizing JSF to construct the user interface.
- ADF Faces filter and mapping: A servlet filter to ensure that ADF Faces is properly initialized by establishing a `AdfFacesContext` object. This filter also processes file uploads.

The JSF servlet and mapping configuration settings are automatically added to the starter `web.xml` file when you first create a JSF project. When you insert an ADF Faces component into a JSF page for the first time, JDeveloper automatically inserts the configuration settings for ADF Faces filter and mapping, and resource servlet and mapping. See ADF Faces Configuration in *Developing Web User Interfaces with Oracle ADF Faces*.

logging.xml

ADF Logger is integrated in the WebLogic enterprise manager, and gives you the flexibility to adjust your log-levels at runtime. You can configure the logging session by editing the logging.xml file.

ADF Logger is a diagnostic tool that you can use in JDeveloper to capture runtime traces messages when you debug an application. You configure the use of this tool by editing the `logging.xml` file.

For information about the `logging.xml` file and using the ADF Logger, see Using the ADF Logger.

B

Oracle ADF Binding Properties

This appendix describes the properties of the **ADF bindings** that you use can access at runtime in an **Oracle ADF** application.

Table B-1 shows the properties that you can use in EL expressions to access values of the ADF binding objects at runtime. The properties appear in alphabetical order.

Table B-1 EL Properties of Oracle ADF Bindings

Runtime Property	Description	Iterator	Action	Attribute	Button	List	Table	Tree
`actionEnabled`	Use `operationEnabled` instead.	n/a	yes	n/a	n/a	n/a	n/a	n/a
`allRowsInRange`	Returns an array of the current set of rows from the associated collection. Calls `getAllRowsInRange()` on the `RowSetIterator` object.	yes	n/a	n/a	n/a	n/a	n/a	n/a
`attributeDef`	Returns the attribute definition for the first attribute with which the binding is associated.	n/a	n/a	yes	yes	yes	n/a	n/a
`attributeDefs`	Returns the attribute definitions for all the attributes to which the binding is associated.	n/a	n/a	yes	yes	yes	n/a	n/a
`attributeValue`	Returns an unformatted and typed (appropriate Java type) value in the current row, for the attribute to which the control binding is bound.	n/a	n/a	yes	yes	yes	n/a	n/a
`attributeValues`	Returns the value of all the attributes to which the binding is associated in an ordered array. Returns an array of unformatted and typed (appropriate Java type) values in the current row for all the attributes to which the control binding is bound.	n/a	n/a	yes	yes	yes	n/a	n/a
`bindings`	Returns a new binding for each cell or attribute exposed under the rows of a tree node binding.	no	no	no	no	no	no	yes
`children`	Returns the child nodes of a tree node binding.	n/a	n/a	n/a	n/a	n/a	n/a	yes
`currentRow`	Returns the current row on an action binding bound to an iterator (for example, built-in navigation actions).	n/a	yes	n/a	n/a	n/a	n/a	n/a

Table B-1 (Cont.) EL Properties of Oracle ADF Bindings

Runtime Property	Description	Iterator	Action	Attribute	Button	List	Table	Tree
`dataControl`	Returns the iterator's associated data provider.	yes	n/a	n/a	n/a	n/a	n/a	n/a
`displayData`	Returns a list of map elements. Each map entry contains the following elements: • `selected`: A boolean `true` if the current entry should be selected. • `index`: The index value of the current entry. • `prompt`: A string value that may be used to render the entry in the UI. • `displayValues`: An ordered list of display attribute values for all display attributes in the list binding.	n/a	n/a	n/a	n/a	yes	n/a	n/a
`displayHint`	Returns the display hint for the first attribute to which the binding is associated. The hint identifies whether the attribute should be displayed or not. For more information, see `oracle.jbo.AttributeHints.displayHint`.	n/a	n/a	n/a	n/a	yes	n/a	n/a

Table B-1 (Cont.) EL Properties of Oracle ADF Bindings

Runtime Property	Description	Iterator	Action	Attribute	Button	List	Table	Tree
`displayHints`	Returns a list of name-value pairs for UI hints for all display attributes to which the binding is associated. The map contains the following elements: • `label`: The label to display for the current attribute. • `tooltip`: The tooltip to display for the current attribute. • `displayHint`: The display hint for the current attribute. • `displayHeight`: The height in lines for the current attribute. • `displayWidth`: The width in characters for the current attribute. • `controlType`: The control type hint for the current attribute. • `format`: The format to be used for the current attribute.	n/a	n/a	n/a	yes	yes	n/a	n/a
`enabled`	Use the `operationEnabled` property.	n/a	n/a	n/a	n/a	n/a	n/a	n/a
`enabledString`	Returns `disabled` if the action binding is not ready to be invoked. Otherwise, returns an empty string (`""`).	n/a	yes	n/a	n/a	n/a	n/a	n/a
`error`	Returns any exception that was cached while updating the associated attribute value for a value binding or when invoking an operation bound by an operation binding.	yes	yes	yes	yes	yes	yes	yes
`estimatedRowCount`	Returns the maximum row count of the rows in the collection with which this **iterator binding** is associated	yes	n/a	n/a	n/a	n/a	yes	yes
`findMode`	Returns `true` if the iterator is currently operating in find mode. Otherwise, returns `false`.	yes	n/a	n/a	n/a	n/a	n/a	n/a

Table B-1 (Cont.) EL Properties of Oracle ADF Bindings

Runtime Property	Description	Iterator	Actio n	Attribute	Button	List	Table	Tree
`fullName`	Returns the fully qualified name of the binding object in the Oracle ADF **binding context**.	yes	yes	yes	yes	yes	yes	yes
`hints`	Returns the value of the UI hint indicated for the binding. See `displayHints` for the list of UI hint keywords.	yes	yes	yes	yes	yes	yes	yes
`inputValue`	Returns the value of the first attribute to which the binding is associated. If the binding was used to set the value on the attribute and the set operation failed, this method returns the invalid value that was being set.	n/a	n/a	yes	yes	yes	yes	yes
`iteratorBinding`	Returns the iterator binding that provides access to the data collection.	n/a	yes	yes	yes	yes	yes	yes
`label`	Returns the label (if supplied by control hints) for the first attribute of the binding.	n/a	n/a	yes	yes	yes	n/a	n/a
`labels`	Returns a map of labels (if supplied by control hints) keyed by attribute name for all attributes to which the binding is associated.	n/a	n/a	yes	yes	yes	yes	n/a
`labelSet`	Returns an ordered set of labels for all the attributes to which the binding is associated.	n/a	n/a	yes	yes	yes	yes	n/a
`mandatory`	Returns whether the first attribute to which the binding is associated is required.	n/a	n/a	yes	yes	yes	n/a	n/a
`name`	Returns the name of the binding object in the context of the **binding container** to which it is registered. Note this property is not visible in the EL expression builder dialog.	yes	yes	yes	yes	yes	yes	yes
`operationEnabled`	Returns `true` or `false`, depending on the state of the action binding. For example, the action binding may be enabled (`true`) or disabled (`false`) based on the currency (as determined, for example, when the user clicks the First, Next, Previous, Last navigation buttons).	n/a	yes	n/a	n/a	n/a	n/a	n/a

Table B-1 (Cont.) EL Properties of Oracle ADF Bindings

Runtime Property	Description	Iterator	Action	Attribute	Button	List	Table	Tree
`rangeSet`	Returns a list of map elements over the range of rows from the associated iterator binding. The elements in this list are wrapper objects over the indexed row in the range that restricts access to the attributes to which the binding is bound. The properties returned on the reference object are: • `index`: The range index of the row this reference is pointing to. • `key`: The key of the row this reference is pointing to. • `keyStr`: The string format of the key of the row this reference is pointing to. • `currencyString`: The current indexed row as a string. Returns "*" if the current entry belongs to the current row; otherwise, returns " ". This property is useful in JSP applications to display the current row. • `attributeValues`: The array of applicable attribute values from the row. You may also access an attribute value by name on a range set like `rangeSet.dname` if `dname` is a bound attribute in the range binding.	n/a	n/a	n/a	n/a	n/a	yes	yes
`rangeSize`	Returns the range size of the ADF iterator binding's row set. This allows you to determine the number of data objects to bind from the data source.	yes	n/a	n/a	n/a	n/a	yes	yes
`rangeStart`	Returns the absolute index in a collection of the first row in range. See the javadoc for `oracle.jbo.RowSetIterator.getRangeStart()`.	yes	n/a	n/a	n/a	n/a	yes	yes
`result`	Returns the result of a method that is bound and invoked by a method action binding.	n/a	yes	n/a	n/a	n/a	n/a	n/a

Table B-1 (Cont.) EL Properties of Oracle ADF Bindings

Runtime Property	Description	Iterator	Actio n	Attribute	Button	List	Table	Tree
`rootNodeBinding`	Returns the root node of a tree binding.	n/a	n/a	n/a	n/a	n/a	n/a	yes
`selectedValue`	Returns the value corresponding to the current selected index in the list or button binding.	n/a	n/a	n/a	yes	yes	n/a	n/a
`tooltip`	Returns the tooltip hint for the first attribute to which the binding is associated.	n/a	n/a	yes	yes	yes	n/a	n/a
`updateable`	Returns `true` if the first attribute to which the binding is associated is updateable. Otherwise, returns `false`.	n/a	n/a	yes	yes	yes	n/a	n/a

C

ADF Security Permission Grants

This appendix lists the security-aware components of Oracle ADF and the actions that their Permission implementation classes define.

Table C-1 shows the ADF components and their permission grants that you can define to create ADF security policies. You add grants to the policy store using the overview editor for ADF security policies. A permission grant specifies the fully qualified permission class name, the fully qualified resource name, the action that can be performed against the resource, and the application role target of the grant. When you enable ADF security to enforce authorization checking against the security policies of the policy store, the operations supported by ADF components will be inaccessible to users who do not possess sufficient access rights as defined by grants to their application role.

For complete details about defining ADF security policies in Fusion web applications, see Enabling ADF Security in a Fusion Web Application.

Table C-1 ADF Security Permission Grants

ADF Component	Grantable Action	Corresponding Implementation
ADF bounded task flow	`View`	The `View` action controls who can read and execute a bounded task flow. Pages that the user accesses within the process of executing a bounded task flow will not be individually security checked and will run under the permission of the task flow.
ADF bounded task flow	`Customize`	Reserved for future use. This action is not checked at runtime.
ADF bounded task flow	`Grant`	Reserved for future use. This action is not checked at runtime.
ADF bounded task flow	`Personalize`	Reserved for future use. This action is not checked at runtime.
ADF page definition	`View`	The `View` action controls who can view the page. Page-level security is checked for pages that have an associated page definition binding file *only* if the page is accessed in the process of an unbounded task flow. There is a one-to-one relationship between the **page definition file** and the web page it secures.
ADF Business Components entity objects	`read`	The `read` action controls who can view a row of the bound collection.
ADF Business Components entity objects	`update`	The `update` action controls who can update any attribute of the bound collection.
ADF Business Components entity objects	`delete`	The `delete` action controls who can delete a row from the bound collection. This action corresponds to the entity object `removeCurrentRow` operation.
ADF Business Components attributes of entity objects	`update`	The `update` action controls who can update a specific attribute of the bound collection.

D

Most Commonly Used ADF Business Components Methods

This appendix lists the most commonly used methods in the interfaces and classes of the ADF Business Components layer of Oracle ADF.
This appendix contains the following sections:

- Methods for Creating Your Own Layer of Framework Base Classes
- Methods Used in the Client Tier
- Methods Used in the Business Service Tier

Methods for Creating Your Own Layer of Framework Base Classes

You can create custom framework classes that extend ADF's base Business Components classes and that your own business components definitions use or extend. This is true for application modules as well.

Before you begin to develop application-specific business components, you can create a layer of classes that extend all of the **ADF Business Components** framework base implementation classes described in this appendix. An example of a customized framework base class for **application module** components might look like this:

```
package com.yourcompany.adfextensions;
import oracle.jbo.server.ApplicationModuleImpl;
public class CustomApplicationModuleImpl extends ApplicationModuleImpl {
  /*
   * We might not yet have any custom code to put here yet, but
   * the first time we need to add a generic feature that all of
   * our company's application modules need, we will be very happy
   * that we thought ahead to leave ourselves a convenient place
   * in our class hierarchy to add it so that all of the application
   * modules we have created will instantly benefit by that new feature,
   * behavior change, or even perhaps, bug workaround.
   */
}
```

A common set of customized framework base classes in a package name of your own choosing like `com.yourcompany.adfextensions`, each importing the `oracle.jbo.server.*` package, would consist of the following classes:

- `public class CustomEntityImpl extends EntityImpl`
- `public class CustomEntityDefImpl extends EntityDefImpl`
- `public class CustomViewObjectImpl extends ViewObjectImpl`
- `public class CustomViewRowImpl extends ViewRowImpl`
- `public class CustomApplicationModuleImpl extends ApplicationModuleImpl`

- `public class CustomDBTransactionImpl extends DBTransactionImpl2`
- `public class CustomDatabaseTransactionFactory extends DatabaseTransactionFactory`

For completeness, you may also want to create customized framework classes for the following classes as well:

- `public class CustomViewDefImpl extends ViewDefImpl`
- `public class CustomEntityCache extends EntityCache`
- `public class CustomApplicationModuleDefImpl extends ApplicationModuleDefImpl`

Overriding anything in these classes would be a fairly rare requirement.

Methods Used in the Client Tier

The client tier contains code that runs on the client and with which an ADF user interacts. There are several interface methods in the client tier for the key ADF Business Components.

All of the interfaces described in this section are designed so that you may expose method for use in the client tier. These interface methods part of the `oracle.jbo.*` package.

This section provides a summary of the most frequently called, written, and overridden methods for the key ADF Business Components interfaces.

> **Note:**
>
> The corresponding implementation classes for these `oracle.jbo.*` interfaces are intentionally designed to *not* be directly accessed by client code. Methods Used in the Business Service Tier shows that the implementation classes reside in the `oracle.jbo.server.*` package and generally have the suffix `Impl` in their name to help remind you not to use them in your client-layer code.

ApplicationModule Interface

An application module is a business service component that acts as a transactional container for other ADF components and coordinates *with* them to implement a number of Java EE design patterns important to business application developers. These design pattern implementations enable your client code to work easily with updatable collections of value objects, based on fast-lane reader SQL queries that retrieve only the data needed by the client, in the way the client wants to view it. Changes made to these value objects are automatically coordinated with your persistent business domain objects in the business service tier to enforce business rules consistently and save changes back to the database. Table D-1 describes the operations that you can perform on an application module using the `ApplicationModule` interface.

Note:

For the complete list of design patterns that ADF Business Components implements, see ADF Business Components Java EE Design Pattern Catalog.

Table D-1 ApplicationModule Interface

If you want to...	Call this ApplicationModule interface method
Access an existing **view object instance** using the assigned instance name (for example, `MyVOInstanceName`)	`findViewObject()`
Create a new view object instance from an existing definition	`createViewObject()`
Create a new view object instance from a SQL Statement	`createViewObjectFromQueryStmt()` **Notes:** This incurs runtime overhead to describe the "shape" of the dynamic query's `SELECT` list. Use this method only when you cannot know the `SELECT` list for the query at design time. Furthermore, if you are creating the dynamic query based on some kind of custom runtime repository, you can follow the steps to create (both read-only and updatable) dynamic view objects without the runtime-describe overhead, as described in Creating a View Object with Multiple Updatable Entities. If only the `WHERE` needs to be dynamic, create the view object at design time, then set the `WHERE` clause dynamically as needed using `ViewObject` APIs.
Access a nested **application module instance** by name	`findApplicationModule()`
Create a new nested application module instance from an existing definition	`createApplicationModule()`
Find a view object instance in a nested application module using a dot-notated name (for example, `MyNestedAMInstanceName.OneOfItsVONames`)	`findViewObject()` **Notes:** You can use this method to find an instance of a view object belonging to a nested application module in a single method call. This way you do not need to first call `findApplicationModule()` to find the nested application module, before calling `findViewObject()` on that nested application module.
Access the current transaction object	`getTransaction()`

In addition to generic application module access, JDeveloper can generate a custom *`YourApplicationModuleName`* interface containing service-level custom methods that you've chosen to expose to the client. You use the **Client Interface** page of the Edit

Application Module dialog to select the methods that you want to appear in your client interface.

Transaction Interface

The `Transaction` interface exposes methods allowing the client to manage pending changes in the current transaction. Table D-2 describes the operations that you can perform on the transaction using the `Transaction` interface.

Table D-2 Transaction Interface

If you want to...	Call this Transaction interface method
Commit pending changes	`commit()`
Roll back pending changes	`rollback()`
Execute a one-time database command or block of PL/SQL	`executeCommand()` **Notes:** Do not use this command with methods that require retrieving `OUT` parameters and that will be executed more than once, or that could benefit from using bind variables. Instead, expose a custom method on your application module.
Validate all pending invalid changes in the transaction	`validate()`
Change the default locking mode	`setLockingMode()` **Notes:** You can set the locking mode in your configuration by setting the property `jbo.locking.mode` to one of the four supported values: `none`, `optimistic`, `pessimistic`, `optupdate`. If you don't explicitly set the locking mode, it will default to `optimistic`. For Fusion web applications, use `optimistic` or `optupdate` modes.
Decide whether to use bundled exception reporting mode or not	`setBundledExceptionMode()` **Notes:** The **ADF Controller** layer support sets this parameter to `true` automatically for Fusion web applications.
Decide whether entity caches will be cleared upon a successful commit of the transaction	`setClearCacheOnCommit()` **Notes:** Default is `false`.
Decide whether entity caches will be cleared upon a rollback of the transaction	`setClearCacheOnRollback()` **Notes:** Default is `true`.
Clear the entity cache for a specific **entity object**	`clearEntityCache()`

ViewObject Interface

A view object is a component that encapsulates a database query and simplifies working with the row set of results it produces. You use view objects to project, filter, join, or sort business data using SQL from one or more tables to cast the data into exactly the format that the user should see on the page or panel. You can create "master-detail" hierarchies of any depth or complexity by connecting view objects together using view links. View objects can produce read-only query results, or when associated with one or more entity objects at design time, can be fully updatable. Updatable view objects can support insertion, modification, and deletion of rows in the result collection, with automatic delegation to the correct business domain objects.

Every view object contains a "default row set" for simplifying the 90 percent of use cases where you work with a single row set of results for the view object's query. A view object implements all the methods on the `RowSet` interface by delegating them to this default `RowSet`. That means you can invoke any `RowSet` methods on any view object as well.

Every view object implements the `StructureDef` interface to provide information about the number and types of attributes in a row of its row sets. So you can call `StructureDef` methods directly on any view object.

Table D-3 describes the operations that you can perform on a view object using the `ViewObject` interface

Table D-3 ViewObject Interface

If you want to...	Call this ViewObject interface method
Set an additional runtime `WHERE` clause on the row set	`setWhereClause()` **Notes:** This `WHERE` clause augments any `WHERE` clause specified at design time in the base view object. It does not replace it.
Set a dynamic `ORDER BY` clause	`setSortBy()`
Create a Query-by-Example criteria collection	`createViewCriteria()` **Notes:** You then create one or more `ViewCriteriaRow` objects using the `createViewCriteriaRow()` method on the `ViewCriteria` object you created. Then you `add()` these **view criteria** rows to the view criteria collection and apply the criteria using the `applyViewCriteria()` method.
Apply a Query-by-Example criteria collection	`applyViewCriteria()`
Set a query optimizer hint	`setQueryOptimizerHint()`
Access the attribute definitions for the key attributes in the view object	`getKeyAttributeDefs()`
Add a dynamic attribute to rows in this view object's row sets	`addDynamicAttribute()`
Clear all row sets produced by a view object	`clearCache()`

Table D-3 (Cont.) ViewObject Interface

If you want to...	Call this ViewObject interface method
Remove a view object instance and its resources	`remove()`
Set an upper limit on the number of rows that the view object will attempt to fetch from the database	`setMaxFetchSize()` **Notes:** Default is -1, which means to impose no limit on how many rows would be retrieved from the database if you iterated through them all. By default, as you iterate through them, they are fetched lazily.

In addition to generic `ViewObject` access, JDeveloper can generate you a custom *`YourViewObjectName`* interface containing view object-level custom methods that you've chosen to expose to the client. You use the **Client Interface** page of the Edit View Object dialog to select the methods that you want to appear in your client interface.

RowSet Interface

A row set is an object that contains a set of rows, typically produced by executing a view object's query.

Every `RowSet` aggregates a "default **row set iterator**" for simplifying the 90 percent of use cases where you need only a single iterator over the row set. A `RowSet` object implements all the methods on the `RowSetIterator` interface by delegating them to this default `RowSetIterator`. This means you can invoke any `RowSetIterator` method on any `RowSet` object (or view object, since it implements `RowSet`, as well for its default `RowSet`).

Table D-4 describes the operations that you can perform on a row set using the `RowSet` interface.

Table D-4 RowSet Interface

If you want to...	Call this RowSet interface method
Set a `WHERE` clause bind variable value	`setWhereClauseParams()` **Notes:** Bind variable ordinal positions are zero-based.
Avoid view object row caching if data is being read only once	`setForwardOnly()`

Table D-4 (Cont.) RowSet Interface

If you want to...	Call this RowSet interface method
Force a row set's query to be (re)executed (in the case of exclusive view object instances) or potentially executed (in the case of shared view object instances)	`executeQuery()` **Notes:** The behavior of this method differs depending on whether the view object belongs to a **shared application module** or not. When reexecuting the query for an exclusive view object (not an instance of a shared module), a new query collection is created. Before executing the query for a shared view object instance, a check is performed to determine whether the results already exist. Already cached results will be reused for the shared view object instance instead of reexecuting the query. If you want to ensure that the results for a shared view object instance are refreshed, you can invoke the `forceExecuteQueryOfSharedVO()` method. However, if at the time of invoking force execute a user is iterating over the collection of a shared view object instance, then the behavior is undefined and exceptions may result.
Estimate the number of rows in a view object's query result	`getEstimatedRowCount()`
Produce an XML document for rows in a view object row set	`writeXML()`
Process all rows from an incoming XML document	`readXML()`
Set whether a row set will automatically see new rows based on the same entity object created through other row sets	`setAssociationConsistent()`
Create a secondary iterator to use for programmatic iteration	`createRowSetIterator()` **Notes:** If you plan to find and use the secondary iterator by name later, then pass in a string name as the argument; otherwise, pass `null` for the name and make sure to close the iterator when done iterating by calling its `closeRowSetIterator()` method.

RowSetIterator Interface

A row set iterator is an iterator over the rows in a row set. By default it allows you to iterate both forward and backward through the rows. Table D-5 describes the operations that you can perform on a row set using the `RowSetIterator` interface.

Table D-5 RowSetIterator Interface

If you want to...	Call this RowSetIterator interface method
Get the first row of the iterator's row set	`first()`
Test whether there are more rows to iterate	`hasNext()`
Get the next row of an iterator's row set	`next()`
Find a row in this iterator's row set with a given key value	`findByKey()` **Notes:** It's important that the `Key` object that you pass to `findByKey` be created using the *exact* same data types as the attributes that comprise the key of the rows in the view object you're working with.
Create a new row to populate for insertion	`createRow()` **Notes:** The new row will already have default values set for attributes which either have a static default value supplied at the entity object or view object level, or if the values have been populated in an overridden `create()` method of the underlying entity object(s).
Create a view row with an initial set of foreign key and/or discriminator attribute values	`createAndInitRow()` **Notes:** You use this method when working with view objects that can return one of a "family" of entity object subtypes. By passing in the correct discriminator attribute value in the call to create the row, the framework can create you the correct matching entity object subtype underneath.
Insert a new row into the iterator's row set	`insertRow()` **Notes:** It's a good habit to always immediately insert a newly created row into the rowset. That way you will avoid a common gotcha of creating the row but forgetting to insert it into the rowset.
Get the last row of the iterator's row set	`last()`
Get the previous row of the iterator's row set	`previous()`
Reset the current row pointer to the slot before the first row	`reset()`
Close an iterator when done iterating	`closeRowSetIterator()`
Set a given row to be the current row	`setCurrentRow()`
Remove the current row	`removeCurrentRow()`
Remove the current row to later insert it at a different location in the same iterator	`removeCurrentRowAndRetain()`
Remove the current row from the current collection but do not remove it from the transaction.	`removeCurrentRowFromCollection()`

Table D-5 (Cont.) RowSetIterator Interface

If you want to...	Call this RowSetIterator interface method
Set/change the number of rows in the range (a "page" of rows the user can see)	`setRangeSize()`
Scroll to view the *n*th page of rows (1-based)	`scrollToRangePage()`
Scroll to view the range of rows starting with row number *n*	`scrollRangeTo()`
Set row number *n* in the range to be the current row	`setCurrentRowAtRangeIndex()`
Get all rows in the range as a row array	`getAllRowsInRange()`

Row Interface

A row is generic value object. It contains attributes appropriate in name and Java type for the view object that it is related to. Table D-6 describes the operations that you can perform on a view object row using the `Row` interface.

Table D-6 Row Interface

If you want to...	Call this Row interface method
Get the value of an attribute by name	`getAttribute()`
Set the value of an attribute by name	`setAttribute()`
Produce an XML document for a single row	`writeXML()`
Eagerly validate a row	`validate()`
Read row attribute values from XML	`readXML()`
Remove the row	`remove()`
Flag a newly created row as temporary (until updated again)	`setNewRowState(Row.STATUS_INITIALIZED)`
Retrieve the attribute structure definition information for a row	`getStructureDef()`
Get the `Key` object for a row	`getKey()`

In addition to generic `Row` access, JDeveloper can generate a custom *`YourViewObjectName`*`Row` interface containing your type-safe attribute getter and setter methods, as well as any desired row-level custom methods that you've chosen to expose to the client. You use the **Client Row Interface** page of the Edit View Object dialog to select the methods that you want to appear in your client interface.

StructureDef Interface

The `StructureDef` interface provides access to runtime metadata about the structure of a `Row` object.

In addition, for convenience every view object implements the `StructureDef` interface as well, providing access to metadata about the attributes in the resulting view rows that its query will produce.

Table D-7 describes the operations that you can perform on a view object row using the `StructureDef` interface.

Table D-7 StructureDef Interface

If you want to...	Call this StructureDef interface method
Access attribute definitions for all attributes in the view object row	`getAttributeDefs()`
Find an attribute definition by name	`findAttributeDef()`
Get attribute definition by index	`getAttributeDef()`
Get number of attributes in a row	`getAttributeCount()`

AttributeDef Interface

The `AttributeDef` interface provides attribute definition information for any attribute of a view object row or entity object instance like attribute name, Java type, and SQL type. It also provides access to custom attribute-specific metadata properties that can be inspected by generic code you write, as well as UI hints that can assist in rendering an appropriate user interface display for the attribute and its value. Table D-8 describes the operations that you can perform on an attribute using the `AttributeDef` interface.

Table D-8 AttributeDef Interface

If you want to...	Call this AttributeDef interface method
Get the Java type of the attribute	`getJavaType()`
Get the SQL type of the attribute	`getSQLType()` **Notes:** The `int` value corresponds to constants in the JDBC class `java.sql.Types`.
Determine the kind of attribute	`getAttributeKind()` **Notes:** A simple attribute is one that returns one of the constants `ATTR_PERSISTENT`, `ATTR_SQL_DERIVED`, `ATTR_TRANSIENT`, `ATTR_DYNAMIC`, `ATTR_ENTITY_DERIVED`. If the attribute is a 1-to-1 or many-to-1 association/viewlink accessor, it returns `ATTR_ASSOCIATED_ROW`. If the attribute is a 1-to-many or many-to-many association/viewlink accessor, it returns `ATTR_ASSOCIATED_ROWITERATOR`
Get the Java type of elements contained in an `Array`-valued attribute	`getElemJavaType()`
Get the SQL type of elements contained in an `Array`-valued attribute	`getElemSQLType()`
Get the name of the attribute	`getName()`
Get the index position of the attribute	`getIndex()`

Table D-8 (Cont.) AttributeDef Interface

If you want to...	Call this AttributeDef interface method
Get the precision of a numeric attribute or the maximum length of a string attribute	`getPrecision()`
Get the scale of a numeric attribute	`getScale()`
Get the underlying column name corresponding to the attribute	`getColumnNameForQuery()`
Get attribute-specific custom property values	`getProperty()`, `getProperties()`
Get the UI `AttributeHints` object for the attribute	`getUIHelper()`
Test whether the attribute is mandatory	`isMandatory()`
Test whether the attribute is queriable	`isQueriable()`
Test whether the attribute is part of the primary key for the row	`isPrimaryKey()`

AttributeHints Interface

The `AttributeHints` interface exposes UI hint information that you can use to render an appropriate user interface display for the attribute and its value. Table D-9 describes the operations that you can perform on an attribute using the `AttributeHints` interface.

Table D-9 AttributeHints Interface

If you want to...	Call this AttributeHints interface method
Get the UI label for the attribute	`getLabel()`
Get the tooltip for the attribute	`getTooltip()`
Get the formatted value of the attribute, using any format mask supplied	`getFormattedAttribute()`
Get the display hint for the attribute	`getDisplayHint()` **Notes:** The display hint will have a string value of either `Display` or `Hide`.
Get the preferred control type for the attribute	`getControlType()`
Parse a formatted string value using any format mask supplied for the attribute	`parseFormattedAttribute()`

Methods Used in the Business Service Tier

The Business Service tier contains the business logic. There are several methods in the business service tier for the key ADF Business Components.

The implementation classes corresponding to the `oracle.jbo.*` interfaces, as described in Methods Used in the Client Tier, are intentionally designed to *not* be directly accessed by client code. They reside in a different package named oracle.jbo.server.* and have the Impl suffix in their name to help remind you not to use them in your client-layer code.

In your business service tier implementation code, you can use any of the same methods that are available to clients, but in addition you can also:

- Safely cast any oracle.jbo.* interface to its oracle.jbo.server.* package implementation class and use any methods on that Impl class as well.
- Override any of the public or protected methods for the base framework implementation classes and write custom code in your component subclass before or after calling super.*methodName*() to augment or change the default functionality.

This section provides a summary of the most frequently called, written, and overridden methods for the key ADF Business Components classes.

Controlling Custom Java Files for Your Components

Before examining the specifics of individual classes, it's important to understand how you can control which custom Java files each of your components will use. When you don't need a customized subclass for a given component, you can just let the base framework class handle the implementation at runtime.

Each business component you create comprises a single XML component descriptor, and zero or more related custom Java implementation files. Each component that supports Java customization has a Java page in its component overview editor in the JDeveloper IDE. By selecting or deselecting the different Java classes, you control which ones will be created for your component. If none of the classes is specified, then your component will be an XML-only component, which simply uses the base framework class as its Java implementation. Otherwise, tick the checkbox of the related Java classes for the current component that you need to customize. JDeveloper will create a custom *subclass* of the framework base class in which you can add your code.

Note:

You can set up global IDE preferences for the Java classes to be generated by default for each ADF business component type by choosing **Tools** > **Preferences** > **Business Components** and ticking the checkboxes to indicate what you want your defaults to be.

A best practice is to *always* generate entity object and view row classes, even if you don't require any custom code in them other than the automatically generated getter and setter methods. These getter and setter methods offer you compile-time type checking that prevents errors surfacing at runtime in response to an attribute having been set to an incorrect kind of value.

ApplicationModuleImpl Class

The ApplicationModuleImpl class is the base class for application module components. Since the application module is the ADF component used to implement a business service, think of the application module class as the place where you can write your service-level application logic. The application module coordinates with view object instances to support updatable collections of value objects that are automatically "wired" to business domain objects. The business domain objects are implemented as ADF entity objects.

Methods You Typically Call on ApplicationModuleImpl

Table D-10 describes the operations that you can perform on an application module using the `ApplicationModuleImpl` class.

Table D-10 Methods You Typically Call on ApplicationModuleImpl

If you want to...	Call this method of the ApplicationModuleImpl class
Perform any of the common application module operations from inside your class, which can also be done from the client	For a list of these methods, see ApplicationModule Interface.
Access a view object instance that you added to the application module's data model at design time	`get`*`ViewObjectInstanceName`*`()` **Notes:** JDeveloper generates this type-safe view object instance getter method for you to reflect each view object instance in the application module's design time data model.
Access the current `DBTransaction` object	`getDBTransaction()`
Access a nested application module instance that you added to the application module at design time	`get`*`AppModuleInstanceName`*`()` **Notes:** JDeveloper generates this type-safe application module instance getter method for you to reflect each nested application module instance added to the current application module at design time.

Methods You Typically Write in Your Custom ApplicationModuleImpl Subclass

Table D-11 describes the operations that you can perform on an application module using your custom `ApplicationModuleImpl` class.

Table D-11 Methods You Typically Write in Your Custom ApplicationModuleImpl Subclass

If you want to...	Write a method like this in your custom ApplicationModuleImpl class
Invoke a database stored procedure	*`someCustomMethod`*`()` **Notes:** Use the appropriate method on the `DBTransaction` interface to create a JDBC `PreparedStatement`. If the stored procedure has `OUT` parameters, then create a `CallableStatement` instead. For sample code that demonstrates encapsulating a call to a PL/SQL stored procedure inside your application module, see Invoking Stored Procedures and Functions.
Expose custom business service methods on your application module	*`someCustomMethod()`* **Notes:** Select the method name on the **Client Interface** page of the Edit Application Module dialog to expose it for client access if required.

JDeveloper can generate a custom *`YourApplicationModuleName`* interface containing service-level custom methods that you've chosen to expose to the client. You can use the **Client Interface** page of the Edit Application Module dialog to select the methods that you want to appear in your client interface.

Methods You Typically Override in Your Custom ApplicationModuleImpl Subclass

Table D-12 describes the operations that you can override on an application module using your custom `ApplicationModuleImpl` class.

Table D-12 Methods You Typically Override in Your Custom ApplicationModuleImpl Subclass

If you want to...	Override this method in your custom ApplicationModuleImpl class
Perform custom setup code the first time an application module is created and each subsequent time it gets used by a different client session.	`prepareSession()` **Notes:** This is the method you'd use to set up per-client context info for the current user in order to use Oracle's Virtual Private Database (VPD) features. It can also be used to set other kinds of PL/SQL package global variables, whose values might be client-specific, on which other stored procedures might rely. This method is also useful to perform setup code that is specific to a given view object *instance* in the application module. If instead of the view object setup code being instance-specific, you want it to be initialized for every instance ever created of that view object component, then put the setup logic in an overridden `create()` method in your `ViewObjectImpl` subclass instead.
Perform custom setup code after the application module's transaction is associated with a database connection from the connection pool.	`afterConnect()` **Notes:** Can be a useful place to write a line of code that uses `getDBTransaction().executeCommand()` to perform an `ALTER SESSION SET SQL TRACE TRUE` to enable database SQL trace logging for the current application connection. These logs can then be processed with the TKPROF utility to study the SQL statements being performed and the query optimizer plans that are getting used. For details about working with the `TKPROF` utility, see section "Understanding SQL Trace and TKPROF" in the "Performing Application Tracing" chapter of the *Oracle Database SQL Tuning Guide*.

Table D-12 (Cont.) Methods You Typically Override in Your Custom ApplicationModuleImpl Subclass

If you want to...	Override this method in your custom ApplicationModuleImpl class
Perform custom setup code before the application module's transaction releases its database connection back to the database connection pool.	`beforeDisconnect()` **Notes:** If you have set `jbo.doconnectionpooling` to `true`, then the connection is released to the database connection pool each time the application module is returned to the **application module pool**. In order to clear custom database state when the Application Module returns to the Application Module pool, applications should override the `beforeDisconnect()`method. The method `resetState()` is not recommended for this purpose since Application Modules may participate in transaction sharing. Cleaning up database state in `resetState()` from one Application Module while others are still referencing the same DB Transaction can lead to issues.
Write custom application module state to the state management XML snapshot.	`passivateState()`
Read and restore custom application module state from the state management XML snapshot.	`activateState()`

DBTransactionImpl2 Class

The `DBTransactionImpl2` class — which extends the base `DBTransactionImpl` class, and is constructed by the `DatabaseTransactionFactory` class — is the base class that implements the `DBTransaction` interface, representing the unit of pending work in the current transaction.

Methods You Typically Call on DBTransaction

Table D-13 describes the operations that you can perform on a transaction using the `DBTransaction` class.

Table D-13 Methods You Typically Call on DBTransaction

If you want to...	Call this method on the DBTransaction class
Commit the transaction	`commit()`
Roll back the transaction	`rollback()`
Eagerly validate any pending invalid changes in the transaction	`validate()`

Table D-13 (Cont.) Methods You Typically Call on DBTransaction

If you want to...	Call this method on the DBTransaction class
Create a JDBC `PreparedStatement` using the transaction's `Connection` object	`createPreparedStatement()`
Create a JDBC `CallableStatement` using the transaction's `Connection` object	`createCallableStatement()`
Create a JDBC `Statement` using the transaction's `Connection` object	`createStatement()`
Add a warning to the transaction's warning list	`addWarning()`

Methods You Typically Override in Your Custom DBTransactionImpl2 Subclass

Table D-14 describes the operations that you can perform on a transaction using your custom `DBTransactionImpl2` subclass.

Table D-14 Methods You Typically Override in Your Custom DBTransactionImpl2 Subclass

If you want to...	Override this method in your custom DBTransactionImpl2 class
Perform custom code before or after the transaction commit operation	`commit()`
Perform custom code before or after the transaction rollback operation	`rollback()`

In order for your custom `DBTransactionImpl2` subclass to be used at runtime, there are you must follow these steps:

1. Create a custom subclass of `DatabaseTransactionFactory` that overrides the create method to return an instance of your custom `DBTransactionImpl2` subclass like this:

```
package com.yourcompany.adfextensions;
import oracle.jbo.server.DBTransactionImpl2;
import oracle.jbo.server.DatabaseTransactionFactory;
import com.yourcompany.adfextensions.CustomDBTransactionImpl;
public class CustomDatabaseTransactionFactory
       extends DatabaseTransactionFactory {
  /**
   * Return an instance of our custom CustomDBTransactionImpl class
   * instead of the default implementation.
   *
   * @return An instance of our custom DBTransactionImpl2 implementation.
   */
  public DBTransactionImpl2 create() {
    return new CustomDBTransactionImpl();
  }
}
```

2. Tell the framework to use your custom transaction factory class by setting the value of the `TransactionFactory` configuration property to the fully qualified class

name of your custom transaction factory. As with other configuration properties, if not supplied in the configuration XML file, it can be provided alternatively as a Java system parameter of the same name.

EntityImpl Class

The `EntityImpl` class is the base class for entity objects, which encapsulate the data, validation rules, and business behavior for your business domain objects.

Methods You Typically Call on EntityImpl

Table D-15 describes the operations that you can perform on an entity object using the `EntityImpl` class.

Table D-15 Methods You Typically Call on EntityImpl

If you want to...	Call this method in the EntityImpl class
Get the value of an attribute	`get`*`AttributeName`*`()` **Notes:** This code-generated getter method calls `getAttributeInternal()`, but provides compile-time type checking.
Set the value of an attribute	`set`*`AttributeName`*`()` **Notes:** This code-generated setter method calls `setAttributeInternal()`, but provides compile-time type checking.
Get the value of an attribute by name	`getAttributeInternal()`
Set the value of an attribute by name	`setAttributeInternal()`
Eagerly perform entity object validation	`validate()`
Refresh the entity from the database	`refresh()`
Populate the value of an attribute without *marking* it as being changed, but sending *notification* of its being changed so that the UI refreshes the value on the screen/page	`populateAttributeAsChanged()`
Access the `Definition` object for an entity	`getDefinitionObject()`
Get the `Key` object for an entity	`getKey()`
Determine the state of the entity instance, irrespective of whether it has already been posted (but not yet committed) in the current transaction	`getEntityState()` **Notes:** This method will return one of the constants `STATUS_UNMODIFIED`, `STATUS_INITIALIZED`, `STATUS_NEW`, `STATUS_MODIFIED`, `STATUS_DELETED`, or `STATUS_DEAD`, indicating the status of the entity instance in the current transaction.

Table D-15 (Cont.) Methods You Typically Call on EntityImpl

If you want to...	Call this method in the EntityImpl class
Determine the state of the entity instance	`getPostState()` **Notes:** This method is typically relevant only if you are programmatically using the `postChanges()` method to post but not yet commit, entity changes to the database and need to detect the state of an entity with regard to its posting state.
Get the value originally read from the database for a given attribute	`getPostedAttribute()`
Get the metadata for a view object or entity object	`getStructureDef()` **Notes:** Downcasting the return value of `EntityImpl.getStructureDef()` method invocation to `EntityCache` is not supported. To obtain the instance of `EntityCache` invoke an existing `getEntityCache()`method on the `EntityImpl` object.
Eagerly lock the database row for an entity instance	`lock()`
Copy the values from `masterRow` that are relevant for the entity object to a newly created row.	`masterRow.findOrCreateAssociationAccessorRS(accessorName)` **Note:** This API returns a `RowSet` and follow the familiar pattern of `createRow` and `insertRow`.

Methods You Typically Write in Your Custom EntityImpl Subclass

Table D-16 describes the operations that you can perform on an entity object using your custom `EntityImpl` subclass.

Table D-16 Methods You Typically Write in Your Custom EntityImpl Subclass

If you want to...	Write a method like this in your custom EntityImpl subclass
Perform attribute-specific validation	`public boolean validate`*`Something`*`(`*`AttrType`*`value)` **Notes:** Register the attribute validator method by adding a MethodValidator rule on the correct attribute in the **Validation** page of the Edit Entity Object dialog.
Perform entity-level validation	`public boolean validate`*`Something`*`()` **Notes:** Register the entity-level validator method by adding a MethodValidator rule on the entity in the **Validation** panel of the Edit Entity Object dialog.
Calculate the value of a transient attribute	Add your calculation code to the generated `get`*`AttributeName`*`()` method.

Methods You Typically Override in Your Custom EntityImpl Subclass

Table D-17 describes the operations that you can override on an entity object using your custom `EntityImpl` subclass.

Table D-17 Methods You Typically Override in Your Custom EntityImpl Subclass

If you want to...	Override this method in your EntityImpl subclass
Set calculated default attribute values, including programmatically populating the primary key attribute value of a new entity instance	`create()` **Notes:** After calling `super.create()`, call the appropriate `set`*`AttrName`*`()` method(s) to set the default values for the attributes.
Modify attribute values before changes are posted to the database	`prepareForDML()`
Augment or change the standard `INSERT`, `UPDATE`, or `DELETE` DML operation that the framework will perform on your entity object's behalf to the database	`doDML()` **Notes:** This method checks the value of the operation flag to the constants `DML_INSERT`, `DML_UPDATE`, or `DML_DELETE` to test what DML operation is being performed.
Perform complex, SQL-based validation after all entity instances have been posted to the database but before those changes are committed	`beforeCommit()` **Notes:** This method is invoked for each entity row in the transaction.

Table D-17   (Cont.) Methods You Typically Override in Your Custom EntityImpl Subclass

If you want to...	Override this method in your EntityImpl subclass
Perform custom processing after all entity instances have been committed to the database	`afterCommit()` **Notes:** This method is invoked for each entity row in the transaction list of pending changes.
Insure that a related, newly created, parent entity gets posted to the database *before* the current child entity on which it depends	`postChanges()` **Notes:** If the parent entity is related to this child entity via a composition association, then the framework already handles posting the changes automatically. If they are only associated (but not composed), then you need to override `postChanges()` to force a newly created parent entity to post before the current, dependent child entity. For an example of the code you typically write in your overridden `postChanges()` method to accomplish this, see Overriding postChanges() to Control Post Order.
Programmatically define default attribute values or to perform custom initialization before a modified row is refreshed back to its initialization state or after the row is first created	`initDefaultExpressionAttributes()` **Notes:** This method is invoked just before the end of the initialization cycle, which makes it suitable for executing custom code.

Note:

It is possible to write attribute-level validation code directly inside the appropriate `set`*`AttributeName`* method of your `EntityImpl` class; however, adopting the `MethodValidator` approach suggested in Table D-16 conveniently places all the validations in effect on the **Validation Rules** page of the overview editor for the attributes of the entity object.

> **WARNING:**
>
> It is also possible to override the `validateEntity()` method to write entity-level validation code; however, if you want to maintain the benefits of the ADF bundled exception mode — where the framework collects and reports a *maximal* set of validation errors back to the client user interface — use the `MethodValidator` approach suggested in Table D-16. This allows the framework to automatically collect all of your exceptions that your validation methods throw without your having to understand the bundled exception implementation mechanism. Overriding the `validateEntity()` method directly shifts the responsibility onto your *own* code to correctly catch and bundle the exceptions that **Oracle ADF** would have caught by default, which is nontrivial and a chore to remember and hand-code each time.

EntityDefImpl Class

The `EntityDefImpl` class is a singleton, shared metadata object for all entity objects of a given type in a single Java VM. For instance, if the Java VM contains two instances of the `EntityDefImpl` class, one with the customers XML data and one with the orders, there would only ever be one instance of each. This class defines the structure of the entity instances and provides methods to create new entity instances and find existing instances by their primary key.

Methods You Typically Call on EntityDefImpl

Table D-18 describes the operations that you can perform on an entity object using the `EntityDefImpl` class.

Table D-18 Methods You Typically Call on EntityDefImpl

If you want to...	Call this method in the EntityDefImpl class
Find an entity object of a given type by its primary key	`findByPrimaryKey()` **Notes:** For a tip about getting `findByPrimaryKey()` to find entity instances of subtype entities as well, see Subtype Entity Objects and the findByPrimaryKey() Method.
Access the current `DBTransaction` object	`getDBTransaction()`
Find any `EntityDefImpl` object by its fully qualified name	`findDefObject()` (static method)
Retrieve the value of an entity object's custom property	`getProperty()`, `getProperties()`
Set the value of an entity object's custom property	`setProperty()`

Table D-18 (Cont.) Methods You Typically Call on EntityDefImpl

If you want to...	Call this method in the EntityDefImpl class
Create a new instance of an entity object	`createInstance2()` **Notes:** Alternatively, you can expose custom `create`*`XXX`*`()` methods with your own expected signatures in that same custom `EntityDefImpl` subclass. See Methods You Typically Write in Your Custom EntityDefImpl Class for details.
Iterate over the entity instances in the cache of this entity type	`getAllEntityInstancesIterator()`
Access an array list of entity definition objects for entities that extend the current one.	`getExtendedDefObjects()`

Methods You Typically Write in Your Custom EntityDefImpl Class

Table D-19 describes the operations that you can perform on an entity object using your custom `EntityDefImpl` class.

Table D-19 Methods You Typically Write on EntityDefImpl

If you want to...	Write a method like this in your custom EntityDefImpl class
Allow other classes to create an entity instance with an initial type-safe set of attribute values or setup information	`create`*`XXX`*`(`*`Type1arg1`*`,...,` *`TypeNargN`*`)` **Notes:** Internally, using this method would create and populate an instance of a `NameValuePairs` object (which implements `AttributeList`) and call the protected method `createInstance()`, passing that `NameValuePairs` object. Make sure that the method is `public` if other classes need to be able to call it.

Methods You Typically Override in Your Custom EntityDefImpl

Table D-20 describes the operations that you can perform on an entity object using the `EntityDefImpl` class.

Table D-20 Methods You Typically Override on EntityDefImpl

If you want to...	Override this method in your custom EntityDefImpl class
Perform custom metadata initialization when this singleton metaobject is loaded	`createDef()`

Table D-20 (Cont.) Methods You Typically Override on EntityDefImpl

If you want to...	Override this method in your custom EntityDefImpl class
Avoid using the `RETURNING INTO` clause to support refresh-on-insert or refresh-on-update attributes	`isUseReturningClause()` **Notes:** Set this method to return `false` to disable the use of `RETURNING INTO`, necessary sometimes when your entity object is based on a view with `INSTEAD OF` triggers that don't support `RETURNING INTO` at the database level.
Control whether the `UPDATE` statements issued for this entity update only changed columns or for all columns	`isUpdateChangedColumns()` **Notes:** Defaults to `true`.
Find any `EntityDefImpl` object by its fully qualified name	`findDefObject()` **Notes:** Static method.
Set the value of an entity object's custom property	`setProperty()`
Allow other classes to create a new instance of an entity object without doing so implicitly via a view object	`createInstance()` **Notes:** If you don't write a custom create method as noted in Methods You Typically Write in Your Custom EntityDefImpl Class, you'll need to override this method and widen the visibility from `protected` to `public` to allow other classes to construct an entity instance.

ViewObjectImpl Class

The `ViewObjectImpl` class is the base class for view objects.

Methods You Typically Call on ViewObjectImpl

Table D-21 describes the operations that you can perform on a view object using the `ViewObjectImpl` class.

Table D-21 Methods You Typically Call on ViewObjectImpl

If you want to...	Call this method in the ViewObjectImpl class
Perform any of the common view object, row set, or row set iterator operations from inside your class, which can also be done from the client	For more information about operations at the view object, row set, or row set iterator level , see ViewObject Interface, RowSet Interface, and RowSetIterator Interface.
Search a view instance's in-memory row set without changing the default row set	`findByViewCriteria()`

Table D-21 (Cont.) Methods You Typically Call on ViewObjectImpl

If you want to...	Call this method in the ViewObjectImpl class
Fetch *all* the rows from the database and then determine the total row count in the view object row set	`getRowCount()` **Notes:** Use this method carefully: Performance may be affected when a large number of rows is involved.
Determine the total number of rows in view object row set after *each* invocation by the client	`getQueryHitCount()`
Determine the total number of rows in the view object row set	`getEstimatedRowCount()` **Notes:** This is the preferred way to find row count since it requires only one database query (after the count is obtained, it will be maintained as rows are added and removed).
Execute a count query but with a cap argument to manage execution in the case where a large number of rows may affect performance	`getCappedQueryHitCount()`
Define a named bind parameter	`defineNamedWhereClauseParam()`
Remove a named bind parameter	`removeNamedWhereClauseParam()`
Set bind variable values on the default row set by name	`setNamedWhereClauseParam()` **Notes:** Only works when you have formally defined named bind variables on your view object.
Set bind variable values on the default row set	`setWhereClauseParams()` **Notes:** Use this method for view objects with binding style of "Oracle Positional" or "JDBC Positional" when you have not formally defined named bind variables.
Retrieve a subset of rows in a view object's row set based on evaluating an in-memory filter expression	`getFilteredRows()`
Retrieve a subset of rows in the current range of a view object's row set based on evaluating an in-memory filter expression	`getFilteredRowsInRange()`
Override the runtime display of criteria items that appear inside an ADF query component. Subclasses may override to return custom `AttributeHints` implementation for the given criteria item. Returns null by default.	`getCriteriaItemAttributeHints()`

Table D-21 (Cont.) Methods You Typically Call on ViewObjectImpl

If you want to...	Call this method in the ViewObjectImpl class
Set the number of rows that will be fetched from the database per roundtrip for this view object	`setFetchSize()` **Notes:** The default fetch size is a single row at a time. This is definitely not optimal if your view object intends to retrieve many rows, so you should either set the fetch size higher at design time on the **Tuning** page of the Edit View Object dialog, or set it at runtime using this method.
Force a row set's query to be (re)executed specifically on a lookup view object instance in a shared application module	`forceExecuteQueryOfSharedVO()` **Notes:** Reexecuting the query forces a new query collection and will prevent the application module cache from being used. You should only use this method when you are sure that you are accessing the shared application module during setup and not during runtime. This method when used during normal runtime may have unintended side-effects that disrupt the navigation of users accessing the collection concurrently. If you want to refresh the collection from the cache without creating a new query collection, call `executeQuery()` instead.

Methods You Typically Write in Your Custom ViewObjectImpl Subclass

Table D-22 describes the operations that you can perform on a view object using your custom `ViewObjectImpl` subclass.

Table D-22 Methods You Typically Write in Your Custom ViewObjectImpl Subclass

If you want to...	Write a method like this in your custom ViewObjectImpl subclass
Provide clients with type-safe methods to set bind variable values without exposing positional details of the bind variables themselves	`*someMethodName*(*Type1arg1*,...,*TypeNargN*)` **Notes:** Internally, this method would call the `setWhereClauseParams()` method to set the correct bind variables with the values provided in the type-safe method arguments.

JDeveloper can generate a custom *`YourViewObjectName`* interface containing view object custom methods that you've chosen to expose to the client. You can use the **Client Interface** page of the Edit View Object to select the methods that you want to appear in your client interface.

Methods You Typically Override in Your Custom ViewObjectImpl Subclass

Table D-23 describes the operations that you can perform on a view object using your custom `ViewObjectImpl` subclass.

Table D-23 Methods You Typically Override in Your Custom ViewObjectImpl Subclass

If you want to...	Override this method in your custom ViewObjectImpl subclass
Initialize custom view object class members (not row attributes) when the view object instance is created for the first time	`create()` **Notes:** This method is useful to perform set up logic that is applicable to every instance of a view object that will ever get created, in the context of any application module. If instead of *generic* view object setup logic, you need to perform logic specific to a given view object *instance* in an application module, then override the `prepareSession()` method of your application module's `ApplicationModuleImpl` subclass and perform the logic there after calling `findViewObject()` to find the view object instance whose properties you want to set.
Write custom view object instance state to the state management XML snapshot	`passivateState()`
Read and restore custom view object instance state from the state management XML snapshot	`activateState()`

Table D-23 (Cont.) Methods You Typically Override in Your Custom ViewObjectImpl Subclass

If you want to...	Override this method in your custom ViewObjectImpl subclass
Customize the execution of the view object query to utilize an alternative data source	`executeQueryForCollection()` **Notes:** By default view objects read their data from the database and automate the task of working with the JDBC layer to process the database result sets. However, by overriding appropriate methods in its custom Java class, you can create a view object that programmatically retrieves data from alterative data sources, as described in Using Programmatic View Objects for Alternative Data Sources. It is not correct to override `executeQueryForCollection()` to make changes to `WHERE` clauses and parameters. By the time the method is called, ADF Business Components assumes that the parameters are already calculated and that row filters are constructed. If you change the query before it gets executed each time, ADF Business Components could never reuse an existing cached statement handle. Instead, when you want to specify bind parameters before the query is executed, you can override `prepareRowSetForQuery()` on `ViewObjectImpl` since it provides access to the row set that is being executed.
Customize the programmatic view object to utilize an alternative data source and determine whether the query collection has more rows to fetch from the query execution	`hasNextForCollection()`
Customize the programmatic view object to utilize an alternative data source and log bind parameter names and values used in the query	`bindParametersForCollection()`
Customize the programmatic view object to utilize an alternative data source and populate each row of the retrieved data	`createRowFromResultSet()`
Customize the programmatic view object to utilize an alternative data source and return a count of the number of rows that will be retrieved	`getQueryHitCount()`
Customize the programmatic view object to utilize an alternative data source and to determine whether the query would return more rows than the capped value specified by a global row fetch limit.	`getCappedQueryHitCount()`

Table D-23 (Cont.) Methods You Typically Override in Your Custom ViewObjectImpl Subclass

If you want to...	Override this method in your custom ViewObjectImpl subclass
Customize the programmatic view object to access the row set that is being executed and change the view object WHERE clause and specify bind parameter values before the query is executed. This is the recommended hook point for overriding query and setting bind parameters. This method is invoked by the framework every time a row set is about to be executed.	`prepareRowSetForQuery()`
Customize the programmatic view object to utilize an alternative data source and release any resources that may be associated with a row set that is being closed	`releaseUserDataForCollection()`
Change or augment the way that the `ViewCriteria` collection of `ViewCriteriaRows` is converted into a Query-by-Example `WHERE` clause	`getViewCriteriaClause(boolean)`
Customize the programmatic view object to implement range paging.	`buildRangePagingQuery()`
Customize the programmatic view object to implement range paging and pass in parameters to control the range start and size for the current range of rows to fetch from a data source.	`bindRangePagingParams()`

ViewRowImpl Class

The `ViewRowImpl` class is the base class for view row objects.

Methods You Typically Call on ViewRowImpl

Table D-24 describes the operations that you can perform on a view object row using your custom `ViewRowImpl` class.

Table D-24 Methods You Typically Call on ViewRowImpl

If you want to...	Call this method in your custom ViewRowImpl class
Perform any of the common view row operations from inside your class, which can also be done from the client	For more information about the row-level operations, see Row Interface.
Get the value of an attribute	get*AttrName*()
Set the value of an attribute	set*AttrName*()

Table D-24 (Cont.) Methods You Typically Call on ViewRowImpl

If you want to...	Call this method in your custom ViewRowImpl class
Access the underlying entity instance to which this view row is delegating attribute storage	`getEntityUsageAliasName()` **Notes:** You can change the name of the entity usage alias name on the **Entity Objects** page of the Edit View Object dialog.
Copy the values from `masterRow` that are relevant for the view link to a newly created row.	`masterRow.findOrCreateViewLinkAccessorRS(accessorName)` **Note:** This API returns a `RowSet` and follow the familiar pattern of `createRow` and `insertRow`.

Methods You Typically Write in Your Custom ViewRowImpl Class

Table D-25 describes the operations that you can perform on a view object row using your custom `ViewRowImpl` class.

Table D-25 Methods You Typically Write on ViewRowImpl

If you want to...	Write a method like this in your custom ViewRowImpl class
Calculate the value of a view object-level transient attribute	`getAttrName()` **Notes:** JDeveloper generates the skeleton of the method for you, but you need to write the custom calculation logic inside the method body.
Perform custom processing of the setting of a view row attribute	`setAttrName()` **Notes:** JDeveloper generates the skeleton of the method for you, but you need to write the custom logic inside the method body if required.
Determine the updateability of an attribute in a conditional way	`isAttributeUpdateable()`

Table D-25   (Cont.) Methods You Typically Write on ViewRowImpl

If you want to...	Write a method like this in your custom ViewRowImpl class
Expose logical operations on the current row, optionally callable by clients	`doSomething()` **Notes:** Often these view-row-level custom methods simply turn around and delegate to a method call on the underlying entity object related to the current row.

JDeveloper can generate a custom *`YourViewObjectName`*`Row` interface containing view row custom methods that you've chosen to expose to the client. You can use the **Client Row Interface** page of the Edit View Object dialog to select the methods that you want to appear in your client interface.

Methods You Typically Override in Your Custom ViewRowImpl Subclass

Table D-26 describes the operations that you can perform on a view object row using your custom `ViewRowImpl` subclass.

Table D-26   Methods You Typically Override in Your Custom ViewRowImpl Subclass

If you want to...	Write a method like this in your ViewRowImpl subclass
Determine the updateability of an attribute in a conditional way	`isAttributeUpdateable()`

E

ADF Business Components Java EE Design Pattern Catalog

This appendix summarizes the Java Platform, Enterprise Edition (Java EE) design patterns that **ADF Business Components** implements for you.
By using the Oracle Application Development Framework's business components building-blocks and related design time extensions to JDeveloper, you get a prescriptive architecture for building richly functional and cleanly layered Java EE business services with great performance.

Table E-1 provides a brief overview of the numerous design patterns that the ADF Business Components layer implements for you. Some are the familiar patterns from Sun's Java EE BluePrints and some are design patterns that ADF Business Components adds to the list. For details about Java EE BluePrints, see the BluePrints page at the Oracle Technology Network website at `http://www.oracle.com/technetwork/java/index-jsp-136701.html`.

Table E-1 Java EE Design Patterns Implemented by ADF Business Components

Pattern Name and Description	How ADF Business Components Implements It
Model/View/Controller Cleanly separates the roles of data and presentation, allowing multiple types of client displays to work with the same business information.	The ADF **application module** provides a generic implementation of a Model/View/Controller "application object" that simplifies exposing the application data model for any application or service, and facilitates declaratively specifying the boundaries of a logical unit of work. Additional UI-centric frameworks and tag libraries provided in JDeveloper help you implement the view and controller layers.
Interface / Implementation Separation Cleanly separates the API or Interface for components from their implementation class.	ADF Business Components enforces a logical separation of client-tier accessible functionality (via interfaces) and its business tier implementation. JDeveloper handles the creation of custom interfaces and client proxy classes automatically.
Service Locator Abstracts the technical details of locating a service so that the client can use it more easily.	ADF application modules are looked up using a simple configuration object which hides the low-level details of finding the service instance behind the scenes. For Fusion web applications, it also hides the implementation of the **application module pool** usage, a lightweight pool of service components that improves application scalability.

Table E-1 (Cont.) Java EE Design Patterns Implemented by ADF Business Components

Pattern Name and Description	How ADF Business Components Implements It
Inversion of Control A containing component orchestrates the lifecycle of the components it contains, invoking specific methods that you can override at the appropriate times, so as to be able to focus more on what the code should do, instead of when it should be executed.	ADF components contain a number of easy-to-override methods that the framework invokes as needed during the course of application processing.
Dependency Injection Simplifies application code, and increases configuration flexibility by deferring component configuration and assembly to the container.	ADF Business Components configures all its components from externalized XML metadata definition files. At runtime, the framework automatically injects dependent objects like **view object instances** into your application module service component and entity objects into your view rows, implementing lazy loading. It supports runtime factory substitution of components by any customized subclass of that component to simplify onsite application customization scenarios. Much of the ADF Business Components functionality is implemented via dynamic injection of validator and listener subscriptions that coordinate the framework interactions depending on what declarative features have been configured for each component in their XML metadata.
Active Record Avoids the complexity of "anything to anything" object/relational mapping, by providing an object that wraps a row in a database table or view, encapsulates the database access, and adds domain logic on that data.	ADF entity objects handle the database mapping functionality you use most frequently, including inheritance, association, and composition support, so you don't have to focus on object/relational mapping. They also provide a place to encapsulate both declarative business rules and one-off programmatic business domain.
Data Access Objects Prevents unnecessary marshalling overhead by implementing dependent objects as lightweight, persistent classes instead of each as an individual enterprise bean. Isolates persistence details into a single, easy-to-maintain class.	ADF view objects automate the implementation of data access for reading data using SQL statements. ADF entity objects automate persistent storage of lightweight business entities. ADF view objects and entity objects cooperate to provide a sophisticated, performant data access objects layer, where any data queried through a view object can optionally be made fully updatable without requiring that you write any "application plumbing" code.
Session Facade Prevents inefficient client access of entity beans and inadvertent exposure of sensitive business information by wrapping entity beans with a session bean.	ADF application modules are designed to implement a coarse-grained "service facade" architecture in any of their supported deployment modes. When deployed as a service interface, they provide an implementation of the Session Facade pattern automatically.

Table E-1 (Cont.) Java EE Design Patterns Implemented by ADF Business Components

Pattern Name and Description	How ADF Business Components Implements It
Value Object Prevents unnecessary network roundtrips by creating one-off "transport" objects to group a set of related attributes needed by a client program.	ADF Business Components provides an implementation of a generic `Row` object, which is a metadata-driven container of any number and kind of attributes that need to be accessed by a client. The developer can work with the generic `Row` interface and do late-bound `getAttribute("Price")` and `setAttribute("Quantity")`calls, or optionally generate early-bound row interfaces like `OverdueOrdersRow`, to enable type-safe method calls like `getPrice()` and `setQuantity()`. Smarter than just a simple "bag 'o attributes", the ADF `Row` object can be introspected at runtime to describe the number, names, and types of the attributes in the row, enabling sophisticated, generic solutions to be implemented.
Page-by-Page Iterator Prevents sending unnecessary data to the client by breaking a large collection into page-sized "chunks" for display.	ADF Business Components provides an implementation of a generic `RowSet` interface which manages result sets produced by executing view object SQL queries. The `RowSet` interface allows you to set a desired page size, for example 10 rows, and page up and down through the query results in these page-sized chunks. Since data is retrieved lazily, only data the user actually visits will ever be retrieved from the database on the backend, and in the client tier the number of rows in the page can be returned over the network in a single roundtrip.
Fast-Lane Reader Prevents unnecessary overhead for read-only data by accessing JDBC APIs directly. This allows an application to retrieve only the attributes that need to be displayed, instead of finding all of the attributes by primary key when only a few are required by the client. Typically, implementations of this pattern sacrifice data consistency for performance, since queries performed at the raw JDBC level do not "see" pending changes made to business information represented by enterprise beans.	ADF view objects read data directly from the database for best performance; however, they give you a choice regarding data consistency. If updateability and/or consistency with pending changes is desired, you need only associate your view object with the appropriate entity objects whose business data is being presented. If consistency is not a concern, view objects can simply perform the query with no additional overhead. In either case, you never have to write JDBC data access code. You need only provide appropriate SQL statements in XML descriptors.
(Bean) Factory Allows runtime instantiation and configuration of an appropriate subclass of a given interface or superclass based on externally configurable information.	All ADF component instantiation is done based on XML configuration metadata through factory classes allowing runtime substitution of specialized components to facilitate application customization.

Table E-1 (Cont.) Java EE Design Patterns Implemented by ADF Business Components

Pattern Name and Description	How ADF Business Components Implements It
Entity Facade Provides a restricted view of data and behavior of one or more business entities.	ADF view objects can surface any set of attributes and methods from any combination of one or more underlying entity objects to furnish the client with a single, logical value object to work with.
Value Messenger Keeps client value object attributes in sync with the middle-tier business entity information that they represent in a bidirectional fashion.	The ADF Business Components value object implementation coordinates with a client-side value object cache to batch attribute changes to the EJB tier and receive batch attribute updates which occur as a result of middle-tier business logic. The ADF Value Messenger implementation is designed to not require any kind of asynchronous messaging to achieve this effect.
Continuations Gives you the simplicity and productivity of a stateful programming model with the scalability of a stateless web solution.	ADF Business Components application module pooling and state management functionality combine to deliver this value-add. Application module pooling eliminates the need to dedicate application server tier resources to individual users and supports a "stateless with user affinity" optimization that you can tune.

F

ADF Equivalents of Common Oracle Forms Triggers

This appendix provides a summary of how basic tasks performed with the most common Oracle Forms triggers are accomplished using Oracle ADF.

This appendix includes the following sections:

- Validation and Defaulting (Business Logic)
- Query Processing
- Database Connection
- Transaction "Post" Processing (Record Cache)
- Error Handling

Validation and Defaulting (Business Logic)

You will learn how Oracle Forms Validation and Defaulting Triggers are performed using Oracle ADF equivalents.

Table F-1 ADF Equivalents for Oracle Forms Validation and Defaulting Triggers

Forms Trigger	ADF Equivalent
`WHEN-VALIDATE-RECORD` Execute validation code at the record level	In the custom `EntityImpl` class for your entity object, write a public method returning a `boolean` type with a method name like `validate`*`XXXX`*`()` and have it return `true` if the validation succeeds or `false` if the validation fails. Then, add a Method validator for this validation method to your entity object at the entity level. When doing that, you can associate a validation failure message with the rule.
`WHEN-VALIDATE-ITEM` Execute validation code at the field level	In the custom `EntityImpl` class for your entity object, write a public method returning a `boolean` type and accepting a single argument of the same data type as your attribute, having a method name like `validate`*`XXXX`*`()`. Have it return `true` if the validation succeeds or `false` if the validation fails. Then, add a Method validator for this validation method to the entity object at the attribute level for the appropriate attribute. When doing that, you can associate a validation failure message with the rule.
`WHEN-DATABASE-RECORD` Execute code when a row in the data block is marked for insert or update	Override the `addToTransactionManager()` method of your entity object. Write code after calling the super.

Table F-1 (Cont.) ADF Equivalents for Oracle Forms Validation and Defaulting Triggers

Forms Trigger	ADF Equivalent
`WHEN-CREATE-RECORD` Execute code to populate complex default values when a new record in the data block is created, without changing the modification status of the record	Override the `create()` method of your entity object and after calling the super, use appropriate `set`*`AttrName`*`()` methods to set default values for attributes as necessary. To immediately set a primary key attribute to the value of a sequence, construct an instance of the `SequenceImpl` helper class and call its `getSequenceNumber()` method to get the next sequence number. Assign this value to your primary key attribute. If you want to wait to assign the sequence number until the new record is saved, but still without using a database trigger, you can use this technique in an overridden `prepareForDML()` method in your entity object. If instead you want to assign the primary key from a sequence using your own `BEFOREINSERTFOREACHROW` database trigger, then use the special data type called `DBSequence` for your primary key attribute instead of the regular `Number` type.
`WHEN-REMOVE-RECORD` Execute code whenever a row is removed from the data block	Override the `remove()` method of your entity object and write code either before or after calling the super.

Query Processing

You will learn how Oracle Forms Query Processing Triggers are performed using Oracle ADF equivalents.

Table F-2 ADF Equivalents for Oracle Forms Query Processing Triggers

Forms Trigger	ADF Equivalent
`PRE-QUERY` Execute logic before executing a query in a data block, typically to set up values for Query-by-Example criteria in the "example record"	Override the `executeQueryForCollection()` method on your **view object** class and write code before calling the super.
`ON-COUNT` Override default behavior to count the query hits for a data block	Override the `getQueryHitCount()` method in your view object and do something instead of calling the super.

Table F-2 (Cont.) ADF Equivalents for Oracle Forms Query Processing Triggers

Forms Trigger	ADF Equivalent
`POST-QUERY` Execute logic after retrieving each row from the data source for a data block.	Generally instead of using a `POST-QUERY` style technique to fetch descriptions from other tables based on foreign key values in the current row, in ADF it's more efficient to build a view object that has multiple participating entity objects, joining in all the information you need in the query from the main table, as well as any auxiliary or lookup-value tables. This way, in a single roundtrip to the database you get all the information you need. If you still need a per-fetched-row trigger like `POST-QUERY`, override the `createInstanceFromResultSet()` method in your view object class.
`ON-LOCK` Override default behavior to attempt to acquire a lock on the current row in the data block	Override the `lock()` method in your entity object class and do something instead of calling the super.

Database Connection

You will learn how Oracle Forms Database Connection Triggers are performed using Oracle ADF equivalents.

Table F-3 ADF Equivalents for Oracle Forms Database Connection Triggers

Forms Trigger	ADF Equivalent
`POST-LOGON` Execute logic after logging into the database	Override the `afterConnect()` method on your custom application module. Since an **application module instance** can stay connected while serving different logical client sessions, you can override the `prepareSession()` method, which is fired after initial login, as well as after any time the application module is accessed by a user that was different from the one that accessed it last time.
`PRE-LOGOUT` Execute logic before logging out of the database	Override the `beforeDisconnect()` method on your custom application module class.

Transaction "Post" Processing (Record Cache)

You will learn how Oracle Forms Transactional Processing Triggers are performed using Oracle ADF equivalents.

Table F-4 ADF Equivalents for Oracle Forms Transactional Triggers

Forms Trigger	ADF Equivalent
`PRE-COMMIT` Execute code before commencing processing of the changed rows in all data blocks in the transaction	Override the `commit()` method in a custom `DBTransactionImpl` class and write code before calling the super. **Note:** For an overview of creating and using a custom `DBTransaction` implementation, see Creating a Custom Database Transaction Framework Extension Class.
`PRE-INSERT` Execute code before a new row in the data block is inserted into the database during "post" processing	Override the `doDML()` method in your entity class, and if the `operation` equals `DML_INSERT`, then write code *before* calling the super.
`ON-INSERT` Override default processing for inserting a new row into the database during "post" processing	Override the `doDML()` method in your entity class, and if the `operation` equals `DML_INSERT`, then write code *instead of* calling the super.
`POST-INSERT` Execute code after new row in the data block is inserted into the database during "post" processing	Override the `doDML()` method in your entity class, and if the `operation` equals `DML_INSERT`, then write code *after* calling the super.
`PRE-DELETE` Execute code before a row removed from the data block is deleted from the database during "post" processing	Override the `doDML()` method in your entity class, and if the `operation` equals `DML_DELETE`, then write code *before* calling the super.
`ON-DELETE` Override default processing for deleting a row removed from the data block from the database during "post" processing	Override the `doDML()` method in your entity class, and if the `operation` equals `DML_DELETE`, then write code *instead of* calling the super.
`POST-DELETE` Execute code after a row removed from the data block is deleted from the database during "post" processing	Override the `doDML()` method in your entity class, and if the `operation` equals `DML_DELETE`, then write code *after* calling the super.
`PRE-UPDATE` Execute code before a row changed in the data block is updated in the database during "post" processing	Override the `doDML()` method in your entity class, and if the `operation` equals `DML_UPDATE`, then write code *before* calling the super.
`ON-UPDATE` Override default processing for updating a row changed in the data block from the database during "post" processing	Override the `doDML()` method in your entity class, and if the `operation` equals `DML_UPDATE`, then write code *instead of* calling the super.
`POST-UPDATE` Execute code after a row changed in the data block is updated in the database during "post" processing	Override the `doDML()` method in your entity class, and if the `operation` equals `DML_UPDATE`, then write code *after* calling the super.

Table F-4 (Cont.) ADF Equivalents for Oracle Forms Transactional Triggers

Forms Trigger	ADF Equivalent
`POST-FORMS-COMMIT` Execute code after Forms has "posted" all necessary rows to the database, but before issuing the data commit to end the transaction	If you want a single block of code for the whole transaction, you can override the `doCommit()` method in a custom `DBTransactionImpl` object and write code before calling the super. To execute entity-specific code before commit for each affected entity in the transaction, override the `beforeCommit()` method on your entity object, and write code there.
`POST-DATABASE-COMMIT` Execute code after database transaction has been committed	Override the `commit()` method in a custom `DBTransactionImpl` class, and write code *after* calling the super.

Error Handling

You will learn how Oracle Forms Error Handling Triggers are performed using Oracle ADF equivalents.

Table F-5 ADF Equivalents for Oracle Forms Error Handling Triggers

Forms Trigger	ADF Equivalent
`ON-ERROR` Override default behavior for handling an error	Install a custom error handler (`DCErrorHandler`) on the ADF `BindingContext`.

G

Performing Common Oracle Forms Tasks in Oracle ADF

This appendix describes how common Oracle Forms tasks are implemented in Oracle ADF. In Oracle Forms, you do some tasks in the data block, and others in the UI. For this reason, the appendix is divided into two sections: tasks that relate to data, and tasks that relate to the UI.
This appendix includes the following sections:

Concepts Familiar to Oracle Forms Developers

ADF Business Components provide components that implement functionality similar to what you are used to in the Java world, with responsibilities divided along the cleanly-separated functional lines.

ADF Business Components implements all of the data-centric aspects of Oracle Forms runtime functionality, but in a way that is independent of the user interface. Responsibilities between the querying and entity-related functions are cleanly separated, resulting in better reuse, as described in Table G-1.

Table G-1 Concepts for Oracle Forms Developers

This concept	Maps to Oracle ADF	With these similarities
Headless Form Module	Application Module	The **application module** component is the "data portion" of the form. The application module is a smart data service containing a data model of master-detail-related queries that your client interface needs to work with. It also provides a transaction and database connection used by the components it contains. It can contain form-level procedures and functions, referred to as service methods, that are encapsulated within the service implementation. You can decide which of these procedures and functions should be private and which ones should be public.
Forms Record Manager	Entity Object	The entity object component implements the "validation and database changes" portion of the data block functionality. In the Forms runtime, this duty is performed by the record manager. The record manager is responsible for keeping track of which of the rows in the data block have changed, for firing the block-level and item-level validation triggers when appropriate, and for coordinating the saving of changes to the database. This is exactly what an entity object does for you. The entity object is a component that represents your business domain entity through an underlying database table. The entity object gives you a single place to encapsulate business logic related to validation, defaulting, and database modification behavior for that business object.

Table G-1 (Cont.) Concepts for Oracle Forms Developers

This concept	Maps to Oracle ADF	With these similarities
Data Block	View Object	The **view object** component performs the "data retrieval" portion of the data block functionality. Each view object encapsulates a SQL query, and at runtime each one manages its own query result set. If you connect two or more view objects in a **master-detail relationship**, that coordination is handled automatically. While defining a view object, you can link any of its query columns to underlying entity objects. By capturing this information, the view object and entity object can cooperate automatically for you at runtime to enforce your domain business logic, regardless of the "shape" of the business data required by the user's task.

Performing Tasks Related to Data

In Oracle Forms, you perform some tasks in the data block, and others in the UI. You will learn how tasks that are particularly related to data are performed.

In Oracle Forms, tasks that relate solely to data are performed in the data block. In **Oracle ADF**, these tasks are done on the business components that persist data (entity objects) and on the objects that query data (view objects).

How to Retrieve Lookup Display Values for Foreign Keys

In Oracle Forms, an editable table often has foreign key lookup columns to other tables. The user-friendly display values corresponding to the foreign key column values exist in related tables. You often need to present these related display values to the user.

In Oracle Forms, this was a complicated task that required adding nondatabase items to the data block, adding a block-level `POST-QUERY` trigger to the data block, and writing a SQL select statement for each foreign key attribute. Additionally, if the user changed the data, you needed to sync the foreign key values with an item-level `WHEN-VALIDATE-ITEM` trigger. This process is much easier in Oracle ADF.

Implementation of the task in Oracle ADF

1. Create a view object that includes the following:
   - The main, editable entity object as the primary entity usage
   - Secondary "reference" entity usages for the one or more associated entities whose underlying tables contain the display text

   For more information, see How to Create Joins for Entity-Based View Objects.
2. Select the desired attributes (at least the display text) from the secondary entity usages as described in How to Select Additional Attributes from Reference Entity Usages.

   At runtime, the data for the main entity and all related lookup display fields is retrieved from the database in a single join.

   If the user can change the data, no additional steps are required. If the user changes the value of a foreign key attribute, the reference information is automatically retrieved for the new, related row in the associated table.

How to Get the Sysdate from the Database

In Oracle Forms, when you wanted to get the current date and time, you retrieved the `sysdate` from the database. In Oracle ADF, you also have the option of getting the system date using a Java method or a Groovy expression.

Implementation of the task in Oracle ADF

To get the system date from the database, you can use the following Groovy expression at the entity level:

```
DBTransaction.currentDbTime
```

Note:

The `DBTransaction` reference is for entity-level Groovy expressions only.

If you want to assign a default value to an attribute using this Groovy expression, see How to Define a Default Value Using an Expression.

To get the system date from Java, you call the `getCurrentDate()` method. For more information, see Accessing the Current Date and Time.

How to Implement an Isolation Mode That Is Not Read Consistent

In Oracle Forms, you might have been concerned with *read consistency*, that is, the ability of the database to deliver the state of the data at the time the SQL statement was issued.

Implementation of the task in Oracle ADF

If you use an entity-based view object, the query sees the changes currently in progress by the current user's session in the pending transaction. This is the default behavior, and the most accurate.

If instead, you want a snapshot of the data on the database without considering the pending changes made by the current user, you can use a read-only view object and reexecute the query to see the latest committed database values. For more information on read-only view objects, see How to Create a Custom SQL Mode View Object.

How to Implement Calculated Fields

Calculated fields are often used to show the sum of two values, but they could also be used for the concatenated value of two or more fields, or the result of a method call.

Implementation of the task in Oracle ADF

Calculated attributes are usually not stored in the database, as their values can easily be obtained programmatically. Attributes that are used in the middle tier, but that are not stored in the database are called *transient attributes*. Transient attributes can be defined at the entity object level or the view object level.

If a transient attribute will be used by more than one view object that might be based on an entity object, then define the attribute at the entity object level. Otherwise, define the transient attribute at the view object level for a particular view object.

To define transient attributes at the entity object level, see Adding Transient and Calculated Attributes to an Entity Object. To define transient attributes at the view object level, see Adding Calculated and Transient Attributes to a View Object.

How to Implement Mirrored Items

In Oracle Forms, you may be used to using mirrored items to show two or more fields that share identical values.

Implementation of the task in Oracle ADF

There is no need to have mirrored items in Oracle ADF, because the UI and data are separated. The same view object can appear on any number of pages, so you don't need to create mirrored items that have the same value. Likewise, a form could have the same field represented in more than one place and it would not have to be mirrored.

How to Use Database Columns of Type CLOB or BLOB

If you are used to working with standard database types, you may be wondering how to use the `CLOB` and `BLOB` types in Oracle ADF.

Implementation of the task in Oracle ADF

In Oracle ADF, use the built-in data types `ClobDomain` or `BlobDomain`. These are automatically created when you reverse-engineer entity objects or view objects from existing tables with these column types. For more information, see How to Set Database and Java Data Types for an Entity Object Attribute.

Performing Tasks Related to the User Interface

In Oracle Forms, you perform some tasks in the data block, and others in the UI. You will learn how tasks particularly related to the UI are performed.

In Oracle ADF, common UI-related tasks (such as master-detail screens, popup list of values, and page layout) are handled quite differently than they were in Oracle Forms.

How to Lay Out a Page

Oracle Forms is based on an absolute pixel or point-based layout, as compared to the container-based approach of JSF.

Implementation of the task in Oracle ADF

See *Developing Web User Interfaces with Oracle ADF Faces* for information on how to lay out a page in Oracle ADF.

How to Stack Canvases

In Oracle Forms, stacked canvases were often used to hide and display areas of the screen.

Implementation of the task in Oracle ADF

The analog of stacked canvases in Oracle ADF is panels (layout containers) with the rendered property set to `true` or `false`. See *Developing Web User Interfaces with Oracle ADF Faces* for more information.

How to Implement a Master-Detail Screen

Master-detail relationships in Oracle ADF are coordinated through a view link. A view link is conceptually similar to an Oracle Forms *relation*.

Implementation of the task in Oracle ADF

For information on how create view links, see Working with Multiple Tables in a Master-Detail Hierarchy. Once you have established a relationship between two view objects with a view link, see How to Enable Active Master-Detail Coordination in the Data Model.

How to Implement an Enter Query Screen

In Oracle Forms, another common task was creating an *enter query screen*. That is, a screen that starts in Find mode.

Implementation of the task in Oracle ADF

In Oracle ADF, this accomplished with a search form. Complete information on how to create a search form is covered in Creating ADF Databound Search Forms. In particular, you may want to look at How to Set Search Form Properties on the View Criteria.

How to Implement an Updatable Multi-Record Table

In Oracle Forms, you may be used to creating tables where you can edit and insert many records at the same time. This can be slightly more complicated when using a JSF page in Oracle ADF, because the operations to edit an existing record and to create a new record are not the same.

Implementation of the task in Oracle ADF

In Oracle ADF, this is done using an input table. To create an input table, see Creating an Input Table.

How to Create a Popup List of Values

In Oracle Forms, it was simple to create a **list of values (LOV)** object and then associate that object with a field in a declarative manner. This LOV would display a popup window and provide the following capabilities:

- Selection of modal values
- Query area at the top of the LOV dialog
- Display of multiple columns
- Automatic reduction of LOV contents, possibly based on the contents of the field that launched the LOV

- Automatic selection of the list value when only one value matches the value in the field when the LOV function is invoked
- Validation of the field value based on the values cached by the LOV
- Automatic popup of the LOV if the field contents are not valid

Implementation of the task in Oracle ADF

To implement a popup list in Oracle ADF, you configure one of the view object's attributes to be of LOV type, and select **Input Text with List of Values** as the style for its UI hint. For a description of how to do this, see Working with List of Values (LOV) in View Object Attributes.

How to Implement a Dropdown List as a List of Values

In Oracle Forms, you could create a list of values (LOV) object and then associate that object with a field in a declarative manner. In Oracle ADF, you can implement an LOV (lookup-value) screen with a search item, usable for a lookup field with many possible values.

Implementation of the task in Oracle ADF

To implement a dropdown list in Oracle ADF, you configure one of the view object's attributes to be of LOV type, and select **Input Text with List of Values** as the style for its UI hint. For a description of how to do this, see Working with List of Values (LOV) in View Object Attributes.

How to Implement a Dropdown List with Values from Another Table

In Oracle Forms, you could create a list of values (LOV) object and then associate that object with a field in a declarative manner. In Oracle ADF, you can implement a dropdown list with string values from a different table. These string values populate the field with an id code that is valid input in the table that the screen is based on.

Implementation of the task in Oracle ADF

To implement a dropdown list of this type in Oracle ADF, you configure one of the view object's attributes to be of LOV type, and select **Choice List** as the style for its UI hint. For a description of how to do this, see Working with List of Values (LOV) in View Object Attributes.

How to Implement Immediate Locking

In Oracle ADF, you can lock a record in the database at the first moment it is obvious that the user is going to change a specific record.

Implementation of the task in Oracle ADF

Immediate row locking can be configured in ADF Business Components, although it is not the default and is typically not used in web application scenarios. For web applications, use the default configuration setting `jbo.locking.mode=optimistic`. For more information, see How to Confirm That Fusion Web Applications Use Optimistic Locking.

How to Throw an Error When a Record Is Locked

When a record has been locked by a user, it's helpful to throw an error to let other users know that the record is not currently updatable.

Implementation of the task in Oracle ADF

Locking rows and throwing an exception if the row is already locked is built-in ADF Business Components functionality. There are a couple of different ways that you can handle the error message, depending on whether you want a static error message or a custom message with information about the current row.

- To throw a static message, register a custom message bundle in your data model project to substitute the default `RowAlreadyLockedException`'s error message with something more meaningful or user-friendly.
- To throw a message that contains information about the row, override the `lock()` method on the entity object, using a try/catch block to catch the `RowAlreadyLocked` exception. After you catch the exception, you can throw an error message that might contain more specific information about the current row.

H

Deploying ADF Applications to GlassFish

This appendix describes how to deploy Oracle ADF applications to a GlassFish application server. It describes how to create deployment profiles, how to create deployment descriptors, and how to deploy the application.
This appendix includes the following sections:

- About Deploying ADF Applications to GlassFish Server
- Running an ADF Application in Integrated WebLogic Server
- Preparing the Application
- Deploying the Application
- Testing the Application and Verifying Deployment

About Deploying ADF Applications to GlassFish Server

In order to run an ADF application, it must be first deployed on an application server such as GlassFish Server. Deployment in the context of web applications is the act of installing the application on a server that allows requests to be handled and so on.

Deployment is the process of packaging application files as an archive file and transferring that file to a target GlassFish application server. You can use JDeveloper to deploy ADF applications directly to the GlassFish Server, or indirectly to an archive file as the deployment target, and then install this archive file to the target GlassFish Server. For application development, you can also use JDeveloper to run an application in Integrated WebLogic Server.

You can use JDeveloper to:

- Run applications in Integrated WebLogic Server

  You can run and debug applications using Integrated WebLogic Server and then deploy to standalone GlassFish Server.
- Deploy directly to the standalone GlassFish Server

  You can deploy applications directly to the standalone GlassFish Server by creating a connection to the server and choosing the name of that server as the deployment target.
- Deploy to an archive file

  You can deploy applications indirectly by choosing an EAR file as the deployment target. The archive file can subsequently be installed on the target GlassFish Server.

Developing Applications with Integrated WebLogic Server

If you are developing an application in JDeveloper and you want to run the application in Integrated WebLogic Server, you do not need to perform the tasks required for deploying directly to Oracle WebLogic Server or to an archive file.

JDeveloper has a default connection to Integrated WebLogic Server and does not require any deployment profiles or descriptors. Integrated WebLogic Server has a preconfigured domain that includes the ADF libraries, as well as the -Djps.app.credential.overwrite.allowed=true setting, that are required to run ADF applications. You can run an application by choosing **Run** from the JDeveloper main menu.

You debug the application using the features described in Testing and Debugging ADF Components.

Developing Applications to Deploy to Standalone GlassFish Server

Typically, for deployment to standalone application servers, you test and develop your application by running it in Integrated WebLogic Server. You can then test the application further by deploying it to standalone GlassFish Server.

In general, you use JDeveloper to prepare the application or project for deployment by:

- Creating a connection to the target GlassFish Server
- Creating deployment profiles (if necessary)
- Creating deployment descriptors that are specific to GlassFish
- Updating application.xml and web.xml to be compatible with the GlassFish Server (if required)

You must already have an installed GlassFish Server. For instructions on obtaining and installing GlassFish, see https://javaee.github.io/glassfish/download.

> **Note:**
>
> For information about the GlassFish Server version supported by JDeveloper and ADF, see the Certification Information link for your JDeveloper release on OTN at http://www.oracle.com/technetwork/developer-tools/jdev/documentation/index.html.

You must also prepare the GlassFish Server for ADF application deployment. For more information, see Configuring GlassFish Server in *Administering Oracle ADF Applications*.

- Installing the ADF runtime into the GlassFish Server installation:
- Setting JVM cache size and simple option
- Enabling Secure Admin on the GlassFish Server to allow for remote logins and remote connections in order for JDeveloper to run on a different machine than GlassFish.
- Creating a global JDBC data source for applications that require a connection to a data source

After the application and the GlassFish Server have been prepared, you can:

- Use JDeveloper to:
  - Directly deploy to the GlassFish Server using the deployment profile and the application server connection.

  - Deploy to an EAR file using the deployment profile. For ADF applications, WAR files can be deployed only as part of an EAR file.
- Use the GlassFish Server's administration tools to deploy the EAR file created in JDeveloper.

Running an ADF Application in Integrated WebLogic Server

In order to test run an ADF application during development, you can use JDeveloper to run an application in Integrated WebLogic Server. You do not have to manually complete many of the steps that are necessary for deployment to a standalone server.

JDeveloper is installed with Integrated WebLogic Server which you can use to test and develop your application. For most development purposes, Integrated WebLogic Server will suffice. When your application is ready to be tested, you can select the run target and then choose the **Run** command from the main menu.

> **Note:**
>
> The first time you run an application in Integrated WebLogic Server, the Create Default Domain dialog appears for you to define an administrative password for the new domain.

When you run the application target, JDeveloper detects the type of Java EE module to deploy based on artifacts in the projects and workspace. JDeveloper then creates an in-memory deployment profile for deploying the application to Integrated WebLogic Server. JDeveloper copies project and application workspace files to an "exploded EAR" directory structure. This file structure closely resembles the EAR file structure that you would have if you were to deploy the application to an EAR file. JDeveloper then follows the standard deployment procedures to register and deploy the "exploded EAR" files into Integrated WebLogic Server. The "exploded EAR" strategy reduces the performance overhead of packaging and unpackaging an actual EAR file.

In summary, when you select the run target and run the application in Integrated WebLogic Server, JDeveloper:

- Detects the type of Java EE module to deploy based on the artifacts in the project and application
- Creates a deployment profile in memory
- Copies project and application files into a working directory with a file structure that would simulate the "exploded EAR" file of the application.
- Performs the deployment tasks to register and deploy the simulated EAR into Integrated WebLogic Server

> **Note:**
>
> JDeveloper ignores the deployment profiles that were created for the application when you run the application in Integrated WebLogic Server.

The application will run in the base domain in Integrated WebLogic Server. This base domain has the same configuration as a base domain in a standalone WebLogic Server instance. In other words, this base domain will be the same as if you had used the Oracle Fusion Middleware Configuration Wizard to create a base domain with the default options in a standalone WebLogic Server instance.

JDeveloper will extend this base domain with the necessary domain extension templates, based on the JDeveloper technology extensions. For example, if you have installed JDeveloper Studio, JDeveloper will automatically configure the Integrated WebLogic Server environment with the ADF runtime template (JRF Fusion Middleware runtime domain extension template).

You can explicitly create a default domain for Integrated WebLogic Server. You can use the default domains to run and test your applications. Open the Application Servers window, right-click **IntegratedWebLogicServer** and choose **Create Default Domain**.

How to Run an Application in Integrated WebLogic Server

You can test an application by running it in Integrated WebLogic Server. You can also set breakpoints and then run the application within the ADF Declarative Debugger.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you run an application in Integrated WebLogic Server. For more information, see Running an ADF Application in Integrated WebLogic Server.

To run an application in Integrated WebLogic Server:

1. In the Applications window, select the project, unbounded task flow, JSF page, or file as the run target.
2. Right-click the run target and choose **Run** or **Debug**.

   The Create Default Domain dialog displays the first time you run your application and start a new domain in Integrated WebLogic Server. Use the dialog to define an administrator password for the new domain. Passwords you enter can be eight characters or more and must have a numeric character.

Preparing the Application

Once your application is developed, you must perform some essential tasks to deploy the ADF application to a standalone GlassFish Server. You also need to configure GlassFish-specific security within the ADF application.

Before you deploy an ADF application to a standalone GlassFish Server, you must perform prerequisite tasks within JDeveloper to prepare the application for deployment.

The tasks are:

- How to Create a Connection to the Target Application Server
- How to Create Deployment Profiles
- How to Create and Edit Deployment Descriptors
- How to Enable JDBC Data Source for GlassFish

- How to Change Initial Context Factory

> **Note:**
>
> ADF Security is not supported on ADF Essentials for GlassFish. You should configure GlassFish-specific security within the ADF application. For information about configuring GlassFish security, see `http://docs.oracle.com/cd/E18930_01/html/821-2418/beabg.html#scrolltoc`.

How to Create a Connection to the Target Application Server

You can deploy applications to the GlassFish Server via JDeveloper application server connections.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create an application server connection. For more information, see Preparing the Application.

You will need to complete this task:

Install the GlassFish Server.

To create a connection to an application server:

1. Launch the Application Server Connection wizard.

   You can:

   - In the Application Servers window, right-click **Application Servers** and choose **New Application Server**.
   - In the New Gallery, expand **General**, select **Connections** and then **Application Server Connection**, and click **OK**.
   - In the Resources window, choose **New** > **IDE Connections > Application Server**.
2. In the Create Application Server Connection dialog, on the Usage page, select **Standalone Server**.
3. On the Name and Type page, enter a connection name.
4. In the **Connection Type** dropdown list, choose:
   - **GlassFish 3.1** to create a connection to the GlassFish Server
5. Click **Next**.
6. On the Authentication page, enter a user name and password for the administrative user authorized to access the GlassFish Server.
7. Click **Next**.
8. On the Configuration page, enter the information for the GlassFish Server:
   - **Host Name**: Enter the name of the machine where the GlassFish Server is running. If no name is entered, the name defaults to localhost.

- **RMI Port**: If necessary, change to the RMI port number for the server. The default is 8686.
- **HTTP Port**: By default, GlassFish listens to port 8080 for HTTP requests. If necessary, you can change the port number here.
- **Admin HTTP Port:** By default, GlassFish uses port 4848 for administration. If necessary, you can change the port number here.
- **Secure Admin Enabled**: select if you want GlassFish to allow remote logins and remote connections

> **Note:**
>
> If you are using GlassFish 3.1.2 and you are running JDeveloper on a different machine than GlassFish, you need to enable Secure Admin in GlassFish to allow remote logins. You do not need to enable Secure Admin if both JDeveloper and GlassFish are on the same machine.

9. Click **Next**.
10. On the Test page, click **Test Connection** to test the connection.

    JDeveloper performs several types of connections tests. The JSR-88 test must pass for the application to be deployable. If the test fails, return to the previous pages of the wizard to fix the configuration.
11. Click **Finish**.

How to Create Deployment Profiles

A **deployment profile** defines the way the application is packaged into the archive that will be deployed to the target environment. The deployment profile:

- Specifies the format and contents of the archive file that will be created
- Lists the source files, deployment descriptors, and other auxiliary files that will be packaged
- Describes the type and name of the archive file to be created
- Highlights dependency information, platform-specific instructions, and other information

You need a WAR deployment profile for each web user interface project that you want to deploy in your application. You need an application-level EAR deployment profile and you must select the projects (such as WAR profiles) to include from a list. When the application is deployed, the EAR file will include all the projects that were selected in the deployment profile.

> **Note:**
>
> If you create your project or application using the ADF Fusion Web Application template, JDeveloper automatically creates default WAR and EAR deployment profiles. Typically, you would not need to edit or create deployment profiles manually.

For ADF applications, you can deploy the application only as an EAR file. The WAR files that are part of the application should be included in the EAR file when you create the deployment profile.

Creating a WAR Deployment Profile

You will need to create a WAR deployment profile for each web-based project you want to package into the application. Typically, the WAR profile will include the dependent data model projects it requires.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create a WAR deployment profile. For more information, see Preparing the Application.

You will need to complete this task:

Create web-based projects. If you used the ADF Fusion Web Application template, you should already have a default WAR deployment profile.

To create WAR deployment profiles for an application:

1. In the Applications window, right-click the web project that you want to deploy and choose **New** and then **From Gallery**.

   You will create a WAR profile for each web project.

2. In the New Gallery, expand **General**, select **Deployment Profiles** and then **WAR File**, and click **OK**.

   If you don't see **Deployment Profiles** in the **Categories** tree, click the **All Features** tab.

3. In the Create Deployment Profile -- WAR File dialog, enter a name for the project deployment profile and click **OK**.

4. In the Edit WAR Deployment Profile Properties dialog, choose items in the left pane to open dialog pages in the right pane. Configure the profile by setting property values in the pages of the dialog. Note that some items list below may not be in your particular application.

   - Ensure that **ADF Faces Runtime 11**, **JSTL 1.2,** and the ADFm, ADFc, and ADFv libraries are marked deployed in the **WEB-INF/lib/Contributors** panel.
   - Select **File Groups > Web Files > Filters > WEB-INF** and deselect **weblogic.xml**.
   - Select **File Groups > Web Files/Classes > Filters > META-INF** and deselect **ejb-jar.xml**.
   - Select **File Groups > Web Files/Classes > Filters > path** and deselect **serviceinterface**.
   - Select **File Groups > Web Files/Classes > Filters > path** and deselect **server**.
   - You might also want to change the Java EE web context root setting. To do so, choose **General** in the left pane.

     By default, when **Use Project's Java EE Web Context Root** is selected, the associated value is set to the project name, for example, `Application1-`

`Project1-context-root`. You need to change this if you want users to use a different name to access the application.

- Select **Glassfish 3.1** as the **Platform**

5. Click **OK** to exit the Deployment Profile Properties dialog.
6. Click **OK** again to exit the Project Properties dialog.
7. Repeat Steps 1 through 6 for all web projects that you want to deploy.

Creating an Application-Level EAR Deployment Profile

The EAR file contains all the necessary application artifacts for the application to run in the application server. If you used the ADF Fusion Web Application template, you should already have a default EAR deployment profile. For more information about the EAR file, see What You May Need to Know About EAR Files and Packaging.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create an application-level EAR deployment profile. For more information, see Preparing the Application.

You will need to complete this task:

> Create the WAR deployment profiles, as described in Creating a WAR Deployment Profile.

To create an EAR deployment profile for an application:

1. In the Applications window, right-click the application and choose **New** and then **From Gallery**.

   You will create an EAR profile for the application.

2. In the New Gallery, expand **General**, select **Deployment Profiles** and then **EAR File,** and click **OK**.

   If you don't see **Deployment Profiles** in the Categories tree, click the **All Features** tab.

3. In the Create Deployment Profile -- EAR File dialog, enter a name for the application deployment profile and click **OK**.
4. In the Edit EAR Deployment Profile Properties dialog, choose items in the left pane to open dialog pages in the right pane. Configure the profile by setting property values in the pages of the dialog.
   - Select **File Groups > Application Descriptors > Filters** and deselect **weblogic-application.xml**.
   - Select **Application Assembly** and then in the **Java EE Modules** list, select all the project profiles that you want to include in the deployment, including any **WAR** profiles.
   - Select **Platform**, select the application server you are deploying to, and then select the target application connection from the **Target Connection** dropdown list. For GlassFish, select **GlassFish 3.1**.
5. Click **OK** again to exit the Edit EAR Deployment Profile Properties dialog.
6. Click **OK** again to exit the Application Properties dialog.

> **Note:**
>
> To verify that your customization classes are put correctly in the EAR class path, you can deploy the EAR profile to file system. Then you can examine the EAR to make sure that the customization class JAR is available in the EAR class path (the `EAR/lib` directory) and not available in the WAR class path (the `WEB-INF/lib` and `WEB-INF/classes` directories).

Viewing and Changing Deployment Profile Properties

After you have created a deployment profile, you can view and change its properties.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you view and change deployment profile properties. For more information, see Preparing the Application.

To view, edit, or delete a project's deployment profile:

1. In the Applications window, right-click the project and choose **Project Properties**.
2. In the Project Properties dialog, click **Deployment**.

   The **Deployment Profiles** list displays all profiles currently defined for the project.
3. In the list, select a deployment profile.
4. To edit or delete a deployment profile, click **Edit** or **Delete**.

How to Create and Edit Deployment Descriptors

Deployment descriptors are server configuration files that define the configuration of an application for deployment and that are deployed with the Java EE application as needed. The deployment descriptors that a project requires depend on the technologies the project uses and on the type of the target application server. Deployment descriptors are XML files that can be created and edited as source files, but for most descriptor types, JDeveloper provides dialogs or an overview editor that you can use to view and set properties. If you cannot edit these files declaratively, JDeveloper opens the XML file in the source editor for you to edit its contents.

In addition to the standard Java EE deployment descriptors (for example, `application.xml` and `web.xml`), you can also have deployment descriptors that are specific to your target application server.

Creating Deployment Descriptors

JDeveloper automatically creates many of the required deployment descriptors for you. If they are not present, or if you need to create additional descriptors, you can use JDeveloper to create them.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create deployment descriptors. For more information, see Preparing the Application.

You will need to complete this task:

Check to see whether JDeveloper has already generated deployment descriptors.

To create a deployment descriptor:

1. In the Applications window, expand the user interface project, select **WEB-INF** and click **New** and then **From Gallery**.
2. In the New Gallery, expand **General**, choose **XML Document** and click **OK**.
3. In the Create XML File dialog, enter a name for your descriptor and click **OK**.

   For instance, you may want to create a `glassfish-web.xml` to define GlassFish-specific information.
4. In the overview editor, add entries for GlassFish.

   The following example shows a sample `glassfish-web.xml` descriptor.

   The `glassfish-web.xml` descriptor can be used to add additional settings that are not defined in `web.xml` or to modify the settings that are already defined in `web.xml`. For instance, you can override the `jdbc/Connections1DS` setting defined in `web.xml` by adding the entries `res-ref-name` and `jndi-name` to `glassfish-web.xml` as shown below. After setting `glassfish-web.xml`, `jdbc/NewConnections1DS` will be used instead of `jdbc/Connections1DS`. Note that the `res-ref-name` section is optional.

```
<glassfish-web-app>
    <context-root>DeptApp-ViewController-context-root</context-root>
    <resource-ref>
        <res-ref-name>jdbc/Connection1DS</res-ref-name>
        <jndi-name>jdbc/NewConnection1DS</jndi-name>
    </resource-ref>
    <class-loader delegate="false"/>
    <property value="true" name="useBundledJsf"/>
</glassfish-web-app>
```

Note:

For EAR files, do not create more than one deployment descriptor file of the same type per application or workspace. These files can be assigned to projects, but have application workspace scope. If multiple projects in an application have the same deployment descriptor, the one belonging to the launched project will supersede the others. This restriction applies to `application.xml`.

The best place to create an application-level descriptor is in the **Descriptors** node of the Application Resources panel in the Applications window. This ensures that the application is created with the correct descriptors.

Application-level descriptors created in the project will be ignored at runtime. Only the application resources descriptors or descriptors generated at the EAR level will be used by the runtime.

Viewing or Modifying Deployment Descriptor Properties

After you have created a deployment descriptor, you can change its properties by using JDeveloper dialogs or by editing the file in the source editor. The deployment descriptor is an XML file (for example, `application.xml`) typically located under the Application Sources node.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you view or modify deployment descriptors. For more information, see Preparing the Application.

To view or change deployment descriptor properties:

1. In the Applications window or in the Application Resources panel, double-click the deployment descriptor.
2. In the overview editor, select either the **Overview tab** or the **Source tab**, and configure the descriptor by setting property values.

   If the overview editor is not available, JDeveloper opens the file in the source editor.

Configuring the application.xml File for Application Server Compatibility

You may need to configure your `application.xml` file to be compliant with your Java EE 1.5.

> **Note:**
>
> Typically, your project has an `application.xml` file that is compatible and you would not need to perform this procedure.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you configure the `application.xml` file. For more information, see Preparing the Application.

To configure the application.xml file:

1. In the Applications window, right-click the application and choose **New** and then **From Gallery**.
2. In the New Gallery, expand **General**, select **Deployment Descriptors** and then **Java EE Deployment Descriptor Wizard**, and click **OK**.
3. In the Create Java EE Deployment Descriptor dialog, on the Select Descriptor page, select **application.xml** and click **Next**.
4. On the Select Version page, select **5.0** to be compatible with Java EE 5.0 or higher and click **Next**.
5. On the Summary page, click **Finish**.
6. Edit the `application.xml` file with the appropriate values.

Configuring the web.xml File for GlassFish Server Compatibility

You may need to configure your `web.xml` file to be compliant with your Java EE version).

The following example shows a `web.xml` file with a setting for a JDBC connection. For more information and other settings that are required, see web.xml.

```
<resource-ref>
      <description>DB Connection</description>
      <res-ref-name>jdbc/Connection1DS</res-ref-name>
      <res-type>javax.sql.DataSource</res-type>
      <res-auth>Container</res-auth>
</resource-ref>
```

> **Note:**
>
> Typically, your project has a `web.xml` file that is compatible and you would not need to perform this procedure. JDeveloper creates a starter `web.xml` file when you create a project.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you configure the `web.xml` file. For more information, see Preparing the Application.

To configure the web.xml file:

1. In the Applications window, right-click the project and choose **New** and then **From Gallery**.
2. In the New Gallery, expand **General**, select **Deployment Descriptors** and then **Java EE Deployment Descriptor**, and click **OK**.
3. In the Create Java EE Deployment Descriptor dialog, on the Select Descriptor page, select **web.xml** and click **Next**.
4. On the Select Version page, select the version you want to use and click **Next**.

   Version 3.0 is compatible with Java EE 6.0 or higher, and version 2.5 is compatible with Java EE 5.0.
5. On the Summary page, click **Finish**.
6. Open the `web.xml` file in the overview editor.
7. Select the **Filters** tab and remove **JpsFilter**.

How to Enable JDBC Data Source for GlassFish

If you are using **ADF Business Components** in your application, you may need to check that the data source defined for the ADF Business Components as part of the application matches the data source defined in the GlassFish container.

You can configure the ADF application to use a JDBC Data Source in the application by editing the application's `bc4j.xcfg` file.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you configure JDBC. For more information, see Preparing the Application.

To configure the bc4j.xcfg file to enable JDBC data source:

1. In the Applications window, expand the model project and click the application module.
2. In the overview editor for the application module, choose **Configurations** and click **bc4j.xcfg**.
3. In the overview editor for `bc4j.xcfg`, select the **Source** tab.
4. Check that the `Custom JDBCDataSource` entry defined in the file matches the JDBC datasource defined for the GlassFish Server as described in Configuring GlassFish Server in *Administering Oracle ADF Applications*.

   For instance, the following example shows the `JDBCDataSource` property defined in the `bc4j.xcfg` file:

   ```
   <BC4JConfig version="11.1" xmlns="http://xmlns.oracle.com/bc4j/
   configuration">
      <AppModuleConfigBag ApplicationName="model.AppModule">
         <AppModuleConfig name="AppModuleLocal" jbo.project="model.Model"
            ApplicationName="model.AppModule" DeployPlatform="LOCAL">
            <Database jbo.TypeMapEntries="OracleApps"/>
            <Security AppModuleJndiName="model.AppModule"/>
            <Custom ns0:JDBCDataSource="jdbc/OracleDS"
              xmlns:ns0="http://xmlns.oracle.com/bc4j/configuration"/>
         </AppModuleConfig>
         <AppModuleConfig name="AppModuleShared" jbo.project="model.Model"
            ApplicationName="model.AppModule" DeployPlatform="LOCAL">
            <AM-Pooling jbo.ampool.maxpoolsize="1"
             jbo.ampool.isuseexclusive="false"/>
            <Database jbo.TypeMapEntries="OracleApps"/>
            <Security AppModuleJndiName="model.AppModule"/>
            <Custom ns0:JDBCDataSource="jdbc/OracleDS"
              xmlns:ns0="http://xmlns.oracle.com/bc4j/configuration"/>
         </AppModuleConfig>
      </AppModuleConfigBag>
   </BC4JConfig>
   ```

How to Change Initial Context Factory

If your application requires JNDI lookup, such as in EJB-based applications, you need to change the initial context factory class name to the GlassFish context factory.

To change the initial context factory:

1. In the Applications window, expand the model project and double-click the **DataControls.dcx** file
2. In the overview editor, click the **Source** tab.
3. Edit the `initial-context-factory` entry to be `initial-context-factory="com.sun.enterprise.naming.SerialInitContextFactory"`

Deploying the Application

Once the target environment is set up and the ADF application is prepared for deployment, the final step is to deploy the application to the target environment.

You can use JDeveloper to deploy ADF applications directly to the GlassFish Server or you can create an archive file and use other tools to deploy to the application server.

> **Note:**
>
> Before you begin to deploy applications that use **Oracle ADF** to the standalone application server, you need to prepare the GlassFish Server environment by performing tasks such as installing the ADF runtime libraries and setting configuration values. See Configuring GlassFish Server in *Administering Oracle ADF Applications*.

Table H-1 describes some common deployment techniques that you can use during the application development and deployment cycle. The deployment techniques are listed in order from deploying on development environments to deploying on production environments. It is likely that in the production environment, the system administrators deploy applications by using the GlassFish Administration Console or scripts.

Table H-1 Deployment Techniques for Development or Production Environments

Deployment Technique	Environment	When to Use
Run directly from JDeveloper	Test or Development	When you are developing your application. You want deployment to be quick because you will be repeating the editing and deploying process many times. JDeveloper contains Integrated WebLogic Server, on which you can run and test your application. Note that you are testing your application in an integrated WebLogic Server and not in an Integrated GlassFish Server. After testing on the Integrated WebLogic Server, you can test further by deploying to the standalone GlassFish Server.
Use JDeveloper to directly deploy to the target GlassFish Server	Test or Development	When you are ready to deploy and test your application on an application server in a test environment. You can also use the test environment to develop your deployment scripts, for example, using Ant.
Use JDeveloper to deploy to an EAR file, then use the target GlassFish Server's tools for deployment	Test or Development	When you are ready to deploy and test your application on an application server in a test environment. As an alternative to deploying directly from JDeveloper, you can deploy to an EAR file and then use other tools to deploy to the application server. You can also use the test environment to develop your deployment scripts, for example, using Ant.

Table H-1 (Cont.) Deployment Techniques for Development or Production Environments

Deployment Technique	Environment	When to Use
Use GlassFish `asadmin` commands or GlassFish Administration Console.	Production	When your application is in a test and production environment. In production environments, system administrators usually use GlassFish `asadmin` commands or the GlassFish Administration Console.

How to Deploy to the Application Server from JDeveloper

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you deploy to the application server from JDeveloper. For more information, see Preparing the Application.

You will need to complete this task:

Create an application-level deployment profile that deploys to an EAR file.

To deploy to the target application server from JDeveloper:

1. In the Applications window, right-click the application and choose **Deploy >** ***deployment profile***.

   Make sure you choose a GlassFish deployment profile.

2. In the Deploy wizard, on the Deployment Action page, select **Deploy to Application Server** and click **Next**.
3. On the Select Server page, select the application server connection, and click **Next**.
4. Click **Finish**.

   During deployment, you can see the process steps displayed in the deployment Log window. You can inspect the contents of the modules (archives or exploded EAR) being created by clicking on the links that are provided in the log window. The archive or exploded EAR file will open in the appropriate editor or directory window for inspection.

   > **Note:**
   >
   > If you are deploying a Java EE application, click the application menu next to the Java EE application in the Applications window.

   For more information on creating application server connections, see How to Create a Connection to the Target Application Server.

What You May Need to Know About Deploying from JDeveloper

When you deploy an application using the EAR deployment profile created for GlassFish, JDeveloper:

- Inserts the required ADF Model, ADF Controller, and ADF View library JARs into the EAR file.
- Inserts `adf-share-glassfish.jar` into the WAR.
- Inserts an entry for the GlassFish application lifecycle listener `ADFGlassFishAppLifeCycleListener` into the `web.xml` file.
- Removes the other listener entries, `ADFConnectionLifeCycleCallBack`, `ADFConfigLifeCycleCallBack`, and `BC4JConfigLifeCycleCallBack`, from the `web.xml` files.

How to Create an EAR File for Deployment

You can also use the deployment profile to create an archive file (EAR file). You can then deploy the archive file using GlassFish Administration Console.

Although an ADF application is encapsulated in an EAR file (which usually includes WAR components), it may have parts that are not deployed with the EAR.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create an EAR file for deployment. For more information, see Preparing the Application.

You will need to complete this task:

Create an application-level deployment profile that deploys to an EAR file.

To create an EAR archive file:

- In the Applications window, right-click the application containing the deployment profile, and choose **Deploy > *deployment profile* > to EAR file**.

  If an EAR file is deployed at the application level, and it has dependencies on a JAR file in the data model project and dependencies on a WAR file in the user interface project, then the files will be located in the following directories by default:

  - *ApplicationDirectory*/deploy/*EARdeploymentprofile*.EAR
  - *ApplicationDirectory*/*ModelProject*/deploy/*JARdeploymentprofile*.JAR
  - *ApplicationDirectory*/*ViewControllerProject*/deploy/*WARdeploymentprofile*.WAR

> **Tip:**
>
> Choose **View** >**Log** to see messages generated during creation of the archive file.

What You May Need to Know About ADF Libraries

An ADF Library is a JAR file that contains JAR services registered for ADF components such as ADF task flows, pages, or application modules. If you want the ADF components in a project to be reusable, you create an ADF Library deployment profile for the project and then create an ADF Library JAR based on that profile.

An application or project can consume the ADF Library JAR when you add it using the Resources window or manually by adding it to the library classpath. When the ADF Library JAR is added to a project, it will be included in the project's WAR file if the **Deployed by Default** option is selected.

For more information, see Reusing Application Components.

What You May Need to Know About EAR Files and Packaging

When you package an ADF application into an EAR file, it can contain the following:

- WAR files: Each web-based view controller project should be packaged into a WAR file.
- ADF Library JAR files: If the application consumes ADF Library JARs, these JAR files may be packaged within the EAR.
- Other JAR files: The application may have other dependent JAR files that are required. They can be packaged within the EAR.

How to Deploy to the Application Server using asadmin Commands

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you deploy to the application server from JDeveloper. For more information, see Preparing the Application.

To deploy to the target application server using asadmin command:

1. Start an `asadmin` shell for GlassFish Server.
2. Invoke the `deploy` command.

   For instance, the following command deploys `application1.ear` to the GlassFish Server.

   ```
   deploy \--target=server \--force /path/application1.ear
   ```

How to Deploy the Application Using Scripts and Ant

You can deploy the application using commands and automate the process by putting those commands in scripts. The `ojdeploy` command can be used to deploy an application without JDeveloper. You can also use Ant scripts to deploy the application. JDeveloper has a feature to help you build Ant scripts. Depending on your requirements, you may be able to integrate regular scripts with Ant scripts.

For more information about commands, scripts, and Ant, see Deploying Using Scripting Commands and Deploying Using Scripts and Ant in *Administering Oracle ADF Applications*.

Testing the Application and Verifying Deployment

In order to ensure the deployment of the ADF application has been successful, you should test-run the application.

After you deploy the application, you can test it from the application server. To test-run your ADF application, open a browser window and enter a URL:

- For non-Faces pages: `http://<host>:port/<context root>/<page>`
- For Faces pages: `http://<host>:port/<context root>/faces/<view_id>`

  where `<view_id>` is the view ID of the ADF task flow view activity.

> **Tip:**
>
> The context root for an application is specified in the user interface project settings by default as *`ApplicationName`*/*`ProjectName`*/`context-root`. You can shorten this name by specifying a name that is unique across the target application server. Right-click the user interface project, and choose **Project Properties**. In the Project Properties dialog, select **Java EE Application** and enter a unique name for the context root.

> **Note:**
>
> `/faces` has to be in the URL for Faces pages. This is because JDeveloper configures your `web.xml` file to use the URL pattern of `/faces` in order to be associated with the Faces Servlet. The Faces Servlet does its per-request processing, strips out `/faces` part in the URL, then forwards the URL to the JSP. If you do not include the `/faces` in the URL, then the Faces Servlet is not engaged (since the URL pattern doesn't match). Your JSP is run without the necessary JSF per-request processing.