Oracle® Fusion Middleware

Developing Applications with Oracle ADF Data Controls

12*c* (12.1.3)

E41270-01

May 2014

Documentation for Oracle Application Development Framework (Oracle ADF) developers that describes how to create and configure data controls for Enterprise JavaBeans (EJB) and plain Java objects and for SOAP and REST web services.

ORACLE

Oracle Fusion Middleware Developing Applications with Oracle ADF Data Controls12*c* (12.1.3)

E41270-01

Primary Author: Landon Ott

Contributing Author: Patrick Keegan

Contributors: Jon Wetherbee, Jim Pham, Vinay Agarwal

Contents

4 Creating and Configuring Bean Data Controls

5 Exposing Web Services Using the ADF Model Layer

6 Exposing URL Services Using the ADF Model Layer

7 Adding Business Logic to Data Controls

A Data Control Feature Comparison

Preface

Welcome to *Developing Applications with Oracle ADF Data Controls*.

Audience

This document is intended for developers who use ADF data controls to abstract business services such as EJBs, plain Java classes, and web services to simplify data binding between those services and UI components.

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 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 Fusion Web Applications with Oracle Application Development Framework*
- *Developing Web User Interfaces with Oracle ADF Faces*
- *Developing Swing Applications with Oracle Application Development Framework*
- *Developing Applications with Oracle JDeveloper*
- *Developing ADF Skins with Oracle ADF Skin Editor*
- *Administering Oracle ADF Applications*
- *Developing Applications with Oracle ADF Desktop Integration*
- *Developing Oracle ADF Mobile Browser Applications*
- *Tuning Performance*

- *Understanding Oracle Web Services Manager*
- *Securing Web Services and Managing Policies with Oracle Web Services Manager*
- *High Availability Guide*
- *Installing Oracle JDeveloper*
- *Installing Oracle ADF Skin Editor*
- *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*
- *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 ADF Business Components: interMedia Domains*
- *Java API Reference for Oracle Metadata Service (MDS)*
- *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 associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

What's New in This Guide

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.

New and Changed Features for Release 12c (12.1.3)

Oracle Fusion Middleware Release 12c (12.1.3) of Oracle JDeveloper and Oracle Application Development Framework (Oracle ADF) includes the following new and changed development features, which are described in this guide:

- New No Paging option in the wizards for creating EJB and bean data controls. See Section 3.4, "Paginated Fetching of Data in EJB Data Controls."
- Ability to use the `AccessMode` annotation on either the session bean class or the interface (remote or local) on which the data control is based. See Section 3.4.4, "How to Specify Access Mode for Individual Objects in the Data Control."
- New capability to add UI hints to a data control by adding annotations directly to the bean classes (meaning that separate XML data control structure files are not needed to define the UI hints). See Section 3.5, "Providing UI Hints for Attributes Using Annotations."
- Ability to specify a configuration class to hold the metadata for custom bean data controls. See Section 4.9, "Creating Custom Bean Data Controls."

For other changes made to Oracle JDeveloper and Oracle Application Development Framework (Oracle ADF) for this release, see the What's New page on the Oracle Technology Network at `http://www.oracle.com/technetwork/developer-tools/jdev/documentation/index.html`.

Other Significant Changes in this Document for Release 12c (12.1.3)

For Release 12c (12.1.3), this document has been changed in other ways. Following are the sections that have been added or changed.

- Updated the section on bean data control annotations to reflect new UI hint annotations and to clarify that `@Property` annotations are for custom properties. See Section 4.8, "Using Annotations to Declare Metadata for Bean Data Controls."

1 Introduction to ADF Model

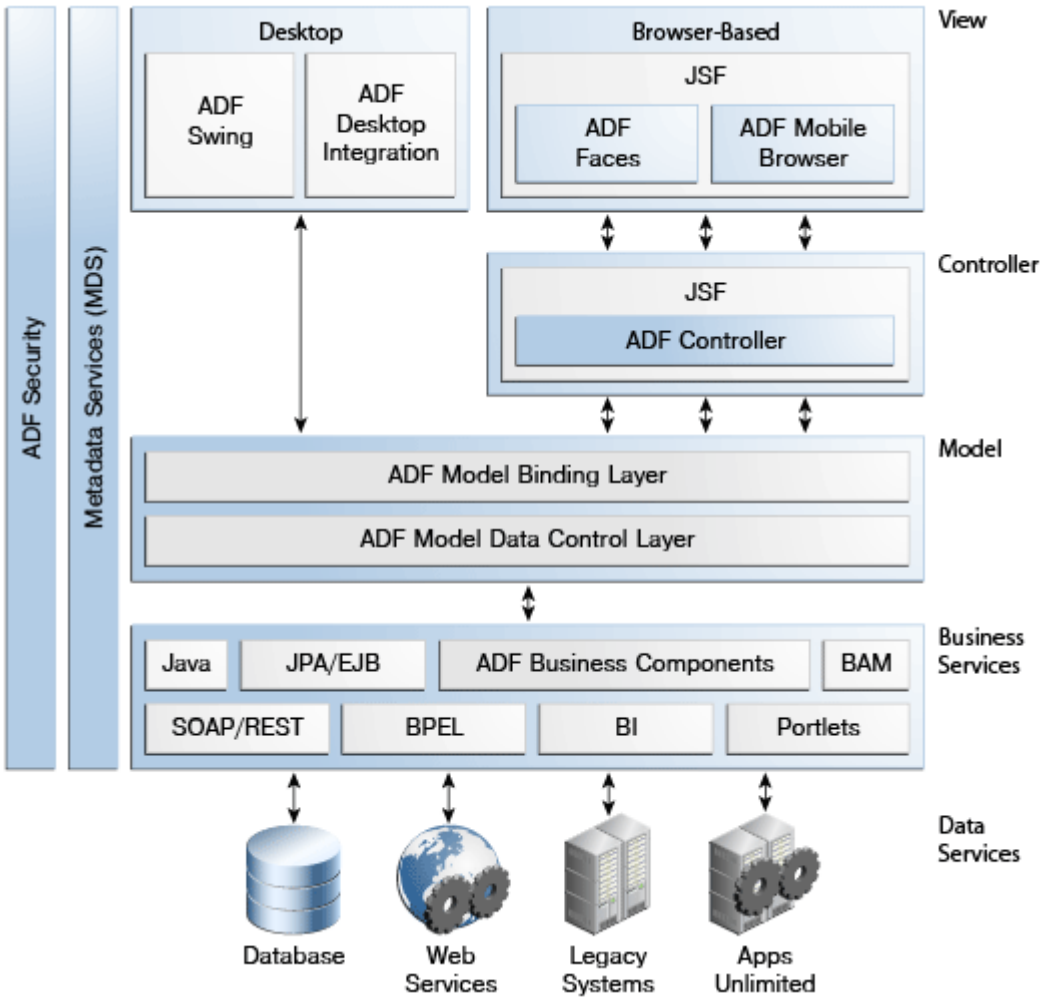
This chapter provides a brief overview of data controls and binding data from business services to user interfaces in Oracle ADF applications. It includes information on types of data controls and then using those data controls to create databound UI components.

This chapter includes the following sections:

1.1 About ADF Model

ADF Model is a declarative data binding facility that enables a unified approach to binding user interfaces to business services without requiring code. ADF Model implements two concepts that enable the decoupling of the user interface technology from the business service implementation: **data controls** and declarative **ADF bindings**. Figure 1–1 shows how the elements of ADF Model fit within an application architecture.

Figure 1–1 ADF Architecture with ADF Model



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. These operations and collections are exposed as data control objects that developers and UI designers can use to create databound UI components, largely without having to consider the type of underlying business service.

Declarative bindings abstract the details of accessing data from data collections in a data control and of invoking its operations. There are three basic categories of declarative binding objects:

- 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.
- 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.

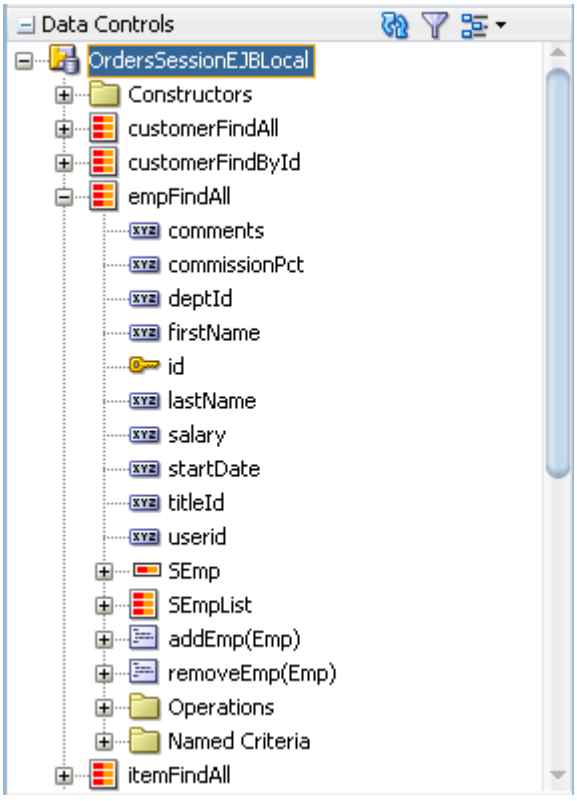
Executable bindings also include bindings that allow searching and nesting a series of pages within another page, as well as bindings that cause operations to occur immediately.

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.

To use the ADF Model layer to bind data, you need to create a data control for your services. The data control will then appear as a tree hierarchy in the Data Controls panel where each subnode in the tree represents an element such as a collection, operation, method or attribute. You then create databound components by dragging and dropping those subnodes onto the visual editor for a web page or other user interface component. When you drag a subnode from a data control to a page, JDeveloper automatically creates the metadata that describes the bindings from the page to the services.

For example, in an application that uses an EJB session facade, developers can create data controls for the facade. Developers can then use the representation of the data control displayed in JDeveloper's Data Controls panel (as shown in Figure 1–2) to create UI components that are automatically bound to the session facade.

Figure 1–2 Data Controls Panel



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*.

In a JSF application, the binding container is accessible during each page request using the EL expression `#{bindings}`. This expression always evaluates to the binding container for the current page.

Example 1–1 shows the code used for binding a checkbox in a form to the `orderFilled` attribute of the `OrdersSessionEJBLocal` data control.

Example 1–1 Binding Code for a Checkbox in a JSF Web Page

```
<af:selectBooleanCheckbox value="#{bindings.orderFilled.inputValue}"
                  label="#{bindings.orderFilled.label}"
                  shortDesc="#{bindings.orderFilled.hints.tooltip}" id="sbc1"/>
```

1.2 Data Control Types

There are various data control implementations and they can be divided into these categories:

- ADF Business Components application modules

  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.

  For a complete guide to using ADF Business Components application modules, see "Implementing Business Services with Application Modules" in *Developing Fusion Web Applications with Oracle Application Development Framework*.

- Data controls that both extend the adapter data control framework ("adapter data controls")

  These include the data controls for EJBs, beans, web services (SOAP-based and RESTful), and URLs, which are covered in this guide.

- Custom data controls based on the bean data control type

- Custom data controls based on the adapter data control framework that do not extend the bean data control

- Custom data controls that do not extend the adapter data control framework

  This category primarily consists of data controls that were developed before the adapter framework was developed.

This guide will primarily focus on creating and using adapter data controls, though much of the information also applies to ADF Business Components data controls.

1.3 Data Controls Not Covered By This Guide

The following are some of the types of data controls that are not covered in this guide:

- Placeholder data controls, which are empty data controls used to help design databound pages for users who do not yet have concrete business services to work with. After you design databound components with the placeholder data control and the business services have been provided, you can rebind the components to those business services. For more information, see "Designing a Page Using Placeholder Data Controls" in *Developing Fusion Web Applications with Oracle Application Development Framework*.

- JMX data controls, which enable you to create a data control based on Java Management Extension (JMX) MBeans.

- JavaBean data controls, which are not to be confused with bean data control. The JavaBean data control is a data control type that was developed before the adapter data control framework. It is missing many of the key features that have since been built into bean data controls, such as support for queryable collections.
- Other data controls within the broader Fusion Middleware stack, such as BAM data controls.

1.4 What You May Need to Know About Non-Adapter Framework Data Controls

If you are working with legacy data controls, you might want to consider migrating to new data controls based on the adapter data control framework. This framework incorporates a number of useful features, such as validation rules and list of value objects. You can use the adapter framework to create a new data control type from scratch, or you can extend one of the existing data control types, such as the bean data control type.

2

Using ADF Data Controls

This chapter describes how to create data controls to abstract business services and use the Data Controls panel to create databound pages. It also includes information on the way declarative binding is specified at design time and implemented at runtime.

This chapter contains the following sections:

2.1 Core Development Steps For Data Control Applications

At a high level, the declarative development process for an application that contains **data controls** usually involves the following core steps:

- Creating an application workspace in JDeveloper: 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. For more information, see the "Creating Applications and Projects" section of *Developing Applications with Oracle JDeveloper*.
- Creating the business services: These can be ADF Business Components, EJB session beans, POJOs, web services, or other services. For more information on developing these services in JDeveloper, see *Developing Applications with Oracle JDeveloper*.
- Creating data controls for your services: Once you have created your business services, you create the data controls that use metadata interfaces to abstract the implementation of those services and describe their operations and data collections, including information about the properties, methods, and types involved. These data controls are displayed in the Data Controls panel and can be dragged to pages to create databound UI components. For more information, see Section 2.2, "Exposing Business Services with Data Controls."
- Adding declarative metadata to your data controls: You can augment your data controls with UI hints, validation rules, criteria for use in search forms, and other features. For more information, see Chapter 7, "Adding Business Logic to Data Controls."
- Implementing the user interface: JDeveloper's Data Controls panel contains a representation of the services for your application. You can drag an object from the Data Controls panel onto a page and select 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 all the page code for you.

For information about creating databound web pages, see "Creating a Databound Web User Interface" in *Developing Fusion Web Applications with Oracle Application Development Framework*.

As part of creating the user interface, you are likely to also want to define task flows to organize the user's workflow within the application. For more information, see "Creating ADF Task Flows" in *Developing Fusion Web Applications with Oracle Application Development Framework*.

For more detailed information on the UI components themselves, see the "Implementing the User Interface with JSF" section in *Developing Fusion Web Applications with Oracle Application Development Framework*.

- Deploying the application: You use wizards and editors to create and edit deployment descriptors, JAR files, and application server connections. For more information, see "Deploying Fusion Web Applications" in *Developing Fusion Web Applications with Oracle Application Development Framework*.

2.2 Exposing Business Services with Data Controls

Once you have your application's services in place, you can use JDeveloper to create data controls that provide the information needed to declaratively bind UI components to those services.

For example, in a Java EE application, you normally create entity beans that represent tables in a database and then create a session facade over all the EJBs. This facade provides a unified interface to the underlying entities. In an Oracle ADF application, you can create a data control for the session bean, and that data control will contain representation of all the EJBs under the session bean.

You generate data controls with the Create Data Control menu item. Data controls consist of one or more XML metadata files that define the capabilities of the services that the bindings can work with at runtime. The data controls work in conjunction with the underlying services.

2.2.1 How to Create ADF Data Controls

You create adapter-based data controls from within the Applications window of JDeveloper.

Note: For applications based on ADF Business Components, the data controls are created automatically when you create an application module. For more information, see "Implementing Business Services with Application Modules" in *Developing Fusion Web Applications with Oracle Application Development Framework*.

Before you begin:

It may be helpful to have a general understanding of using data controls. For more information, see Section 2.2, "Exposing Business Services with Data Controls."

You will need to complete this task:

Create an application workspace and add the business services on which you want to base your data control. For information on creating an application workspace,

see "Creating Applications and Projects" in *Developing Applications with Oracle JDeveloper*.

To create a data control:

1. Right-click the top-level node for the data model project in the application workspace and choose **New** and then **From Gallery**.
2. In the New Gallery, expand **Business Tier**, select **Data Controls**, select the type of data control that you want to create, and click **OK**.
3. Complete the remaining steps of the wizard.

   For information on creating adapter-based data controls, see subsequent chapters of this guide for the different types of data controls.

> **Note:** In some cases, you can create a data control by right-clicking the class or object on which the data control will be based and choosing Create Data Control.

2.2.2 What Happens in Your Project When You Create a Data Control

When you create a data control, JDeveloper creates the data control definition file (`DataControls.dcx`), opens the file in the overview editor, and displays the file's hierarchy in the Data Controls panel. This file enables the data control to work directly with the services and the bindings.

You can see the code from the corresponding XML file by clicking the Source tab in the editor window.

2.2.2.1 DataControls.dcx Overview Editor

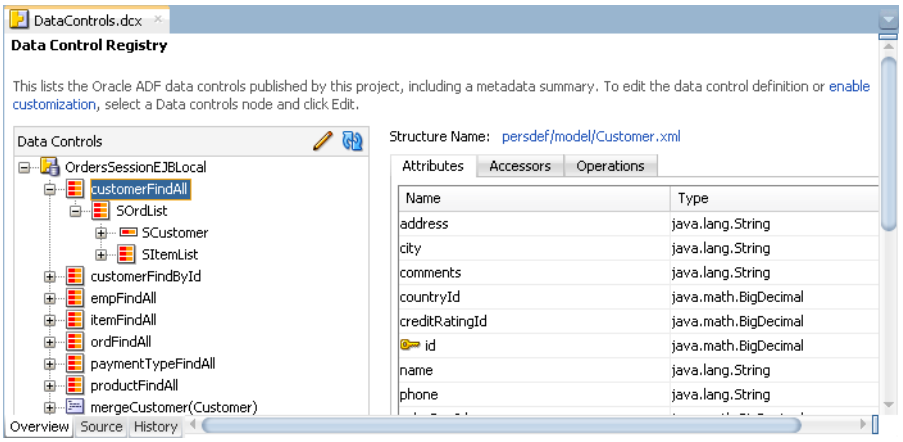
The overview editor for the `DataControls.dcx` file provides a view of the hierarchies of data control objects and exposed methods of your data model.

See Table 2–1 for a description of the icons that are used in the overview editor and Data Controls panel.

You can change the settings for a data control object by selecting the object and clicking the **Edit** icon. For more information about editing a data control, see Section 7.2, "Configuring Data Controls."

Figure 2–1 shows the overview editor for an EJB data control.

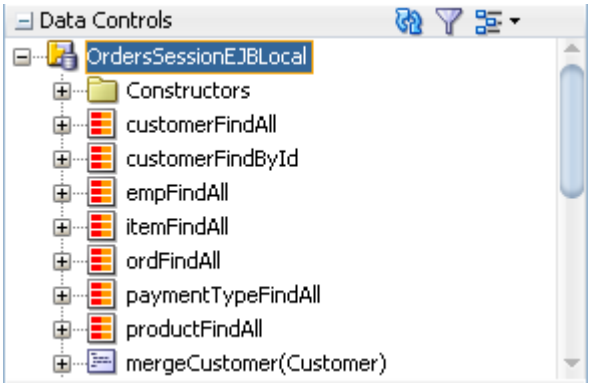
Figure 2–1 DataControls.dcx File in the Overview Editor



2.2.2.2 Data Controls Panel

The Data Controls panel serves as a palette, from which you can create databound UI components by dragging nodes from the Data Controls panel to the design editor for a web page. The Data Controls panel appears in the Applications window once you have created a data control. Figure 2–2 shows the Data Controls panel for an EJB data control.

Figure 2–2 Data Controls Panel



2.2.3 Display of Business Services in the Data Controls Panel

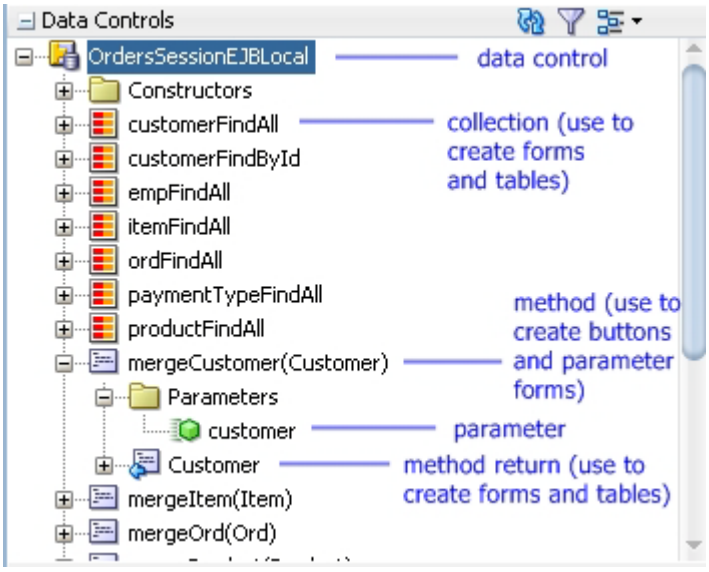
The Data Controls panel lists all the data controls that have been created for the application's business services and exposes the hierarchies of collections (row sets of data objects), attributes, methods, and built-in operations that are available for binding to UI components.

For example, Figure 2–3 shows the Data Controls panel with the `OrdersSessionEJBLocal` data control. The collection nodes (such as `customerFindAll`, `customerFindById`, `empFindAll`) represent data collections that are returned by query methods (in this case, getter methods in the session facade). These collection objects can be dropped on to pages to create UI components such as forms and tables.

Other service methods (in the case of JPA-based data controls, methods that are not prefixed with `get`) are represented by method icons. These objects can be dropped on a form as a command button or link. If a method accepts arguments, those arguments

appear in a Parameters node as parameters nested inside the method's node. Objects that are returned by the methods appear as well.

Figure 2–3 Data Controls Panel Main Nodes

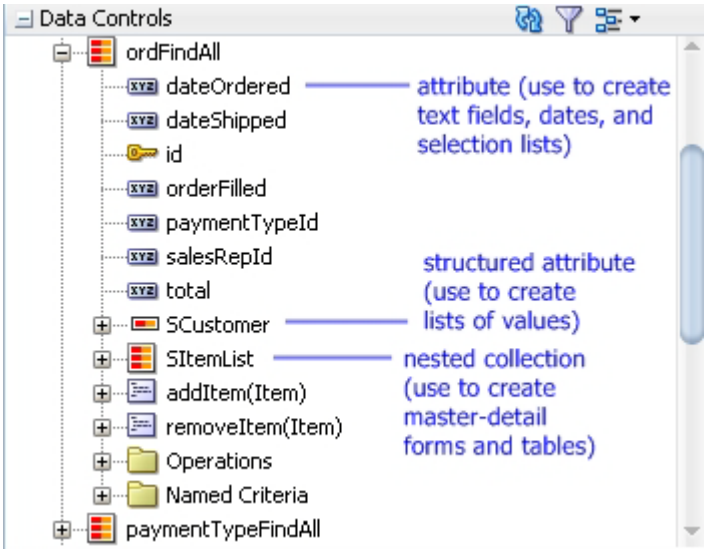


Tip: If the Data Controls panel is not visible, see "How to Open the Data Controls Panel" in *Developing Fusion Web Applications with Oracle Application Development Framework*.

Each returned object displays any attributes, methods, and nested collections that were defined on the associated object. Figure 2–4 shows the attributes and methods defined on the `Item` bean that is returned by the `itemFindAll` collection.

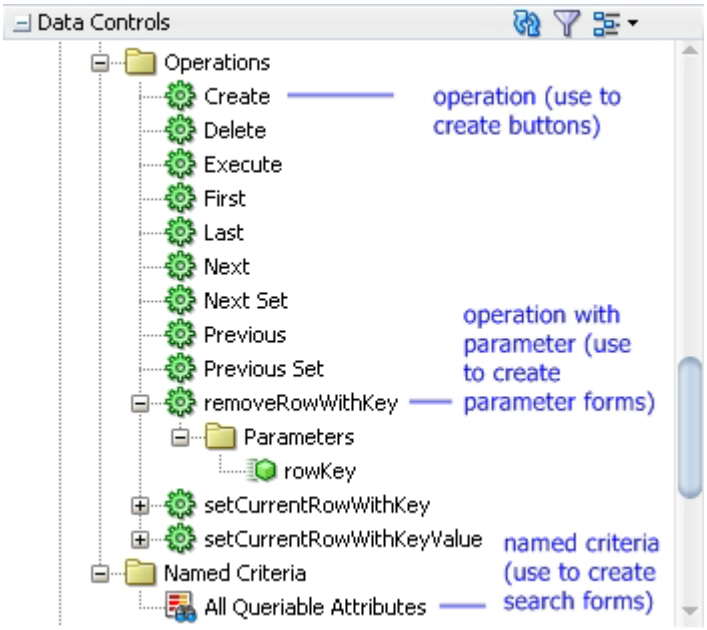
Note: Whenever changes are made to the services on which a data control is based, the data control incorporates those changes. If changes to the services are not immediately reflected in the Data Controls panel, you can refresh the panel manually. For more information, see "How to Refresh the Data Controls Panel" in *Developing Fusion Web Applications with Oracle Application Development Framework*.

Figure 2–4 Child Nodes to Returned Collections



Depending on data control type, various built-in operations are exposed. For some data controls, declarative named criteria are also available, which you can use to create search forms.Figure 2–5 shows operations and the default named criteria for an EJB data control.

Figure 2–5 Data Control Panel Operations and Named Criteria



2.2.4 Data Control Built-in Operations

The data control framework defines a standard set of operations for data controls. These operations are implemented using functionality of the underlying business service. At runtime, when one of these data collection operations is invoked by name by the data binding layer, the data control delegates the call to an appropriate service

method to handle the built-in functionality. For example, in EJB and bean data controls, the `Next` operation relies on the bean collection's iterator.

Most of the built-in operations affect the current row. However, the `execute` operation refreshes the data control itself, and the `commit` and `rollback` operations affect all changes made within the boundaries of a transaction.

The operations available vary by data control type and the functionality of the underlying business service. Here is the full list of built-in operations:

- `Create`: Creates a new row that becomes the current row. For JPA-based data controls, this new row is also added to the row set.
- `CreateInsert`: Creates a new row that becomes the current row and inserts it into the row set. Available only for ADF Business Components application modules.
- `Create With Parameters`: Uses named parameters to create a new row that becomes the current row and inserts it into the row set. Available only for ADF Business Components application modules.
- `Delete`: Deletes the current row.
- `Execute`: Refreshes the data collection by executing or reexecuting the accessor method.

  `ExecuteWithParams`: Refreshes the data collection by first assigning new values to variables that passed as parameters, then executing or reexecuting the associated query. This operation is only available for data control collection objects that are based on parameterized queries.
- `First`: Sets the first row in the row set to be the current row.
- `Last`: Sets the last row in the row set to be the current row.
- `Next`: Sets the next row in the row set to be the current row.
- `Next Set`: Navigates forward one full set of rows.
- `Previous`: Sets the previous row in the row set to be the current row.
- `Previous Set`: Navigates backward one full set of rows.
- `removeRowWithKey`: Tries to find a row using the serialized string representation of the row key passed as a parameter. If found, the row is removed.
- `setCurrentRowWithKey`: Tries to find a row using the serialized string representation of the row key passed as a parameter. If found, that row becomes the current row.
- `setCurrentRowWithKeyValue`: Tries to find a row using the primary key attribute value passed as a parameter. If found, that row becomes the current row.
- `commit`: Persists to the database all changes that are made in the current transaction.
- `rollback`: Reverts all changes made within the context of the current transaction.

2.3 Creating Databound UI Components from the Data Controls Panel

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 2–1 describes what each icon represents, where it appears in the Data Controls panel hierarchy, and what components it can be used to create.

Table 2–1 Data Controls Panel Icons and Object Hierarchy

Icon	Name	Description	Used to Create...
	Data Control	Represents a data control.	Serves as a container for the other objects and is not used to create anything.
	Collection	Represents a named data collection returned by an accessor method or operation.	Forms, tables, graphs, trees, range navigation components, and master-detail components. For more information about using collections on a data control to create forms, see "Creating a Basic Databound Page" in *Developing Fusion Web Applications with Oracle Application Development Framework*. For more information about using collections to create tables, see "Creating ADF Databound Tables" in *Developing Fusion Web Applications with Oracle Application Development Framework*. For more information about using master-detail relationships to create UI components, see "Displaying Master-Detail Data" in *Developing Fusion Web Applications with Oracle Application Development Framework*. For information about creating graphs, charts, and other visualization UI components, see "Creating Databound Chart and Gauge Components" in *Developing Fusion Web Applications with Oracle Application Development Framework*.
	Structured Attribute	Represents a returned object that is neither a Java primitive type (represented as an attribute) nor a collection of any type.	Label, text field, date, list of values, and selection list components.
	Attribute	Represents a discrete data element in an object (for example, an attribute in a row).	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" in *Developing Fusion Web Applications with Oracle Application Development Framework*. For information about creating lists, see "Creating Databound Selection Lists and Shuttles" in *Developing Fusion Web Applications with Oracle Application Development Framework*.
	Key Attribute	Represents an object attribute that has been declared as a primary key attribute, either in data control structure file or in the business service itself.	Label, text field, date, list of values, and selection list components.

Table 2–1 (Cont.) Data Controls Panel Icons and Object Hierarchy

Icon	Name	Description	Used to Create...
	Method	Represents a method or operation in the data control or one of its exposed structures that may accept parameters, perform some business logic and optionally return single value, a structure, or a collection.	Command components. For methods that accept parameters: command components and parameterized forms. For information about creating command components from methods, see "Using Command Components to Invoke Functionality in the View Layer" in *Developing Fusion Web Applications with Oracle Application Development Framework*. For information about creating parameterized forms, see "Using Parameters to Create a Form" in *Developing Fusion Web Applications with Oracle Application Development Framework*.
	Method Return	Represents an object that is returned by a method or other operation. 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, or operations that can be performed on the parent collection.	For single values: text fields and selection lists. For collections: forms, tables, trees, and range navigation components. 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 more information about executables, see "Executable Binding Objects Defined in the Page Definition File" in *Developing Fusion Web Applications with Oracle Application Development Framework*.
	Operation	Represents a built-in data control operation that performs actions on the parent object.	UI command components, such as buttons, links, and menus. For more information, see "Creating Command Components Using Data Control Operations" and "Creating a Form to Edit an Existing Record" in *Developing Fusion Web Applications with Oracle Application Development Framework*.
	Parameter	Represents a parameter value that is declared by the method or operation under which it appears.	Label, text, and selection list components.
	Named criteria	Represents a metadata-based query from which you can create a user search form. Named criteria are available for EJB and (JPA-based) bean data controls. You can create custom view criteria and add them to the Data Controls panel. See Section 7.6, "Filtering Result Sets with Named Criteria."	Search forms. For information on creating search forms, see "Creating ADF Databound Search Forms" in *Developing Fusion Web Applications with Oracle Application Development Framework*.

2.3.1 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 Section 2.3, "Creating Databound UI Components from the Data Controls Panel."

You will need to complete these tasks:

- Create a data control as described in Section 2.2.1, "How to Create ADF Data Controls."
- Create a web page as described in "Creating a Web Page" in *Developing Fusion Web Applications with Oracle Application Development Framework*.

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 2–1.

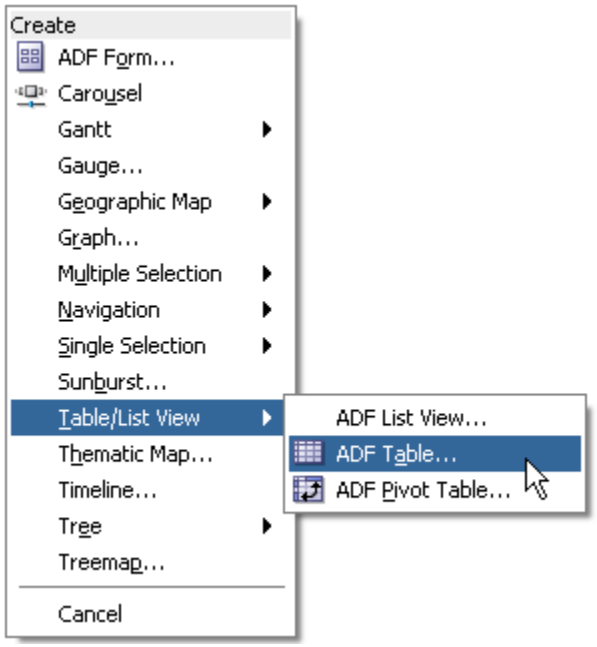
   Tip: If you need to drop an operation or method onto a method activity in a task flow, you can simply drag and drop it onto the activity in the diagram.

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 2–6 shows the context menu displayed when a data collection from the Data Controls panel is dropped on a page.

Figure 2–6 Dropping Component From Data Controls Panel



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 **ADF Table** from the context menu, the Edit Table Columns dialog launches.

The UI components selected by default are determined first by any UI hints set on the corresponding business object. If no UI 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 and choose **ADF Table** from the context menu, a table appears in the visual editor, as shown in Figure 2–7.

Figure 2–7 Databound UI Component: ADF Table

Product Id	Supplier Id	Category Id	Name
#{...ProductId}	#{...SupplierId}	#{...CategoryId}	#{...ProductName}
#{...ProductId}	#{...SupplierId}	#{...CategoryId}	#{...ProductName}
#{...ProductId}	#{...SupplierId}	#{...CategoryId}	#{...ProductName}

By default, the UI components created when you use the Data Controls are bound to attributes in the ADF data control and may have built-in features, such as:

- Databound labels
- Tooltips
- Formatting
- Basic navigation buttons

- Validation, if validation rules are attached to a particular attribute.

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" in *Developing Fusion Web Applications with Oracle Application Development Framework*.

> **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).

2.3.2 What Happens When You Use the Data Controls Panel

When a web application is built using the Data Controls panel, JDeveloper does the following:

- 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.

  A `DataBindings.cpx` files defines 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" in *Developing Fusion Web Applications with Oracle Application Development Framework*. The `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" in *Developing Fusion Web Applications with Oracle Application Development Framework*.

- 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" in *Developing Fusion Web Applications with Oracle Application Development Framework*.

- Adds several libraries to the view project, including the following:
  - ADF Faces Databinding Runtime
  - ADF Model Runtime
  - MDS Runtime

- 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. For more information about the page definition file, see "Working with Page Definition Files" in *Developing Fusion Web Applications with Oracle Application Development Framework*.

Tip: The current binding container is also available from `AdfContext` for programmatic access.

- Configures the page definition file, which includes adding definitions of the binding objects referenced by the page.
- Adds the given component to the 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" in *Developing Fusion Web Applications with Oracle Application Development Framework*.
- Adds all the libraries, files, and configuration elements required by the UI components. For more information on the artifacts required for ADF Faces databound components, see the "ADF Faces Configuration" appendix in *Developing Web User Interfaces with Oracle ADF Faces*.

2.3.3 What You May Need to Know About Iterator Result Caching

When a data control modifies a collection, the data control must instantiate a new instance of the collection in order for the ADF Model layer to understand that it has been modified. In other words, although some action in the client may change the collection, that change will not be reflected in the UI unless a new instance of the collection is created. However, for performance reasons, accessor and method iterators cache their results set (by default, the `cacheResults` attribute on the iterator is set to `true`). This setting means that the iterator is refreshed and a new instance of the collection is created only when the page is first rendered. The iterator is not refreshed when the page is revisited, for example, if the page is refreshed using partial page rendering, or if the user navigates back to the page.

For example, say you want to allow sorting on a table on your page. Because you want the page to refresh after the sort, you add code to the listener for the sort event that will refresh the table using partial page rendering (for more information, see the "Rerendering Partial Page Content" chapter of *Developing Web User Interfaces with Oracle ADF Faces*). Because the instance of the collection for the table has already been instantiated and is cached, the accessor iterator will not reexecute, which means that a new instance of the collection with the new sort order will not be created, so the sort order on the page will remain the same.

To work around this issue, you can either configure the iterator so that it does not cache the results, or you can place a button on the page that can be used to reexecute the iterator when the page is refreshed.

Note: If your page uses the navigation operations to navigate through the collection, do not set `CacheResults` to false, as that will cause navigation to stop working. You must use a button to reexecute the iterator. For more information about using the navigation operations, see "Incorporating Range Navigation into Forms" in *Developing Fusion Web Applications with Oracle Application Development Framework*.

2.3.3.1 Setting an Iterator to Not Cache Its Result Set

To set an iterator to not cache its result set:

1. Open the page definition file, and in the Structure window, select the iterator whose results should not be cached.
2. In the Properties window, expand the **Advanced** section and select **false** from the **CacheResults** dropdown list.

2.3.3.2 Using a Button to Reexecute the Iterator

To use a button to reexecute the iterator:

1. In the ADF Faces page of the Components window, expand the General Controls panel, and drag a **Button** onto the page.
2. In the Structure window, right click the button and choose **Bind to ADF Control**.
3. In the Bind to ADF Control dialog, expand the collection associated with the iterator to reexecute, expand that collection's Operations node, and select **Execute**.

3

Creating and Configuring EJB Data Controls

This chapter describes how to create data controls for EJB session beans and also includes information on paging and failover support.

This chapter includes the following sections:

3.1 About EJB Data Controls

Data controls are an abstraction provided by ADF Model than enable you to work with data and business services in a declarative manner and easily create UI components. For more general information, see Section 1.1, "About ADF Model."

You can create data controls for EJB session beans to simplify the creation of web applications that rely on object model data, such as accessing data from a database. In addition to the features that are common for all adapter data controls (such as declarative UI hints and validation rules), EJB data controls have the following features built in:

- Range paging in order to improve performance of queries to large data sets
- Declarative named criteria to simplify creation of search-by-example forms
- Transactional operations (based on transactional business methods in the session bean)
- Failover support (based on failover methods that you implement)

3.1.1 EJB Data Control Use Cases and Examples

You can use EJB data controls to do the following kinds of things:

- Create highly functional web pages that are bound to an EJB session facade, without manually writing any binding code.
- Create an application that integrates existing EJB business services with ADF features such as ADF Model data binding, ADF Faces, and ADF task flows.

- Take advantage of UI hints, validation rules, and other declarative metadata to provide consistent application of prompts, tooltips, format masks, and error messages throughout the application. For more information, see Chapter 7, "Adding Business Logic to Data Controls."
- Take advantage of ADF Model features to declaratively add query-by-example forms and list-of-value (LOV) components to pages. For more information on creating LOV components, see Section 7.7, "Creating List of Values Objects."
- Enable MDS customization on the application, which allows customers to customize an application without modifying the source code.

3.1.2 Additional Functionality for EJB Data Controls

You may find it helpful to understand other ADF and JDeveloper features before you implement your data controls. Following are links to other sections that may be useful.

- General data control features: Before beginning work with EJB data controls, it is important to understand the broader data control concepts. For more information, see Chapter 2, "Using ADF Data Controls."
- ADF Model and data binding: When you create forms in an ADF web application, you use ADF Model and data binding. For more information, see "Using ADF Model in a Fusion Web Application" in *Developing Fusion Web Applications with Oracle Application Development Framework* and *Java API Reference for Oracle ADF Model*.
- ADF Faces: When you create databound UI components, they are likely to be from the ADF Faces component set. For more information, see "Creating a Databound Web User Interface" in *Developing Fusion Web Applications with Oracle Application Development Framework*.
- ADF task flows: Task flows extend JSF page flows to provide a modular and transaction-aware approach to navigation and application control. For more information, see "Creating ADF Task Flows" in *Developing Fusion Web Applications with Oracle Application Development Framework*.
- Java EE support in JDeveloper: JDeveloper provides support for creating Java EE components in a way that is optimized for use with EJB data controls. For more information, see "Developing with EJB and JPA Components" in *Developing Applications with Oracle JDeveloper*.

3.2 Preparing a Session Bean to Use With a Data Control

An EJB data control encapsulates an EJB session bean and exposes the bean's code elements as data control objects, which can then be used to bind those code elements to UI components. When you create a data control based on an EJB session bean, the data control relies on coding patterns in the bean. This section shows the mapping between given code patterns and data control objects and shows patterns that you can use in your beans to maximize the data control features.

Note: The data control *does not replace* the bean code. Rather, it serves as a thin adapter layer between the bean and the binding layer. The methods in your bean are used at runtime as you have coded them. Any declarative metadata that you specify in the data control, such as UI hints or validation rules, augments the session bean's business logic. Likewise, if you add or change methods after you have created the data control, the data control works with those new or revised methods.

3.2.1 Supported Types and Constructs in EJB Data Controls

EJB (and bean) data controls are compatible with classes that use the following Java types and constructs:

- Java primitive types and arrays
- Complex Java types, such as your own beans
- Java scalar types, including types from the `java.math`, `java.sql`, and `java.util` packages
- Collection types encompassed by the `java.util.Collection`, `java.util.List`, and `java.util.Map` packages
- Generics, strongly-typed collections, and wildcards
- Java Persistence API (JPA) features such as:
  - `@Id` annotations to determine primary keys
  - `@NamedQuery` annotations and the full range of JPQL syntax for named queries
  - `@JoinColumn` annotations to denote master-detail and list-of-value (LOV) relationships.

3.2.2 EJB Data Control Objects

When you create a data control based on an EJB session bean (or a POJO bean), the data control exposes several different types of objects, each of which you can bind to a variety of UI components.

For EJB and bean data controls, data control objects are exposed for collections that are returned by any public method starting with `get`, as well as complex types and primitives that are returned by a collection or a `get` method. In these cases, the name of the data control object typically matches the name of the `get` method, but with the leading `get` removed from the name.

If a returned object has a relationship to another object defined through a `JoinColumn` annotation, then a child object is displayed in the Data Controls panel.

If the fields of a bean are exposed with public `get` methods, those fields are exposed as child attributes of the collection or structured attribute in the Data Controls panel. For `get` methods that return arrays and simple collections composed of primitives or strings, a child attribute node called `element` is exposed the Data Controls panel.

For methods that are not pre-pended with `get`, the methods are exposed as method objects in the Data Controls panel. Returns of these methods are exposed as method return objects.

In addition, built-in data control operations become available for many of those objects. See Section 2.2.4, "Data Control Built-in Operations" for the full list of those operations.

> **Note:** EJB and bean data controls also expose bean constructors, which you can use to create buttons in the UI for creating new instances of the bean. This provides an alternative to using the data control's `Create` operation for creating instances. Buttons created from constructor nodes in the Data Controls enable the user to create a new object instance without adding that object to the persistence context (which typically happens during the `Create` operation). You can declaratively access that object instance by binding to the `result` property of the binding object for the constructor.

Table 3–1 shows how the various data control objects map to the elements of EJB classes. For information on using these objects, see Section 2.3, "Creating Databound UI Components from the Data Controls Panel."

Table 3–1 Important EJB and Bean Data Control Objects

Icon	Name	Description
	Collection	Is exposed for any public `get` method that returns a collection (any object implementing `java.util.Collection`). The children under a collection may be attributes of the collection, related collections, custom methods that return a value from the collection, or built-in operations that can be performed on the collection.
	Structured Attribute	Is exposed for any public `get` method that returns a complex Java type that is not a collection.
	Attribute	Is exposed for any public `get` method that returns a Java primitive or `String`, such as a column in an entity bean).
	Key Attribute	Is exposed for any attribute that is marked as the primary key (or which is part of a composite primary key). A key attribute can be designated with an `@Id` annotation in the entity bean class or in the data control structure file for a bean. For more information, see Section 3.2.4.1, "Recommended Entity Bean Elements" and Section 7.3.1, "How to Designate an Attribute as Primary Key."
	Method	Represents methods that are not pre-pended with `get`. These methods may return single values, structures, or collections.
	Method Return	Represents an object that is returned by a method or other operation. 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, or operations that can be performed on the parent collection.
	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.

Table 3–1 (Cont.) Important EJB and Bean Data Control Objects

Icon	Name	Description
	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.
	Named criteria	Represents a metadata-based query from which you can create a user search form. You can create custom view criteria and add them to the Data Controls panel. See Section 7.6, "Filtering Result Sets with Named Criteria."
	Constructor	Represents a constructor for one of the creatable types encompassed by a bean or EJB data control. You can use this data control object to create a command control that the user can click to create a new instance of that type.

3.2.3 About the Session Facade Pattern

EJB data controls are based on the EJB session facade design pattern, in which a session bean mediates access to individual entity beans, which contain the code to query database tables. When you create a data control based on a session bean, the data control exposes top-level objects based on the session bean's methods and lower level objects based on the detail in the entity beans that is retrieved by the session bean's accessor methods.

3.2.4 EJB Data Control Prerequisites and Considerations

In order to take advantage of the full functionality of EJB data controls, you need to include some elements in your classes that the data controls can use to present the structure of the services.

Your project should contain the following types of classes:

- JPA entity classes for every database table that your application needs to query.
- One or more session beans that contain accessor methods to the entity beans and other business methods. EJB data controls are based on sessions beans.
- Optionally, (POJO) service facade classes that mirror the structure of the session beans. These service facade classes are useful for testing the services without having to run them in an application server container. You can also create data controls for service facade classes in order to test the services with the data control.

3.2.4.1 Recommended Entity Bean Elements

Your entity beans should contain the following elements:

- `@NamedQuery` annotations containing queries that return each row of the collection. You can also take advantage of the constructs of the Java Persistence Query Language (JPQL) to add more selective queries. For more information on JPQL, see `http://docs.oracle.com/javaee/6/tutorial/doc/bnbtg.html`.
- `@JoinColumn` annotations for any columns that reference other tables (or other columns in the same table) by foreign key. The generated data control will then expose these joined columns as master-detail relationships and simplify the creation of UI components that rely on those relationships.
- `add`*`EntityBeanName(CollectionType collectionParam)`* and `remove`*`EntityBeanName(CollectionType collectionParam)`* methods for each of the collections represented by the entity beans. The data control's `Create` and `Delete` operations rely on these methods to add and remove rows in a collection.

- Primary keys designated for each entity bean. This is necessary for the `Create` operation to function properly when creating new rows at runtime.

  You can designate primary key columns in entity beans by adding (or generating) `@Id` annotations for the appropriate columns.

- A strategy for generating primary key values when creating new records. This is particularly important when you are using explicit commit models where you need to persist a new record upon its creation but before the user fills in the details for the other fields. For more information, see Section 3.2.8, "About Generating IDs for Primary Keys with the @GeneratedValue Annotation."

3.2.4.2 Recommended Session Facade Elements

It is recommended that your session beans (and/or Java service facade classes) contain the elements in the following list in order to integrate with data control features. If you use JDeveloper's wizards to create your classes, these elements are generated for you by default.

- Getter methods that return the results of named queries specified in the entity beans. For example, the method shown in Example 3–1 would return the results of a named query called `Customer.findById`.

  When you create a query that includes named parameters, the data control object that is created for that query includes the `ExecuteWithParams` built-in operation, which you can use to quickly create forms that are based on parameters that are supplied at runtime.

  > **Tip:** If you have multiple named queries for an entity bean, you can create a getter method for each named query to create distinct data control collection objects for each query.

- The `queryByRange(String jpqlStmt, int firstResult, int maxResults)` method, in order to take advantage of built-in support for JPA named queries, named criteria metadata, scrollable access mode, and range paging access mode. The data control will use this method to perform all named queries, instead of invoking the getter methods for the queries in the session bean.

  Example 3–2 shows the code that is generated for the `queryByRange(String jpqlStmt, int firstResult, int maxResults)` method when you use JDeveloper to generate a session bean (or a plain Java facade using the Java Service Facade wizard). You can change the implementation of this method, but you need to keep the signature as is, since that's what the ADF Model runtime looks for.

- (If your bean has transactional behavior), the following methods with these exact signatures:

  - `public boolean isTransactionDirty()`
  - `public void rollbackTransaction()`
  - `public void commitTransaction()`

  These methods are used to implement the data control's `commit` and `rollback` operations. When you use the JDeveloper wizards for creating Java service facade classes and EJB stateful container-managed session beans, working implementations of these three methods are generated in the classes by default.

Example 3–1 Getter Method That Returns the Results of a Named Query

```
/** <code>select o from Customer o where o.id = :custId</code> */
```

```
public List<Customer> getCustomerFindById(BigDecimal custId) {
    if (custId != null) {
        Long custIdLong = new Long(String.valueOf(custId));
        return em.createNamedQuery("Customer.findById").setParameter("custId",
custIdLong).getResultList();
    } else {
        return getCustomerFindAll();
    }
```

Example 3–2 Code Listing for queryByRange() Method

```
public Object queryByRange(String jpqlStmt, int firstResult, int maxResults) {
    Query query = em.createQuery(jpqlStmt);
    if (firstResult > 0) {
        query = query.setFirstResult(firstResult);
    }
    if (maxResults > 0) {
        query = query.setMaxResults(maxResults);
    }
    return query.getResultList();
}
```

3.2.4.3 What You May Need to Know About Overloaded Get Methods

When you create a session bean or service facade class to be consumed by a data control, the class should not use overloaded `getXxx()` methods (i.e. multiple versions of a method that have the same name but each of which takes different parameters). At runtime, the data control is unable to properly distinguish between the different versions of the method. To work around this constraint, rename any overloaded `getXxx()` methods you may have and give them names that are unique within their class.

3.2.5 Creating EJBs for a Data Control in JDeveloper

JDeveloper's wizards enable you to create EJB entity classes and session beans that are optimized for use with data controls. For more information, see "How to Work with an EJB Business Services Layer" in *Developing Applications with Oracle JDeveloper*.

3.2.6 What You May Need to Know About How EJB and Bean Data Controls Use Getter Methods

When you use the `queryByRange(String jpqlStmt, int firstResult, int maxResults)` in your session bean (and the data control uses the `oracle.adf.model.adapter.bean.jpa.JPQLDataFilterHandler` handler), the data control will use this method to perform all named queries, instead of invoking the getter methods for the queries in the session bean. If you have custom logic that you have added to the getter methods, it will not be called when the query is run. If you need that custom logic to run, you can incorporate it into the `queryByRange(String jpqlStmt, int firstResult, int maxResults)` method.

When you do not include the `queryByRange(String jpqlStmt, int firstResult, int maxResults)` in your session bean, the data control based on that bean is generated with the `oracle.adf.model.adapter.bean.DataFilterHandler` handler. In this case, the data control will invoke the session bean's getter methods directly, but you will not be able to use declarative named criteria on objects in the data control and you will need to manually implement methods for scrollable and range paging to

work. For more information on the use of named criteria, see Section 7.6, "Filtering Result Sets with Named Criteria." For more information on paging, see Section 3.4, "Paginated Fetching of Data in EJB Data Controls."

3.2.7 About Commit Models for EJB Session Beans

EJB data controls can be used for different varieties of EJB session beans, whether they are bean-managed or container managed or whether they are stateful or stateless. JDeveloper's Create Session Bean wizard can generate code for the following types of session beans on which you can create a data control for updating data:

- stateless container-managed transactions (CMT) with implicit commit
- stateless bean-managed transactions (BMT) with implicit commit
- stateful CMT with implicit commit
- stateful BMT with implicit commit
- stateful CMT with explicit commit
- stateful BMT with explicit commit

3.2.7.1 Implicit Commit Models

For data controls based on any of the implicit commit variants, the transactional operations `Commit` and `Rollback` are not provided. For these beans, any use of the `persistEntity` or `mergeEntity` methods update the data in the underlying data source.

3.2.7.2 Explicit Commit Models

When you work with beans with an explicit commit model, the underlying data source is not updated until the `Commit` operation is called. This enables a user to create or make changes to multiple rows in multiple tables and then later commit them all to the data source with one click.

For data controls based on any of the explicit commit variants, you still might need to add steps to merge or persist changes to the persistence context before those changes can be committed. For example, you might need to have the user click buttons both to persist new rows and to commit the group of changes.

However, for some session beans, it is possible to configure the data control so that its `Create` operation automatically calls the entity's persist method when it is invoked. This is the default behavior for data controls for session beans created with JDeveloper's Create Session Bean wizard that are configured as stateful with container-managed transactions and an explicit commit model. For more information, see Section 3.3.8, "About Automatically Persisting New Rows."

Where it is not possible to persist new rows immediately upon their creation, you can use a managed bean to override or combine operations and methods. For more information, see "Overriding Declarative Methods" in *Developing Fusion Web Applications with Oracle Application Development Framework*.

Also for many beans, and particularly for stateful CMT beans with explicit commit, you need to implement ID generation for the primary key columns. For more information, see Section 3.2.8, "About Generating IDs for Primary Keys with the @GeneratedValue Annotation."

3.2.8 About Generating IDs for Primary Keys with the @GeneratedValue Annotation

In general it is convenient to let JPA auto-generate primary key values when you create a new instance of an entity. For many data models, you must generate values to populate the primary key columns of your tables to avoid constraint violation errors when the user tries to persist a new row. In JPA entity classes that use a simple primary key type (like `Integer`, `long`, `String`, etc.), you can use `@GeneratedValue` annotation for an entity's primary key column to designate how and from where the new record obtains that value. You then need to ensure that those values are generated by the database.

The Create Entities from Tables wizard in JDeveloper can help you configure your entities to have their primary key values automatically generated and assigned. This option is available to entities which use a simple primary key type, such as `Integer`, `String`, `long`, `int`, etc. When this option is enabled, the entity's primary key field is annotated with `@GeneratedValue`, which indicates how the key value should be generated. The wizard enables you to choose between the `SequenceGenerator` and `TableGenerator` strategies.

Once your classes have the `@GeneratedValue` annotation, you need to make sure that the values are generated by the database upon creation of a new row and passed to the entity. You can do this in one of the following ways:

- Update the persistence unit for the entities to have the schema objects automatically created upon deployment by setting the `eclipselink-ddl-generation` property to `create-tables` or `drop-and-create-tables`. For more information, see Section 3.2.9, "How to Change a Persistence Unit's Schema Generation Behavior."
- Manually update the online database schema to incorporate the ID generation objects specified in the `@GeneratedValue` annotations.

  When you use the Create Entities from Tables wizard in JDeveloper, schema objects for any sequence or table generators are created and displayed under the **Offline Database Sources** node for the data model project in the Applications window. You can add the object to the online schema by right-clicking the node for the object, choosing **Generate to > *ConnectionName***, and completing the ensuing Generate Database Objects from Database wizard.
- If you already have ID generation objects in the live database schema, manually change the annotation attributes to refer to those objects. For example, for the `@SequenceGenerator` annotation, you would change the `sequenceName` attribute.

3.2.9 How to Change a Persistence Unit's Schema Generation Behavior

By default, when you use the Create Entities from Tables wizard to create entities, it creates a persistence unit with the `eclipselink-ddl-generation` property configured to use its default value (`none`). This means that EclipseLink, which is the default persistence provider for the generated entities, will not generate any Data Definition Language (DDL) statements or schema changes at runtime.

However, when you are developing the application, it might be useful to configure this property so that schema objects are created or re-created in the database schema each time that you test deploy the application. That will save you from having to manually keep the database schema in sync with incremental changes that you make as you develop the data model. For example, if you set the `eclipselink-ddl-generation` property to `create-tables`, EclipseLink will try to generate any schema objects that are specified by the entities and which do not yet exist in the schema, including sequences and tables for ID generation.

To change a persistence unit's schema generation behavior:

1. In the Applications window, expand the data model project and double-click the persistence unit file (which is located at **Application Sources > META-INF > persistence.xml** by default).
2. If there are multiple persistence units, in the overview editor for the persistence unit file, select the persistence unit for your project and click the **Go to the Persistence Unit** icon.
3. In the overview editor for the persistence unit, select the **Schema Generation** page.
4. From the **Generation Type** dropdown list, select the type of schema generation that you want.

For more information on the `eclipselink-ddl-generation` options, see the EclipseLink documentation at `http://www.eclipse.org/eclipselink/documentation/2.4/jpa/extensions/p_ddl_generation.htm`.

WARNING: Before deploying your finished application to a production environment, the database schema should be finalized, and you should change the `eclipselink-ddl-generation` property back to `none`.

3.2.10 How to Automatically Update a Session Facade

If you change an entity bean, you can use JDeveloper to quickly update the session bean and, if applicable, its remote and local interfaces. The Edit Session Facade wizard enables you to generate code in the session bean to expose named queries and methods that have been added to your entity classes. If you have a data control based on that session bean, any newly added methods become available to you in the Data Controls panel immediately after you refresh the panel.

To update a session facade based on updated entity beans:

1. In the Applications window, right-click the session bean and choose Edit Session Facade.
2. In the Specify Session Facade Options dialog, select any methods that you would like generated in the session bean.

3.2.11 What You May Need to Know About Refreshing JPA Queries

By default, when a JPA query is run, the results of the query are cached and that cache is used if the query is re-issued from the same session. Therefore, if a change occurs to the database from a second user's session after the first user's initial query, the first user might not see the changes made by the second user when re-running the query (e.g. when refreshing a page).

To make sure that a fresh JPA query is always run, you can use a hint on the query to refresh the cache.

In JPA 2.0, you can apply the following hint to the query:

```
setHint("javax.persistence.cache.storeMode", "REFRESH")
```

For the `getCustomerFindAll()` method, this would look like the following:

```
public List<Customer> getCustomerFindAll() {
```

```
    return em.createNamedQuery("Customer.findAll").
        setHint("javax.persistence.cache.storeMode", "REFRESH").
        getResultList();
}
```

In JPA 1.0, you can apply the following hint (assuming TopLink is your persistence provider):

```
setHint("eclipselink.refresh", "true")
```

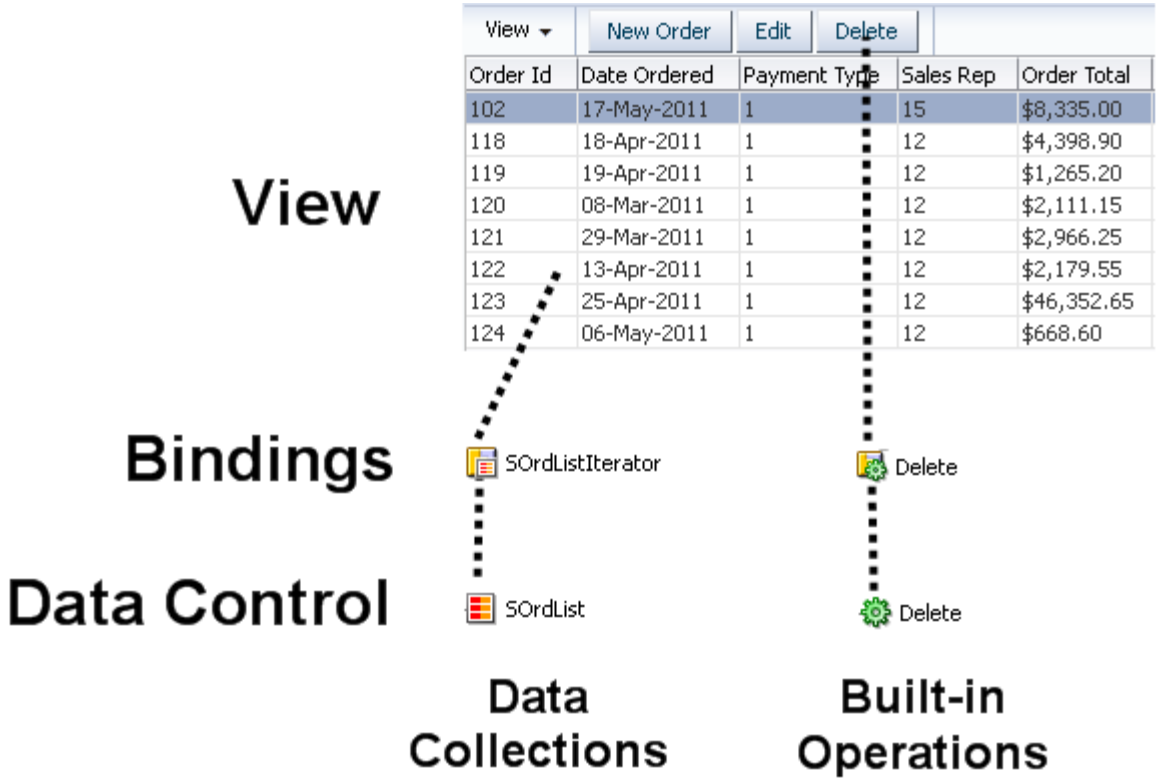
3.3 Exposing Session Bean Services with ADF Data Controls

Once you have your application's services in place, you can use JDeveloper to create data controls that provide the information needed to declaratively bind UI components to those services.

In a standard Java EE application, you normally create entity beans that represent tables in a database and then create a session facade over all the EJBs. This facade provides a unified interface to the underlying entities. You then would typically use other classes to coordinate the interaction between the user interface and the services exposed by the session facade.

In an Oracle ADF application, you can eliminate the need to programmatically coordinate the model and view layers by creating a data control to encapsulate the services represented by the service facade and enable declarative data binding between the layers. You then create UI components that are declaratively bound to the services through the data control. The bindings take the form of EL expressions that reference the data control. Figure 3–1 illustrates the coordination between UI components and data control objects that is possible with declarative bindings.

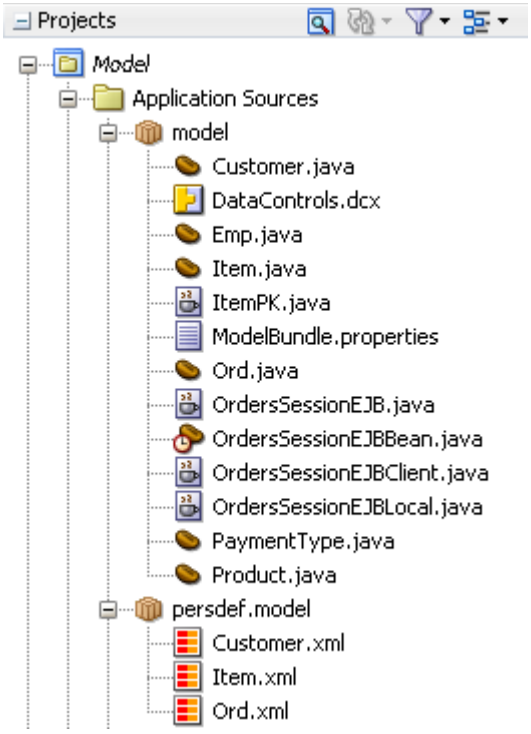
Figure 3–1 Binding Between View and Service



Data controls consist of one or more XML metadata files that define the capabilities of the services that the bindings can work with at runtime. The data controls work in conjunction with the underlying beans without changing the implementation of the beans.

For example, Figure 3–2 shows an EJB data model project in the Applications window. The project has a number of entity beans that represent database tables, such as the `Customer` bean, the `Product` bean, the `Order` bean, and so on. The project also contains a session bean, `OrdersSessionEJBBean`, which is used to access the beans created from tables. This session bean also contains service methods for persisting, merging, and removing records. There is a data control for the session bean, which allows developers to declaratively create UI pages based on the methods of the session bean and the entity beans that the session bean encapsulates. In addition, there are XML files in the `persdef.model` package that correspond with the `Customer`, `Product`, and `Order` beans that contain additional metadata such as UI hints and validation rules. These additional XML files are only needed if you are adding metadata for a given data collection.

Figure 3–2 EJB Model Project



3.3.1 How to Create EJB Data Controls

You create data controls from within the New Gallery or the Applications window.

Before you begin:

It may be helpful to have a general understanding of using EJB data controls. For more information, see Section 3.3, "Exposing Session Bean Services with ADF Data Controls".

You may also find it helpful to understand general data control features and functionality that you may need to use in the application with the data controls. For more information, see Section 3.1.2, "Additional Functionality for EJB Data Controls."

In addition, you may find it helpful to understand the code patterns and constructs in your session bean that the data control uses. For more information, see Section 3.2, "Preparing a Session Bean to Use With a Data Control."

You need to complete this task:

Create an application workspace, JPA/EJB 3.0 entities, and one or more session beans for the entities. For more information, see "How to Work with an EJB Business Services Layer" in *Developing Applications with Oracle JDeveloper*.

To create an EJB data control:

1. In the Applications window, right-click the session bean for which you want to create a data control and choose **Create Data Control**.
2. In the Choose Session EJB page of the Create EJB Data Control wizard, specify a name for the data control instance.

Note: You can create multiple data control instances with different behavior for the same bean. For more information, see Section 3.3.11, "How to Create Different Data Controls for a Single Bean."

3. In the Choose Session EJB Business Interface page of the wizard, choose **Local** or **Remote**. For web applications, typically you would choose **Local**.
4. In the Choose ADF Data Controls Features page, select any of the following checkboxes for additional data control features that you would like to use in your application. (This page of the wizard only appears when you are creating the data control over a stateful session bean.)

   Methods will be added to the session bean to implement the selected data control features.

   - **Transactions**. Selecting this feature generates the `commitTransaction()`, `rollbackTransaction()`, and `isTransactionDirty()` methods in your session bean and maps them with the data control's `commit` and `rollback` operations. If you have used JDeveloper to create a stateful and container-managed session bean, these methods should already be implemented, in which case the **Transactions** checkbox would be selected by default.
   - **Custom CRUD**. This feature enables you to provide your own implementation of persistence behavior. EJB data controls already have CRUD functionality from JPA, so you would only select this feature if you want to override JPA's functionality. For more information, see Section 3.3.5, "What You May Need to Know About CRUD Operations in an EJB Data Control."
   - **Failover**. For more information, see Section 3.6, "Enabling Failover in an EJB Data Control."
5. In the EJB Data Control Options page, select any additional options.
   - **Access Mode**. Enables you to set how the data control fetches and stores data in memory. For more information, see Section 3.4, "Paginated Fetching of Data in EJB Data Controls."
   - **Support Named Criteria**. When selected, the data control includes built-in support for declarative named criteria, which can be used to create quick search forms. For more information, see Section 7.6, "Filtering Result Sets with Named Criteria."

     The Support Named Criteria option is only available for JPA-based beans that also contain a `queryByRange()` method. The data control uses the bean's `queryByRange()` method to handle all queries that otherwise would be carried out by individual getter methods on the bean. For more information, see Section 3.2.6, "What You May Need to Know About How EJB and Bean Data Controls Use Getter Methods."
   - **Generate Metadata**. You can select this option to automatically generate metadata files for all of the beans represented by the data control upon the creation of the data control. This option is not necessary, since metadata files are created on demand when you edit a data control. However, this option might be useful if you plan to make the application available for MDS customization. For more information, see Section 7.2.4, "What You May Need to Know About MDS Customization of Data Controls."
6. Click **Finish**.

Note: If you later rename the bean on which a data control is based, you must again use the Create Data Control command in order to regenerate the data control's metadata. When doing so, you can keep the same data control instance name.

If you merely make changes to a bean after the data control is created, you do *not* have to regenerate the data control. The data control incorporates any changes made to the bean. However, you might need to close and reopen the project in order for the data control to incorporate the changes to the underlying beans.

3.3.2 What Happens in Your Project When You Create an EJB Data Control

When you create a data control based on an EJB session bean, JDeveloper does the following things:

- Creates the data control definition file (`DataControls.dcx`) and opens the file in the overview editor.

  Depending on the configuration of the bean on which the data control is based and the options that you have chosen in the wizard, various elements and properties are configured in the `DataControls.dcx` file and can be seen in its source view. These elements and properties include:

  - `<CreatableTypes>`, which specifies the entities that are encompassed by the facade class and for which declarative metadata can be created.
  - `DataControlHandler`, which specifies a handler class that implements various features for the data control, including support for paging and named criteria.
  - `AccessMode`, which determines how the data control fetches data in the running application. For more information, see Section 3.4, "Paginated Fetching of Data in EJB Data Controls."
  - `EagerPersist`, which determines whether new rows are added to the persistence context when created. For more information, see Section 3.3.8, "About Automatically Persisting New Rows."

- Displays the hierarchy of the resulting data control objects in the Data Controls panel.
- If you have selected any features on the ADF Data Controls Features page, adds methods to implement those features to the session bean.
- If you have selected Generate Metadata, generates XML data control structure files for the high-level data control objects. These files hold any declarative metadata, such as UI hints and validation rules, that you define for given data control objects. If you have not selected this option, the data control structure files are created on demand when you use the `DataControls.dcx` overview editor to add declarative metadata to data control objects. For more information, see Chapter 7, "Adding Business Logic to Data Controls."

For general information on the overview editor and Data Controls panel, see Section 2.2.2, "What Happens in Your Project When You Create a Data Control." For information specific to EJB and bean data controls, see Section 3.3.3, "How EJB and Bean Data Controls Appear in the IDE."

3.3.3 How EJB and Bean Data Controls Appear in the IDE

Once you have created an EJB or bean data control, you can use the overview editor for the `DataControls.dcx` file to further configure the data control, and you can use the Data Controls panel to create databound UI components.

3.3.3.1 DataControls.dcx Overview Editor for EJB and Bean Data Controls

The overview editor for the `DataControls.dcx` file provides a view of the master-detail hierarchies of your data model as well as methods from the session facade. When you select a node, you can view the fields that can be mapped to database columns in the corresponding entity class in the Attributes tab. In the Accessors tab, you can view fields for the corresponding entity class that have entity relationships defined (such as `OneToMany` and `ManyToOne`). In the Operations tab for collections, you can view entity methods that the data control uses for standard operations, such as the add and remove methods of the collection accessors.

See Table 2–1 for a description of the icons that are used in the overview editor and Data Controls panel.

You can change the settings for a data control by selecting an element and clicking the **Edit** icon. For more information about editing a data control, see Chapter 7, "Adding Business Logic to Data Controls."

3.3.3.2 Data Controls Panel for EJB and Bean Data Controls

The Data Controls panel serves as a palette, from which you can create databound UI components by dragging nodes from the Data Controls panel to the design editor for a web page. For information on the contents of the Data Controls panel, see Section 2.2.3, "Display of Business Services in the Data Controls Panel."

In addition for EJB and JPA-based bean data controls, nodes for named criteria appear, as shown in Table 2–1. Named criteria are used to create quick search forms. By default, an implicit named criteria called All Queriable Attributes appears for each queriable collection when you create the data control. You can create additional declarative named criteria for each collection as described in Section 7.6, "Filtering Result Sets with Named Criteria."

For information on creating databound UI components from a data control, see Section 2.3, "Creating Databound UI Components from the Data Controls Panel."

3.3.3.3 EJB and Bean Data Control Built-in Operations

EJB data controls also provide standard built-in data control operations that you can use to create command components in your user interface for page navigation and data operations.

For information on all of the operations available to data controls, see Section 2.2.4, "Data Control Built-in Operations."

3.3.4 What You May Need to Know About the Support Named Criteria Option and Paging

The Support Named Criteria option that is offered when you create an EJB or JPA-based bean data control affects the handler class that is used by the data control, which also affects features such as access mode.

If you select the Scrollable or Range Paging access mode when creating your data control, you should leave the Support Named Criteria option selected. Otherwise, the data control is generated to use the `DataFilterHandler` handler, which means that you

would have to manually implement paging methods. (The `JPQLDataFilterHandler`, which is used if you select the Scrollable or Range Paging access mode and keep the Support Named Criteria option selected, implements the access mode for you without requiring further coding.)

For more information on access modes in data controls, see Section 3.4, "Paginated Fetching of Data in EJB Data Controls."

3.3.5 What You May Need to Know About CRUD Operations in an EJB Data Control

When you create an EJB data control, CRUD (Create/Read/Update/Delete) features rely on the session bean's service methods and the entity beans encompassed by the session bean.

For example, a session bean's persist and merge methods are represented in the Data Controls panel and you can use them to create buttons that allow the user to persist and merge the current instance of the object.

There are also built-in data control operations available that you can use to bind data operations to the UI. These operations in turn call the appropriate methods in the session facade and entity classes. In addition, they may update the state of the ADF iterator. For example, when invoked, the `Create` operation for a data collection calls the constructor of the entity class that represents that collection to create the instance and then calls the appropriate persist method in the session bean to add the new instance to the JPA persistence context. Then the newly created instance is added to the ADF iterator.

> **Tip:** Where applicable, it is generally best to use these built-in operations, because they handle communication with both the JPA entity manager and the ADF iterators, keeping the two in sync.

You can also implement your own logic for CRUD (Create/Read/Update/Delete) operations if you do not wish to rely on JPA for these features. For more information on implementing custom CRUD operations, see Section 4.6, "Enabling Custom CRUD Operations in a Bean Data Control."

3.3.6 What You May Need to Know About the Merge and Persist Methods

If, when you created your session bean, you chose to expose merge and persist methods, then those methods appear in the Data Controls panel and you can use them to create buttons that allow the user to merge and persist the current instance of the object. Which you use depends on whether the page will need to interact with the instance once updates are made. If you want to be able to continue to work with the instance, then you need to use the persist method.

The merge methods are implementations of the JPA `EntityManager.merge` method. This method takes the current instance, copies it, and passes the copy to the `PersistenceContext`. It then returns a reference to that persisted entity and not to the original object. This means that any subsequent changes made to that instance will not be persisted unless the merge method is called again.

The persist methods are implementations of the JPA `EntityManager.persist` method. Like the merge method, this method passes the current instance to the `PersistenceContext`. However, the context continues to manage that instance so that any subsequent updates will be made to the instance in the context.

3.3.7 What You May Need to Know About Remove Methods

When you have a data control that is based on a session bean or a Java service facade that contains remove methods (to remove objects from the JPA persistence context), those methods are exposed in the Data Controls panel along with other facade methods. However, when deleting an entity from a form or table bound to an ADF iterator, you should generally use the `Delete` operation instead of a remove method. The `Delete` operation calls the remove method on the facade and also notifies the ADF iterator of the changes, ensuring that the iterator and the persistence context stay in sync. If you call the remove methods directly in order to take advantage of any custom behavior that you have coded within them, you also need to provide the code to refresh the ADF iterator.

3.3.8 About Automatically Persisting New Rows

For most variants of EJB (and JPA-based bean) data controls, the `Create` operation creates a new row and inserts it into the row set. However, it does not add the newly created object to the persistence context. That is generally the desired behavior for stateless and implicit-commit data models, since attempting to persist the instance immediately upon creating it may violate constraint violations due to mandatory fields being empty.

However, some models support the option of calling the persist method during the `Create` operation, because the underlying DML to persist the row is deferred until commit time. When the Create Session Bean wizard in JDeveloper is used to generate a stateful session bean using CMT and explicit commit behavior, it generates a model that supports this eager persist behavior. The advantage this model has in a conversational web application is that the application can combine these steps of create and persist into a single gesture (typically a button). After creating multiple instances, the user can click a commit button to apply all of the pending changes, at which point a transaction is begun and the DML statements are executed in the database to perform any pending `INSERT`, `UPDATE` or `REMOVE` operations.

When you create a data control for an EJB or a bean, the class is scanned to see if it is a candidate for this "eager persist" behavior. If so, the data control's `EagerPersist` property is set to `true`. You can override the initial value to enable or disable this option as you wish.

Calling an entity's persist method eagerly is generally desirable only when the following conditions are met:

- The session bean has explicit commit behavior (since implicit commit behavior would mean that the object would be committed as soon as it was created).
- The model does not attempt to execute SQL right away when the persist method is called.
- You have set up ID generation for the primary key column. For more information, see Section 3.2.8, "About Generating IDs for Primary Keys with the @GeneratedValue Annotation."
- The EJB session been is stateful and uses the following:
  - container-managed transactions
  - the `@TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED)` annotation for the session bean class,
  - the `@TransactionAttribute(TransactionAttributeType.REQUIRED` for the commit method

If you use JDeveloper's Create Session Bean wizard to create a stateful session bean with container-managed transactions and an explicit commit model, any data control that you create for this bean will be generated with the `EagerPersist` property set to `true`. Otherwise, `EagerPersist` is set to `false`.

3.3.9 How to Change the EagerPersist Property

If your bean meets the conditions to work with eager persist behavior (as described in Section 3.3.8, "About Automatically Persisting New Rows") but it is not turned on (which might be the case if you did not use the Create Entities from Tables wizard to create the entity classes), you can manually set the `EagerPersist` property to `true`. Likewise, if the property is set to `true` but you do not want eager persist behavior, you can change it to `false`.

To change the value of the EagerPersist property for a data control:

1. In the Data Controls panel, right-click the data control's root node and choose Edit Definition.
2. In the ejb-definition Properties dialog, change the value of the `EagerPersist` property.

3.3.10 What You May Need to Know About the Persistence Context and Resubmitting Queries

When you have a data control based on a stateful container-managed session bean, pre-commit changes made in a session are not reflected in any subsequent query that is made to the data source. So, if a user is in the middle of a transaction and performs an operation that requeries the data source, the subsequent refresh of the data on the page will not include changes made in the session.

3.3.11 How to Create Different Data Controls for a Single Bean

You can create multiple instances of a data control for a single bean, which can be useful if you need to make contrasting data control features available to the UI developer. For example, you may want to make it possible for a UI developer to create some UI components with scrollable paging and others with range paging.

Before you begin:

It may be helpful to have a general understanding of using EJB data controls. For more information, see Section 3.3, "Exposing Session Bean Services with ADF Data Controls".

You may also find it helpful to understand general data control features and functionality that you may need to use in the application with the data controls. For more information, see Section 3.1.2, "Additional Functionality for EJB Data Controls."

In addition, you may find it helpful to understand the code patterns and constructs in your session bean that the data control uses. For more information, see Section 3.2, "Preparing a Session Bean to Use With a Data Control."

You need to complete this task:

Create a data control as shown in Section 3.3.1, "How to Create EJB Data Controls."

To create an additional data control instance for a bean:

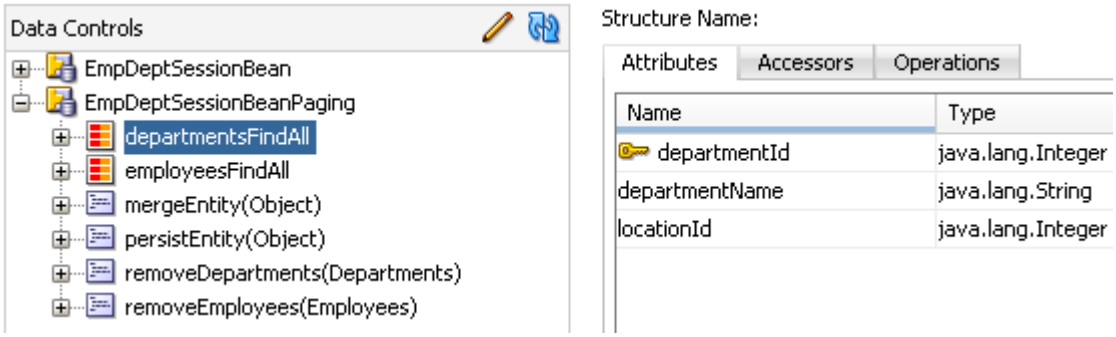
1. Right-click the bean for which you want to create the additional data control instance and choose **Create Data Control**.

2. In the Choose Session EJB page (or Choose Bean Class) page of the wizard, specify a unique name for the data control instance (different from any previous data control instances that you have created).
3. Complete the wizard with the options that you want for that instance of the data control.

3.3.12 What Happens When You Create an Additional Data Control Instance

When you create an additional data control instance for a bean, an additional high-level node appears in the overview editor for the `DataControls.dcx` file, as shown in Figure 3–3 and in the Data Controls panel.

Figure 3–3 Data Control Overview Editor with Two Data Control Instances



Note: When you add declarative metadata such as UI hints and validators to a data control, the different data control instances for a bean use that same metadata. For more information on adding declarative metadata, see Chapter 7, "Adding Business Logic to Data Controls."

3.4 Paginated Fetching of Data in EJB Data Controls

When you create an EJB or bean data control, you can use the wizard's **Access Mode** dropdown list to determine how records are accessed from the database and whether to limit the number of records that are held in memory at a time.

In EJB and bean data controls, there are the following possibilities for fetching and storing data in memory:

- Scrollable access mode.

  If you accept the defaults when creating the data control, the data access mode is set to `scrollable`. This means that the data that your application needs to display is retrieved from the database as needed (in increments equal to the range size specified by the UI component's iterator) and stored in memory. Then, when the user scrolls forward through the application, additional rows are fetched as needed and stored in memory. All rows that have been fetched remain in memory.

  For example, if the running application contains a table that displays rows 1 through 20 on a web page and the table's iterator has a range size of 25 (the default), the data control will fetch the first 25 rows. If the user scrolls down to display rows 477 through 496 of the result set, the data will be fetched in sets of 25 as the user scrolls until rows 26 through 500 are fetched. At that point, a total of 500 rows will be stored in memory.

This is the default mode for data controls using `oracle.adf.model.adapter.bean.DataFilterHandler` and `oracle.adf.model.adapter.bean.jpa.JPQLDataFilterHandler`. However, for data controls using `oracle.adf.model.adapter.bean.DataFilterHandler`, you still need to add paging methods to your data control to implement the access mode. For more information, see Section 4.4.1, "How To Manually Implement Pagination Support in a Data Control."

- Range paging access mode

  To limit the amount of records that are fetched and stored in memory at a time, you can use the `rangePaging` access mode. As with scrollable mode, range paging mode allows your applications to fetch data in increments. The main difference in range paging mode is that only the most recently fetched increment is retained in memory. So, for example, if the accessor iterator's `rangeSize` attribute is set to 25, no more than 25 records will be held in memory at any given time.

  In a range paging version of the scrollable example above, the data control would fetch rows 1 through 25 and hold them in memory in order to display rows 1 through 20. If the user scrolled down, the data control would fetch data in increments of 25 as the user was scrolling but release the previous 25 records from memory as it fetched a new range. By the time the user reached rows 477 through 496 as in the example above, only rows 476 through 500 would be in memory.

  When scrolling to a position that displays data from multiple increments, only the data from the increment last fetched is held in memory.

  You can set page ranging when creating the data control by selecting Range Paging in the Access Mode dropdown of the Create EJB Data Control wizard.

  Note: When you use range paging in a data control, the built-in navigation operation `Last` does not work on databound UI components created from that data control.

- No pagination. When there is no pagination, all available data for a UI component is fetched.

  You can configure the data control to not use any paging by selecting No Paging in the Access Mode dropdown of the Create EJB Data Control wizard.

  You can also use annotations to turn off paging for specific collections. For more information, see Section 3.4.4, "How to Specify Access Mode for Individual Objects in the Data Control."

- Custom pagination. If the built-in pagination options do not suit your needs, you can implement your own pagination by implementing a custom handler class. For more information, see Section 4.4.2, "How to Implement a Custom Handler for Querying and Pagination."

For more information about access mode and data control handlers, see Section 3.4.3, "What You May Need to Know About the Scrollable and Range Paging Modes."

3.4.1 How to Change Paging Mode for a Data Control

If you want to change the paging mode for a data control, you can do so in the Data Controls panel.

Note: For data controls using the `oracle.adf.model.adapter.bean.DataFilterHandler` or`oracle.adf.model.adapter.bean.jpa.JPQLDataFilterHandler` handler, the default access mode is `scrollable`.

Before you begin:

It may be helpful to have a general understanding of access modes for EJB and bean data controls. For more information, see Section 3.4, "Paginated Fetching of Data in EJB Data Controls."

You may also find it helpful to understand general data control features and functionality that you may need to use in the application with the data controls. For more information, see Section 3.1.2, "Additional Functionality for EJB Data Controls."

You need to complete this task:

Create an EJB or bean data control. For more information, see Section 3.3.1, "How to Create EJB Data Controls."

To change paging mode for a data control:

1. In the Data Controls panel, right-click the data control's node and choose Edit Definition.
2. In the ejb-definition Properties or the bean-definition Properties dialog, select `rangePaging` or `scrollable` from the **AccessMode** dropdown list.
3. If you are changing the data control to use range paging, make sure that the data control's `FactoryClass` property is specified as `oracle.adf.model.adapter.bean.BeanDCFactoryImpl`.

   You can access the `FactoryClass` property in the source editor for the `DataControls.dcx` file or in the Properties window that appears when you open `DataControls.dcx` in the source editor or overview editor.

3.4.2 How to Set Range Size for a Data Control that Uses Range Paging

When you set a data control's access mode to `rangePaging`, the data control determines the range size by reading the `rangeSize` property of the accessor iterator of each component that is bound to a collection in the data control.

To set the range size for a component:

1. In the Applications window, select the page containing the component that is bound to the data control.
2. In the Structure window, select the component that is bound to the data control collection.
3. In the Properties window, expand the Behavior node, and set the `rangeSize` property to the desired value.

For more information on iterator bindings, see "Iterator Bindings Created in the Page Definition File" in *Developing Fusion Web Applications with Oracle Application Development Framework*.

3.4.3 What You May Need to Know About the Scrollable and Range Paging Modes

Data controls that support scrollable and range paging modes rely on methods in the bean class to implement that functionality. The method that the data control uses depends on the data control handler class that the data control uses.

For JPA-based data controls, typically the `JPQLDataFilterHandler` handler is specified. `JPQLDataFilterHandler` relies on the presence of JPA queries and a `queryByRange()` method in the bean. For more information, see Section 3.2.4, "EJB Data Control Prerequisites and Considerations."

For non-JPA bean data controls (and for EJB and JPA-based bean data controls that do not have a `queryByRange()` method), `DataFilterHandler` is specified. To implement range paging in data controls that use this handler, you need to add code to your bean class as shown in Section 4.4.1, "How To Manually Implement Pagination Support in a Data Control."

For data controls that do not have either of these handler classes (such as EJB data controls where you have explicitly turned off named criteria support), there is no built-in support for scrollable or range paging. However, you can write your own handler class to implement paging support. For more information, see Section 4.4.2, "How to Implement a Custom Handler for Querying and Pagination."

3.4.4 How to Specify Access Mode for Individual Objects in the Data Control

If your data control encompasses multiple collections of different sizes, you may wish to set different access modes for some of the collections. You can do so by placing annotations on the accessor methods in the bean that the data control represents.

For the methods on which the annotations are used, the annotations override the access mode set for the data control. If an accessor method does not have such an annotation, it inherits its access mode from the one that is defined for the data control.

To specify access mode for individual objects in a bean or EJB data control:

1. Open the class on which the data control is based.
2. Add annotations for the methods for which you want a different access mode than that generally specified for the data control.

> **Note:** These annotations only work on getter methods.

Example 3–3 shows the necessary import statements and the available annotations and how they can be used on a collection.

Example 3–3 Access Mode Annotations

```
import oracle.adf.model.adapter.bean.annotation.AccessMode;
import oracle.adf.model.adapter.bean.annotation.AccessModeType;

...
   * List with scrollable access
   */
  @AccessMode(type=AccessModeType.SCROLLABLE)
  public List<Employees> getEmployeesScrollable() {
 ...
   * List with range paging.
   */
  @AccessMode(type=AccessModeType.RANGE_PAGING)
```

```
  public List<Employees> getEmployeesRangePaging() {
 ...
   * List with no paging.
   */
  @AccessMode(type=AccessModeType.NO_PAGING)
  public List<Employees> getEmployeesNoPaging() {
...
```

Note: You must place the annotation on the class that is specified by the `Definition` attribute of the `AdapterDataControl` element in the `DataControls.dcx` file. By default, this is the bean implementation class.

In the 12.1.2 release of Oracle ADF, the `Definition` attribute was set to a session bean's business interface (remote or local) by default, meaning that the annotations needed to be placed on the business interface instead of the bean implementation class. If you would like to retain that behavior in this release, you can change the value of the `Definition` attribute so that it points to the business interface.

3.4.5 What You May Need to Know About Sorting Tables Based on Range Paginated Collections

By default, if a user sorts a table that is bound to a JPA-based data control, the ADF Model runtime forces the iterator to return all rows into memory for sorting, even if the back-end JPQL queries have already done the sort at the database level, which can cause memory problems if collection is too large. If you are using range paging for a collection, you can disable the ADF Model runtime full in-memory sort and have the data control handle it instead, based on just the currently selected range.

To use the data control to handle the sort for range paginated collections:

1. In the Applications window, double-click the `DataControls.dcx` file to open it in the overview editor.
2. In the overview editor, select the node for the data control that you want to edit.
3. In the Properties window, set the `ImplementsSort` property to true.

3.5 Providing UI Hints for Attributes Using Annotations

After you create a data control for an EJB session bean or a JavaBeans component, you can use Java annotations to create declarative metadata for data object attributes to provide defaults for how the attributes are displayed in UI components. These *UI hints* can then be used by UI components to automatically display the queried information to the user in a consistent, locale-sensitive way. For example, you can use UI hints to provide defaults for label and tooltip text when a UI component is created from the data control attribute corresponding to the field returned by that getter.

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. When you use the Data Controls panel to create UI components based on attributes that are annotated with these UI hints, the EL expressions for accessing the hints are generated in the UI component code. For more information on the syntax for these EL expressions, see "How to Access UI Hints Using EL Expressions" in *Developing Fusion Web Applications with Oracle Application Development Framework*.

Using annotations, you can set the following UI hints:

- `label` - the component's label.
- `tooltip` - the component's tooltip.
- `display` - whether or not to display the attribute in forms and tables for the collection the UI. If the attribute is set to `false`, the attribute does not appear among the collection's attributes in the Data Controls panel.
- `width` - component width, in pixels.
- `height` - component height, in pixels.
- `autoSubmit` - whether or not the component will automatically submit when an appropriate action takes place (a click, text change, etc.).
- `controlType` - the control type the client UI will use to display the attribute. The default value for this hint is `ControlHintType.DEFAULT`, which is interpreted by the client to select the most appropriate component depending on the Java type of the attribute. You can keep this default value for Java-typed attributes. (Other possible values for this hint are only relevant if the attribute is based on a non-Java type.)
- `formatType` - a class that defines the kind of formatting that will be applied to the attribute (such as date, currency, or percentage).
- `format` - a format mask to determine specifically how a numeric or date value is displayed. For example, for a date, it could specify `dd/MM/YYYY` to indicate that the date is shown with day of the month first, month second, and year last and that each part is delimited by a slash (/).
- `timezoneId` - for attributes of type `Date`, can be used to specify a fixed time zone to be used. If this UI hint is not set, the time zone is determined by the user's browser (or other client that displays the UI).

The UI hints that you can set through annotations comprise a subset of the metadata that you can set on data controls declaratively using XML data control structure files. Using data control structure files, you can also set declarative validation rules, the default value for the attribute, and other metadata that for the object as a whole. For more information on all of the ways you can configure a data control through data control structure files, see Chapter 7, "Adding Business Logic to Data Controls."

> **Note:** If you set UI hints for an attribute both using annotations and in an XML data control structure file, the settings in the data control structure file take precedence.

There are three high-level annotations for providing UI hints for bean classes:

- `oracle.adf.model.adapter.bean.annotation.AttributeHint` - enables you to specify default label, tooltip, height, and width for the field. In addition, you can specify whether the attribute is displayed or hidden, whether the field is submitted automatically when a user completes entry, and what type of component should be generated for the field.
- `oracle.adf.model.adapter.bean.annotation.DateFormatter` - enables you to specify a format mask and a time zone for a `Date` attribute. You can use the `format` element to set a format mask, such as `MM/dd/YYYY` See the API documentation for the `java.text.SimpleDateFormat` at `http://docs.oracle.com/javase/7/docs/api/java/text/SimpleDateFormat.html` for information on the syntax for the date and time pattern strings that you can use for the format mask.

- `oracle.adf.model.adapter.bean.annotation.Formatter` - enables you to specify the formatting for other numeric types such as `oracle.jbo.format.DefaultCurrencyFormatter` and `oracle.jbo.format.PercentageFormatter`.

In addition, there are the following support classes that enumerate possible values for the above annotations and which might need to be imported into your class when using the above annotations.

- `oracle.adf.model.adapter.bean.annotation.ControlHintType` - provides a list of component types that you can reference from the `AttributeHint` annotation's `controlType` element.
- `oracle.adf.model.adapter.bean.annotation.FormatterType` - provides a list of formatter types that are referenced by the `DateFormatter` and `Formatter` annotations.
- `oracle.adf.model.adapter.bean.annotation.TimeZoneID` - provides a list of time zones that you can reference from the `DateFormatter` annotation.

> **Note:** When adding annotations for attribute hints to a method, you can press use JDeveloper's completion insight feature to help fill in the values. If the completion insight popup does not open automatically as you are typing, press Ctrl-Space.

See Example 3–4 for an example of how you might use UI hint annotations on a getter method in an entity bean class. Note that the annotations become available only after you generate the data control or after you add the `ADF Model Generic Runtime` library to the EJB project.

Example 3–4 Use of AttributeHint and DateFormatter Annotations on a Method

```
import oracle.adf.model.adapter.bean.annotation.AttributeHint;
import oracle.adf.model.adapter.bean.annotation.ControlHintType;
import oracle.adf.model.adapter.bean.annotation.DateFormatter;
import oracle.adf.model.adapter.bean.annotation.FormatterType;
import oracle.adf.model.adapter.bean.annotation.TimeZoneID
...
    @AttributeHint (
        label = "Hire Date",
        tooltip = "Type date in the form MM/dd/YYYY",
        display = true,
        controlType = ControlHintType.DEFAULT,
        width = 40,
        height = 20,
        autoSubmit = true)
    @DateFormatter (
        type = FormatterType.SIMPLE_DATE,
        format = "MM/dd/YYYY",
        formatter = "",
        timezoneId = TimeZoneID.DEFAULT)
    public Date getHireDate() {
        return hireDate;
    }
```

> **Note:** The `formatter` element of the `@Formatter` and `@DateFormatter` annotations is an advanced option that enables you to specify a separate handler class for providing the date or number format. For example, you could use this element to provide a handler that extends `oracle.jbo.format.DefaultDateFormatter`.

Example 3–5 shows how you might use the `@Formatter` annotation to specify that an attribute be displayed with currency formatting.

Example 3–5 Use of Formatter Annotation

```
import oracle.model.adapter.bean.annotation.Formatter;
import oracle.model.adapter.bean.annotation.FormatterType;
...
    @Formatter (type = FormatterType.CURRENCY)
    public Integer getMinSalary() {
        return minSalary;
    }
```

The following formatters can be assigned through the `@Formatter type` element.

- `BIGDECIMAL` - type defined by `oracle.jbo.format.DefaultBigDecimalFormatter`

  `CURRENCY` type defined by `oracle.jbo.format.DefaultCurrencyFormatter`

  `DATE` - `oracle.jbo.format.DefaultDateFormatter`

  `PERCENTAGE` - type defined by `oracle.jbo.format.PercentageFormatter`

  `NUMBER` - type defined by `oracle.jbo.format.DefaultNumberFormatter`

> **Note:** When you apply annotations for UI hints, you can not see the affect of the hints in the design-time view of pages that you create based on the data controls. However, you can test and verify the hints using the ADF Model Tester. For more information on using the tester, see Section 7.8, "Testing Data Object Metadata Using the Oracle ADF Model Tester."

3.6 Enabling Failover in an EJB Data Control

You can configure EJB data controls that are based on stateful session beans to have their state managed by the ADF Model runtime and enable failover of the objects encapsulated by the data control. Failover support for EJB data controls works in the same way as it does for bean data controls.

For information on generating method stubs in the session bean for this failover support, see Section 3.3.1, "How to Create EJB Data Controls." For more information on implementing those methods, see Section 4.5, "Enabling Failover in a Bean Data Control."

4

Creating and Configuring Bean Data Controls

This chapter describes how to create data controls for JavaBeans components that are based on the Java Persistence API (JPA) and plain Java objects (also known as "POJOs").

This chapter includes the following sections:

4.1 About Bean Data Controls

Data controls are an abstraction provided by ADF Model than enable you to work with data and business services in a declarative manner and easily create UI components. For more general information on data controls, see Section 1.1, "About ADF Model."

You use the bean data control type to create data controls for plain Java objects (POJOs) and JPA-based Java service facade classes.

4.1.1 About JPA-Based Bean Data Controls

Similar to EJB data controls, JPA-based bean data controls enable you to work with persistent data. Such data controls are based on POJO service facade classes that provide accessor methods to JPA entity classes. The main difference in JPA-based bean data controls is that you must provide the code to manage the persistence instead of being able to rely on an EJB container. For more information on EJB data controls, see Section 3.1, "About EJB Data Controls."

4.1.2 About non-JPA Bean Data Controls

You can also create bean data controls for Java classes that do not work with JPA functions. Such data controls do not provide persistence functionality or named criteria support, but they do include other features of adapter data controls such as the ability to add declarative validation rules on attributes and support for UI hints.

Unlike JPA-based bean data controls, non-JPA data controls do not have built-in implementations for handling transactions or updating of data sources. However, you can code your own implementation for that functionality in your bean class within specific methods whose signatures the data control recognizes. For more information, see Section 4.7, "Adding Transactional Behavior to a non-JPA Bean Data Control" and Section 4.6, "Enabling Custom CRUD Operations in a Bean Data Control."

Non-JPA data controls have support for scrollable paging and range paging, but require some manual coding. For more information, see Section 4.4.1, "How To Manually Implement Pagination Support in a Data Control."

You can also create a custom data control type that extends bean data controls. For more information, see Section 4.9, "Creating Custom Bean Data Controls."

4.1.3 Additional Functionality for Bean Data Controls

You may find it helpful to understand other ADF and JDeveloper features before you implement your data controls. Following are links to other sections that may be useful.

General data control features: Before beginning work with EJB data controls, it is important to understand the broader data control concepts. For more information, see Chapter 2, "Using ADF Data Controls."

ADF Model and data binding: When you create forms in an ADF web application, you use ADF Model and data binding. For more information, see "Using ADF Model in a Fusion Web Application" in *Developing Fusion Web Applications with Oracle Application Development Framework* and *Java API Reference for Oracle ADF Model*.

ADF Faces: When you create databound UI components, they are likely to be from the ADF Faces component set. For more information, see "Creating a Databound Web User Interface" in *Developing Fusion Web Applications with Oracle Application Development Framework*.

ADF task flows: Task flows extend JSF page flows to provide a modular and transaction-aware approach to navigation and application control. For more information, see "Creating ADF Task Flows" in *Developing Fusion Web Applications with Oracle Application Development Framework*.

4.2 Preparing a Bean to Expose with a Data Control

A bean data control serves as a metadata wrapper for a bean class and exposes the bean's code elements as data control objects, which can then be used to bind those code elements to UI components.

4.2.1 Supported Types and Constructs in Bean Data Controls

Bean data controls support the same Java types and constructs as EJB data controls. For more information, see Section 3.2.1, "Supported Types and Constructs in EJB Data Controls."

4.2.2 Bean Data Control Objects

When you create a data control based on a bean, the data control exposes several different types of objects, each of which you can bind to a variety of UI components. Bean data controls exposes the same types of objects as EJB data controls. For more information, see Section 3.2.2, "EJB Data Control Objects.".

4.2.3 Bean Data Control Prerequisites and Considerations

Bean data controls are based on classes that meet the JavaBeans specification. For example, for a class to be a valid data control source, it needs to have a public default constructor.

In order to take advantage of the full functionality of JPA-based data controls, you need to include some elements in your classes that the data controls can use to present the structure of the services. These elements are largely the same as for EJB data controls. The main difference is that you use a Java service facade class for bean data controls instead of an EJB session bean. For more information, see Section 3.2.4, "EJB Data Control Prerequisites and Considerations."

4.2.4 How to Create a Service Facade for a JPA-Based Bean Data Control

You can create a bean data control based on a POJO class that contains service methods to access Java entity classes. JDeveloper has a wizard that helps you create this facade class that contains all of the necessary methods to work with the data control.

Before you begin:

It may be helpful to have a general understanding of the code conventions that the service facade and the data control will rely on. For more information, see Section 4.2, "Preparing a Bean to Expose with a Data Control."

You may also find it helpful to understand general data control features and functionality that you may need to use in the application with the data controls. For more information, see Section 4.1.3, "Additional Functionality for Bean Data Controls."

You need to complete this task:

- Create an application workspace that contains a project with JPA entities as described in "How to Create JPA Entities" in *Developing Applications with Oracle JDeveloper*.

To create a service facade for a JPA-based Bean Data Control:

1. Right-click the package that contains the JPA entity classes and choose Java Service Facade.
2. In the Java Service Class page of the Create Java Service Facade wizard, do the following:
   - Specify a name for the service class.
   - Specify the persistence unit on which to base the service facade.
   - Select the **Implicit** or **Explicit** radio button to determine commit behavior.

     If you select **Explicit**, transactional methods are generated into the class, which are in turn exposed as `Commit` and `Rollback` operations if you later create a data control based on the class.

If you select **Implicit**, you can use the `persistEntity()`, `mergeEntity()`, and `removeEntity()` methods that are generated by the wizard to act as the methods for adding and removing rows from the data source.

3. In the Java Service Facade Methods page, select the methods which you want to generate.

   Note: If you subsequently plan to create a data control and you want to use the built-in pagination and JPA querying support, be sure to select the `queryByRange()` method.

4. Click **Finish** to exit the wizard.

4.3 Exposing Java Collections and Methods With Bean Data Controls

Once you have your bean in place, you can use JDeveloper to create data controls that provide the objects needed to declaratively bind UI components to the bean's services as well as any built-in operations provided by the data control.

4.3.1 How to Create a JPA-Based Bean Data Control

You create JPA-based bean data controls from within the New Gallery or the Applications window.

Before you begin:

It may be helpful to have a general understanding of bean data controls. For more information, see Section 4.3, "Exposing Java Collections and Methods With Bean Data Controls."

You may also find it helpful to understand general data control features and functionality that you may need to use in the application with the data controls. For more information, see Section 4.1.3, "Additional Functionality for Bean Data Controls."

In addition, you may find it helpful to understand the code patterns and constructs in your bean that the data control uses. For more information, see Section 4.2, "Preparing a Bean to Expose with a Data Control."

You need to complete this task:

Create a Java service facade class, as described in Section 4.2.4, "How to Create a Service Facade for a JPA-Based Bean Data Control."

To create a bean data control:

1. In the Applications window, right-click the bean for which you want to create a data control and choose **Create Data Control**.

   For a JPA-based data control, use the Java service facade class as the base for the data control.

2. In the Choose Bean Class page of the Create Bean Data Control wizard, specify a name for the data control instance.

   Note: You can create multiple data control instances with different behavior for the same bean. For more information, see Section 3.3.11, "How to Create Different Data Controls for a Single Bean."

3. In the Choose ADF Data Controls Features page, select any of the following checkboxes for additional data control features that you would like to use in your application. Methods will be added to the bean to implement the data control features.

   - **Transactions**. Selecting this feature generates the `commitTransaction()`, `rollbackTransaction()`, and `isTransactionDirty()` methods in your bean. If you have used JDeveloper to create a Java Service Facade with explicit commit behavior, these methods should already be implemented. For more information, see Section 4.7, "Adding Transactional Behavior to a non-JPA Bean Data Control."
   - **Custom CRUD**. This feature enables you to provide your own implementation of persistence behavior. JPA-based bean data controls already have CRUD functionality from JPA, so you would only select this feature if you want to override JPA's functionality. For more information, see Section 4.6, "Enabling Custom CRUD Operations in a Bean Data Control."
   - **Failover**. For more information, see Section 4.5, "Enabling Failover in a Bean Data Control."

4. Optionally, click the **New Wrapper Class** icon (to the right of the Implementation Class field) to generate a separate class that extends the bean class and holds the custom methods created on this page of the wizard.

5. In the Bean Data Control Options page, select any additional options.

   - **Access Mode**. Enables you to set how the data control fetches and stores data in memory. For more information, see Section 3.4, "Paginated Fetching of Data in EJB Data Controls."
   - **Support Named Criteria**. When selected, the data control includes built-in support for declarative named criteria, which can be used to create quick search forms. For more information, see Section 7.6, "Filtering Result Sets with Named Criteria." This option is only available for JPA-based data controls, on which it is selected by default.

     The Support Named Criteria option is only available for JPA-based beans that also contain a `queryByRange()` method. The data control uses the bean's `queryByRange()` method to handle all queries that otherwise would be carried out by individual getter methods on the bean. For more information, see Section 3.2.6, "What You May Need to Know About How EJB and Bean Data Controls Use Getter Methods."
   - **Generate Metadata**. You can select this option to immediately generate metadata files for any beans represented by the data control that are in the current project. This option is typically not necessary, since such metadata files are created on demand when you edit a data control. However, this option might be useful if you plan to make the application available for MDS customization. For more information, see Section 7.2.4, "What You May Need to Know About MDS Customization of Data Controls."

4.3.2 How to Create a non-JPA-Based Bean Data Control

You create bean data controls from within the New Gallery or the Applications window.

Before you begin:

It may be helpful to have a general understanding of bean data controls. For more information, see Section 4.3, "Exposing Java Collections and Methods With Bean Data Controls."

You may also find it helpful to understand general data control features and functionality that you may need to use in the application with the data controls. For more information, see Section 4.1.3, "Additional Functionality for Bean Data Controls."

In addition, you may find it helpful to understand the code patterns and constructs in your bean that the data control uses. For more information, see Section 4.2, "Preparing a Bean to Expose with a Data Control."

You need to complete these tasks:

1. Create an application workspace and a project. For more information, see "Creating Applications and Projects" in *Developing Applications with Oracle JDeveloper*.
2. Create a bean class in the project. For more information, see "How to Create a New Java Class or Interface" in *Developing Applications with Oracle JDeveloper*.

To create a non-JPA bean data control:

1. In the Applications window, right-click the bean for which you want to create a data control and choose **Create Data Control**.
2. In the Choose Bean Class page of the Create Bean Data Control wizard, specify a name for the data control instance.

   Note: You can create multiple data control instances with different behavior for the same bean. For more information, see Section 3.3.11, "How to Create Different Data Controls for a Single Bean."

3. In the Choose ADF Data Controls Features page, select any of the following checkboxes for additional data control features that you would like to use in your application. For each feature that you select, methods with signatures that are recognized by the data control will be added to the bean. Within those methods, you can write your implementation for the respective data control features.
   - **Transactions**. Selecting this feature generates the `commitTransaction()`, `rollbackTransaction()`, and `isTransactionDirty()` methods in your bean. For more information, see Section 4.7, "Adding Transactional Behavior to a non-JPA Bean Data Control."
   - **Custom CRUD**. This feature enables you to provide your own implementation of persistence behavior. For more information, see Section 4.6, "Enabling Custom CRUD Operations in a Bean Data Control."
   - **Failover**. For more information, see Section 4.5, "Enabling Failover in a Bean Data Control."
4. Optionally, click the **New Wrapper Class** icon (to the right of the Implementation Class field) to generate a separate class that extends the bean class and holds the methods created on this page of the wizard.
5. In the Bean Data Control Options page, select any additional options.
   - **Access Mode**. Enables you to set how the data control fetches and stores data in memory. For more information, see Section 4.4, "Paginated Fetching of Data

in Bean Data Controls." The Support Named Criteria option must be selected for you to be able to select an access mode.

- **Generate Metadata**. You can select this option to automatically generate metadata files for all of the beans represented by the data control upon the creation of the data control. This option is not necessary, since metadata files are created on demand when you edit a data control. However, this option might be useful if you plan to make the application available for MDS customization. For more information, see Section 7.2.4, "What You May Need to Know About MDS Customization of Data Controls."

4.3.3 What Happens in Your Project When You Create a Bean Data Control

When you create a data control based on a JavaBeans component, JDeveloper does the following:

- Creates the data control definition file (`DataControls.dcx`) and opens the file in the overview editor.
- Displays the hierarchy of the resulting data control objects in the Data Controls panel.
- If you have selected any features on the ADF Data Controls Features page, adds methods to implement those features to the session bean.

For general information on the overview editor and Data Controls panel, see Section 2.2.2, "What Happens in Your Project When You Create a Data Control." For information specific to JPA-based data controls, see Section 3.3.3, "How EJB and Bean Data Controls Appear in the IDE."

If you have selected any features on the ADF Data Controls Features page, methods to implement those features are added to the bean (or to the wrapper class if you have chosen to generate one).

If the bean has accessor methods that query collections and you have selected **Support Named Criteria** in the wizard, the data control will be generated to use `oracle.adf.model.adapter.bean.jpa.JPQLDataFilterHandler`, which provides support in the data control for declarative named criteria and paginated queries.

If the bean doesn't have accessor methods that query collections, the data control will be generated to use `oracle.adf.model.adapter.bean.DataFilterHandler`. Besides not supporting named criteria, `oracle.adf.model.adapter.bean.DataFilterHandler` also does not have paging functionality fully implemented. However, you can add a method to your bean to implement paging. For more information, see Section 4.4.1, "How To Manually Implement Pagination Support in a Data Control."

4.3.4 What You May Need to Know About Primary Keys for Non-JPA Bean Data Controls

Even when you are creating non-JPA beans, you can use the `@Id` annotation to mark a field as a primary key column.

However, for performance reasons, you might find it preferable to designate the primary key at the data control level so that you do not introduce a dependency on the JPA runtime library. For more information, see Section 7.3.1, "How to Designate an Attribute as Primary Key."

4.4 Paginated Fetching of Data in Bean Data Controls

When you create a bean data control, you can use the wizard's **Access Mode** dropdown list to determine how records are accessed from the database and whether to limit the number of records that are held in memory at a time.

Paging support for bean data controls generally follows the same principals as paging for EJB data controls. For more information, see Section 3.4, "Paginated Fetching of Data in EJB Data Controls." However, especially for non-JPA-based data controls, you might want or need to fill in part of the implementation yourself.

4.4.1 How To Manually Implement Pagination Support in a Data Control

With non-JPA data controls (or any EJB or bean data control that uses the `oracle.adf.model.adapter.bean.jpa.DataFilterHandler` handler), you need to add three methods for each collection in the session or service facade in order for the ADF Model runtime to implement scrollable paging and range paging. The method signatures should take the following form:

```
List<EntityBeanName> getEntityBeanNameList()
List<EntityBeanName> getEntityBeanNameList(int firstResult, int maxResults)
long getEntityBeanNameListSize()
```

> **Note:** JPA-based data controls typically use the `oracle.adf.model.adapter.bean.jpa.JPQLDataFilterHandler` handler and thus do not require manual implementation of these methods. However, a data control can get assigned the `oracle.adf.model.adapter.bean.jpa.DataFilterHandler` handler if the bean it is based on does not contain the `queryByRange(String jpqlStmt, int firstResult, int maxResults)` method. For more information, see Section 3.2.4.2, "Recommended Session Facade Elements."

4.4.2 How to Implement a Custom Handler for Querying and Pagination

If the built-in querying and paging options are not sufficient for your application, you can implement your own custom paging and querying behavior by providing your own data handler class for your data control.

Before you begin:

It may be helpful to have a general understanding of bean data controls. For more information, see Section 4.3, "Exposing Java Collections and Methods With Bean Data Controls."

You may also find it helpful to understand general data control features and functionality that you may need to use in the application with the data controls. For more information, see Section 4.1.3, "Additional Functionality for Bean Data Controls."

In addition, you may find it helpful to understand the code patterns and constructs in your bean that the data control uses. For more information, see Section 4.2, "Preparing a Bean to Expose with a Data Control."

You need to complete this task:

Create a bean data control or an EJB data control. For more information, see Section 4.3.1, "How to Create a JPA-Based Bean Data Control," Section 4.3.2, "How to Create a non-JPA-Based Bean Data Control," and Section 3.3.1, "How to Create EJB Data Controls."

Note: When you create the data control, it is not important which access mode you select. In the following procedure, you will manually override that access mode.

To implement a custom handler for querying and pagination:

1. Write a custom data control handler class and add it to the data control's project.

   You can sub-classes an existing handler, such as `oracle.adf.model.adapter.bean.jpa.JPQLDataFilterHandler` or `oracle.adf.model.adapter.bean.DataFilterHandler`. See Example 4–1 for an outline of a custom handler class.

2. In the Source view of the `DataControls.dcx` file, type the fully-qualified class name of the handler as the value for the `DataControlHandler` attribute of each data control.

   `DataControlHandler` is an attribute of the `ejb-definition` element of EJB data controls and an attribute of the `bean-definition` element for bean data controls.

Example 4–1 Custom Data Control Handler

```
public class MyJPQLDataFilterHandler  extends JPQLDataFilterHandler
{
 public boolean invoke(Map bindingContext,
                                 OperationBinding action,
                                 DataFilter filter)
 {
    /** TODO: Users provide custom criteria. */
 }

 public Object invoke(RowContext rowCtx, String name,
                              DataFilter filter)
 {
    /** TODO: Users provide custom criteria. */
 }

}
```

4.5 Enabling Failover in a Bean Data Control

You can configure bean data controls to have their state managed by the ADF runtime. This can be particularly useful if your application will run in a high availability or cluster environment. If the application is running on a server node that fails, the most recent snapshot of the data control's session state can be restored to another node in the cluster. Likewise, if a user's session is interrupted, the failover support can be used to restore the data control's state when the session is resumed.

You enable failover by implementing methods for saving and restoring the state of the data control. The ADF Model runtime manages the calling of these methods and handles the details of distributing the states of the data control. EJB and bean data control support for failover requires the following three methods, which you can add to your bean class through the data control wizard or manually:

- `public Serializable createSnapshot()`. Saves the state of the data control and underlying bean as a serialized object and returns a handle for that object. The ADF runtime calls this method whenever the user submits any changes to a

component bound via the data control (or when the user merely refreshes the page).

In this method you need to add the logic that determines what gets saved and return the corresponding `Serializable` handle.

- `public void restoreSnapshot(Serializable handle)`. Restores the session state based on the most recently created snapshot. This method is called whenever the ADF Model runtime needs to restore a session that has been interrupted for whatever reason, such as a broken connection, a server failure, etc.

  In this method you need to fill in the implementation for restoring the data control from the handle returned by `createSnapshot()`.

- `public void removeSnapshot(Serializable handle)`. Removes the snapshot that is associated with the handle returned by the `createSnapshot()` method.

4.5.1 What You May Need to Know About Calling PageFlowScope from the Constructor

In bean data controls that implement failover, you should not call `PageFlowScope` from the bean's constructor. During application failover, the data control objects are re-instantiated with a call to the constructor. Because that call happens outside of the ADF lifecycle, `FacesContext` is not available, nor is anything that depends upon it for its implementation.

4.6 Enabling Custom CRUD Operations in a Bean Data Control

JPA-based data controls are tightly integrated with the CRUD (create/read/update/delete) support in JPA. For example, built-in data control operations such as `Create` and `Delete` use the corresponding JPA persistence methods to carry out those operations when invoked at runtime.

However, if your data control is not JPA-based, you can provide your own implementation to integrate your bean's CRUD model with the data control.

The following are the method stubs that are added to your bean data control class when you use the Create Bean Data Control wizard to create a bean data control and enable custom CRUD support:

- `public Object createRowData(RowContext p0)`. Used to create a new row of data. At runtime, this method is called when the `Create` operation is invoked. You can provide your own implementation of this method to customize how a new row is persisted.

- `public boolean removeRowData(RowContext p0)`. Used to remove a row of data. At runtime, this method is called when the `Delete` operation is invoked.

- `public boolean setAttributeValue(oracle.binding.AttributeContext p0, Object p1)`. This method is called by the ADF Model runtime when a new value is to be set on an attribute in a bean.

- `public Object registerDataProvider(oracle.binding.RowContext p0)`. This method is called by the data binding facility before the object is modified or marked as removed, so the row can be marked dirty by the data control.

  This method needs to be present if you are implementing custom CRUD functionality, though typically you do not need to provide your own implementation.

- `public void validate()`. Validates a transaction if it is dirty.

This method needs to be present if you are implementing custom CRUD functionality, though typically you do not need to provide your own implementation of this method.

For information on creating a non-JPA data control, see Section 4.3.2, "How to Create a non-JPA-Based Bean Data Control."

4.7 Adding Transactional Behavior to a non-JPA Bean Data Control

When you create a JPA-based bean with the Create Java Service Facade wizard, you can generate code to support an explicit commit model. When a data control is based on such a class, the bean's transactional methods are exposed as `Commit` and `Rollback` operations in the data control.

For non-JPA bean data controls, you can generate stub methods when you create the data control to support the `Commit` and `Rollback` operations. Within those stub methods, you can write your own implementation of transactional behavior. At runtime, when the `Commit` and `Rollback` operations are invoked those methods are called.

The following are the method stubs that are added to your bean data control class when you use the Create Bean Data Control wizard to create a bean data control and enable transactional support:

- `public boolean isTransactionDirty()`. Used to mark whether there are any pending changes to be committed (or rolled back).
- `public void commitTransaction()`. Commits all pending changes to the data source.
- `public void rollbackTransaction()`. Rolls back pending changes.

Note: In order to implement transaction support in a non-JPA data control, you must also implement custom CRUD methods. For more information, see Section 4.6, "Enabling Custom CRUD Operations in a Bean Data Control"

For information on creating a non-JPA data control, see Section 4.3.2, "How to Create a non-JPA-Based Bean Data Control."

4.8 Using Annotations to Declare Metadata for Bean Data Controls

The Adapter Data Control Framework provides several Java annotations that you can use to specify metadata for bean and EJB data controls.

- `oracle.adf.model.adapter.bean.annotation.AttributeHint`, `oracle.adf.model.adapter.bean.annotation.DateFormatter`, and `oracle.adf.model.adapter.bean.annotation.Formatter`. Use to provide UI hints for a data control without having to create XML data control structure files. For more information, see Section 3.5, "Providing UI Hints for Attributes Using Annotations."
- `oracle.adf.model.adapter.bean.annotation.AccessMode`. Use to override the `AccessMode` flag in the `DataControls.dcx` file. You can use this annotation to control the paging feature in a JPA-based data control at the level of the accessor method, thus making it possible to use different access modes in a single data control.

```
// AccessModeType can be either NO_PAGING, RANGE_PAGING or
SCROLLABLE)@AccessMode(type = AcessModeType.RANGE_PAGING)
public Collection<EmpBean> getListEmpBean() {}
public Collection<EmpBean> getListEmpBean(int firstResult, int maxResults) {}
public int getListEmpBeanSize() {}
}
```

For more information, see Section 3.4.4, "How to Specify Access Mode for Individual Objects in the Data Control."

- `oracle.adf.model.adapter.bean.annotation.AccessModeType`. Must also be imported when using the `oracle.adf.model.adapter.bean.annotation.AccessMode` annotations in a class.
- `oracle.adf.model.adapter.bean.annotation.Property`. Use to define a custom property for an attribute. For example:

```
@Property(name = "myProp1", value = "myProp1Value")
public Long getEmpno() {}
```

- `oracle.adf.model.adapter.bean.annotation.Properties`. Use to specify multiple properties for an attribute. For example:

```
@Properties( {
  @Property(name = "myProp1", value = "myProp1Value"),
  @Property(name = "myProp2", value = "myProp2Value"),
  })
public Long getEmpno() {}
```

- `oracle.adf.model.adapter.bean.annotation.ElementType`. Use to define the type for the collection. For example, an element type of `EmpBean` can be defined with `@ElementType(name = "model.EmpBean"`. This is a substitute for using a generic collection (for example, `Collection<EmpBean>`) and is recommended only if you are using JDK 1.4 or before or if you need to override the element type of the collection.

```
@ElementType(name = "model.EmpBean")
public Collection getListEmpBean() {}
```

- `oracle.adf.model.adapter.bean.annotation.Id`. Use to define the primary key for the attribute. For example:

```
@Id
public int getPrimaryKey() {}
```

  This annotation should not be used for JPA-based data controls that encompass entity classes that use the `javax.persistence.Id` annotation.

4.9 Creating Custom Bean Data Controls

You can modify or enhance the functionality on the bean data control type by extending the definition and implementation classes for the bean data control type to create a custom data control.

Creating a custom bean-based data control consists of a combination of the following main steps:

- (Optionally) extending `oracle.adf.model.adapter.bean.BeanDataControl` in order to customize the way the data control handles filtering, failover, updating data sources, and other data control features.
- (Optionally) extending `oracle.adf.model.adapter.bean.BeanDCDefinition` to create a new implementation class.

If you create a new implementation class, you need to register that implementation class in the `DataControls.dcx` file. In the source view of the file, replace the value of the data control's `ImplDef` property (`oracle.adf.model.adapter.bean.BeanDataControl`) with the fully-qualified name of the new implementation class.

- (Optionally) creating new structure definition types for the types of data structures that will be represented by your data control.
- (Optionally) defining a configuration class to hold the metadata that is provided in the data control's `DataControls.dcx` file. For example, you can extend `oracle.adf.model.adapter.bean.BeanDCConfiguration`, which is the configuration class for standard bean data controls.

  Then, in the data control implementation class, you need to override the `getConfiguration()` method that is inherited from `oracle.adf.model.adapter.AbstractDefinition` and cast its return to the new configuration class.
- (Optionally), in the data control class, implementing some of the existing supported features (such as failover or custom CRUD support).

For API documentation of the classes used by the bean data control and other data control types, see the*Java API Reference for Oracle ADF Model*.

5

Exposing Web Services Using the ADF Model Layer

This chapter describes how to create ADF data controls for SOAP and REST web services so that you can better use those services in the user interface.

This chapter includes the following sections:

- Section 5.1, "About Web Service Data Controls in ADF Applications"
- Section 5.2, "Creating Web Service Data Controls"
- Section 5.3, "Securing Web Service Data Controls"

5.1 About Web Service Data Controls in ADF Applications

Web services allow enterprises to expose business functionality irrespective of the platform or language of the originating application because the business functionality is exposed in such a way that it is abstracted to a message composed of standard XML constructs that can be recognized and used by other applications.

Web services are modular business services that can be easily integrated and reused, and it is this that makes them ideally suited as components within SOA. JDeveloper helps you to create top-down web services (services created starting from a WSDL), bottom-up web services (created from the underlying implementation such as a Java class or a PL/SQL stored procedure in a database), and services created from existing functionality, such as exposing an application module as a service.

5.1.1 Web Service Data Control Use Cases and Examples

You can consume web services in web applications, and common reasons for wanting to do so are:

- To add functionality which would be time-consuming to develop with the application, but which is readily available as a web service
- To access an application that runs on different architecture
- To access an application that is owned by another team when their application must be independently installed, upgraded, and maintained, especially when its data is not replicated locally (for example, when other methods of accessing their application cannot be used)

5.1.2 Additional Functionality for Web Service Data Controls in ADF Applications

You may find it helpful to understand other Oracle ADF features before you start working with web service data controls. In addition, it may be helpful to understand

support for developing web services in JDeveloper. Following are links to other functionality that may be of interest.

- 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 more general information on using data controls, see Chapter 2, "Using ADF Data Controls."
- If you are working behind a firewall and you want to use a web service that is outside the firewall, you must configure the web browser and proxy settings in JDeveloper, as described in "Setting Browser Proxy Information" in *Developing Fusion Web Applications with Oracle Application Development Framework*.
- If you are developing the web services that you will later expose through data controls, you can use JDeveloper features to simplify the process. For more information, see "Developing and Securing Web Services" in *Developing Applications with Oracle JDeveloper*.

The following chapters provide information about specific objects you can use in data controls:

- For information about using collections on a data control to create forms, see "Creating a Basic Databound Page" in *Developing Fusion Web Applications with Oracle Application Development Framework*.
- For information about using collections to create tables, see "Creating ADF Databound Tables" in *Developing Fusion Web Applications with Oracle Application Development Framework*.
- For information about using master-detail relationships to create UI components, see "Displaying Master-Detail Data" in *Developing Fusion Web Applications with Oracle Application Development Framework*.
- For information about creating lists, see "Creating Databound Selection Lists and Shuttles" in *Developing Fusion Web Applications with Oracle Application Development Framework*.
- For information about creating graphs, charts, and other visualization UI components, see "Creating Databound Chart and Gauge Components" in *Developing Fusion Web Applications with Oracle Application Development Framework*.

5.2 Creating Web Service Data Controls

The most common way of using web services in an application developed using Oracle ADF is to create a data control for an external web service. A typical reason for doing this is to add functionality that is readily available as a web service, but which would be time consuming to develop with the application, or to access an application that runs on a different architecture.

Additionally, you can reuse components created by Oracle ADF to make them available as web services for other applications to access.

5.2.1 How to Create a Data Control for a SOAP-based Web Service

JDeveloper allows you to create a data control for an existing web service using just the WSDL for the service. You can browse to a WSDL on the local file system, locate one in a UDDI registry, or enter the WSDL URL directly.

Note: If you are working behind a firewall and you want to use a web service that is outside the firewall, you must configure the web browser and proxy settings in JDeveloper. For more information, see "Setting Browser Proxy Information" in *Developing Fusion Web Applications with Oracle Application Development Framework*.

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 Section 5.2, "Creating Web Service Data Controls."

You may also find it helpful to understand additional functionality that can be added using other web services features. For more information, see Section 5.1.2, "Additional Functionality for Web Service Data Controls in ADF Applications."

You will need to complete this task:

Create an application workspace and a project in that workspace. Depending on how you are organizing 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*.

To create a data control for a SOAP-based web service:

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, expand **Business Tier**, select **Web Services** and then **Web Service Data Control (SOAP/REST)**, and click **OK**.
3. In the Create Web Service Data Control wizard, on the Data Source page, type a name for the data control, select the **SOAP** radio button, enter a WSDL URL, and the specify the specific web service to be accessed by the data control.
4. On the Data Control Operations page, shuttle the operations that you want the data control to support to the Selected panel.

   Optionally, select the **Include HTTP Header Parameter** checkbox. For more information, see Section 5.2.3, "How to Include a Header Parameter for a Web Service Data Control."
5. On the Response Format page, specify the format of the SOAP response.
6. On the Endpoint Authentication page, specify the authentication details for the endpoint URL, and click **Finish**.

5.2.2 How to Create a Data Control for a RESTful Web Service

JDeveloper allows you to create a data control for a REST web service using a connection to the web service and schema for the methods that you want to invoke.

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 Section 5.1, "About Web Service Data Controls in ADF Applications."

You may also find it helpful to understand additional functionality that can be added using other web services features. For more information, see Section 5.1.2, "Additional Functionality for Web Service Data Controls in ADF Applications."

You will need to complete these tasks:

- Create an application workspace and a project in that workspace. Depending on how you are organizing 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*.
- If you have any local XSD files that you need to include to describe response or payload formats, copy those to the project's resource path as described in Section 5.2.9, "What You May Need to Know About Making an XML Schema Available to a REST Data Control."

To create a data control for a RESTful web service:

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, expand **Business Tier**, select **Web Services** and then **Web Service Data Control (SOAP/REST)**, and 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.
4. In the Connection field, select the URL connection to use.

   If you have not established a URL connection, click the **Create a new URL connection** icon to open the Create URL Connection dialog. In that dialog, enter a name for the connection and a base URL. Do not include any resources or parameters in the URL.

   Note: In the Create URL Connection dialog, you can click **Test Connection** to verify that you can connect to the URL. However, the URL's server may be configured to not accept requests on the base URL, meaning that the test will fail. Regardless of that fact, you can click **OK** to create the connection.

   If you have such a base URL 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**.

   ---

5. On the Resources page, specify a resource for the connection by completing the following sub-steps for each resource:

   a. Click the **Add** button to add a resource path.

   b. Type the name of the resource path inline.

      As part of the resource path you can also enter path parameters.

      Enter any path parameters in the form `##paramName##`. For example, if the web service to be accessed supplies stock quotes, the full URL for the resource for one of the stocks is `http://www.example.com/quotes/ACOMPANY`, and the path parameter name is `ticker`, you would enter the following as the resource path:`/quotes/##ticker##`.

Note: You can also use a parameter to provide dynamic input for the source path (for example, `/##servicename##/##ticker##`) where the user would be expected to also provide the service name (such as `quote`).

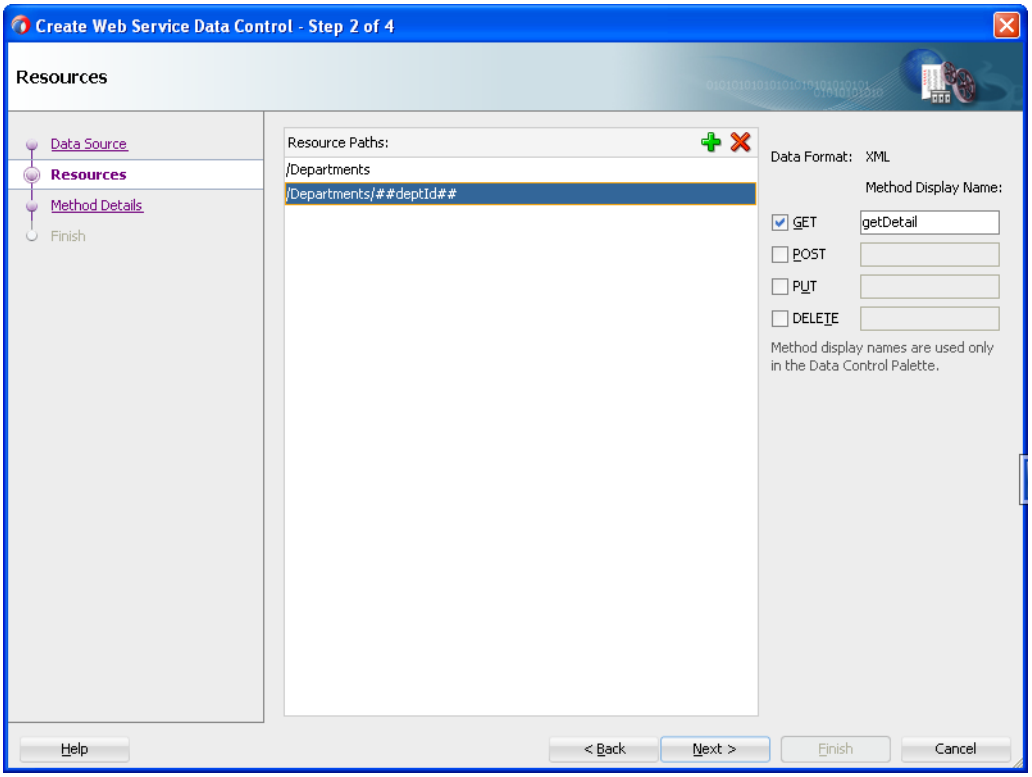
If you wish to create any resources with query parameters (i.e. parameters that take the form `?ParamName=ParamValue`), you can specify those parameters on the next page of the wizard.

c. In the right side of the dialog, select one of the checkboxes to specify a REST method for the resource path and then type a name that you can use to identify that method in the Data Controls panel.

6. Repeat step 5 for each resource you would like to include in the data control.

   Figure 5–1 shows the Resources page of the wizard with two resources entered, the second of which includes a path parameter. As shown in the figure, the second resource is selected and a GET method is specified for it.

Figure 5–1 Resources Page of Create Web Service Data Control Wizard



7. On the Method Details page, select a method and specify an XSD that provides the response format for that method.

   If the XSD files that you are providing are local, you first need to place them in the project's resource path as described in Section 5.2.9, "What You May Need to Know About Making an XML Schema Available to a REST Data Control."

   If you do not have a schema for the response format, you can create one yourself by recreating the XML for the resource based on its documentation and then creating an XSD from that XML file. In JDeveloper, you can generate an XSD file

from an XML document by choosing **File > New > From Gallery > XML > XML Schema from XML Document**.

> **Tip:** You can also use an XSL style sheet to define custom elements based on the response XML to include in the data control. When you use this approach, the schema that you specify in the **Response XSD** field must be based on the style sheet that you specify in the **Response XSL** field. For more information, see Section 5.2.7, "How to Add Custom Attributes to a REST Web Service Data Control."

8. If the method is a `PUT` or `POST` method, specify a schema for the request payload in the Payload XSD field.
9. For the selected method, specify any URL parameters and default values.

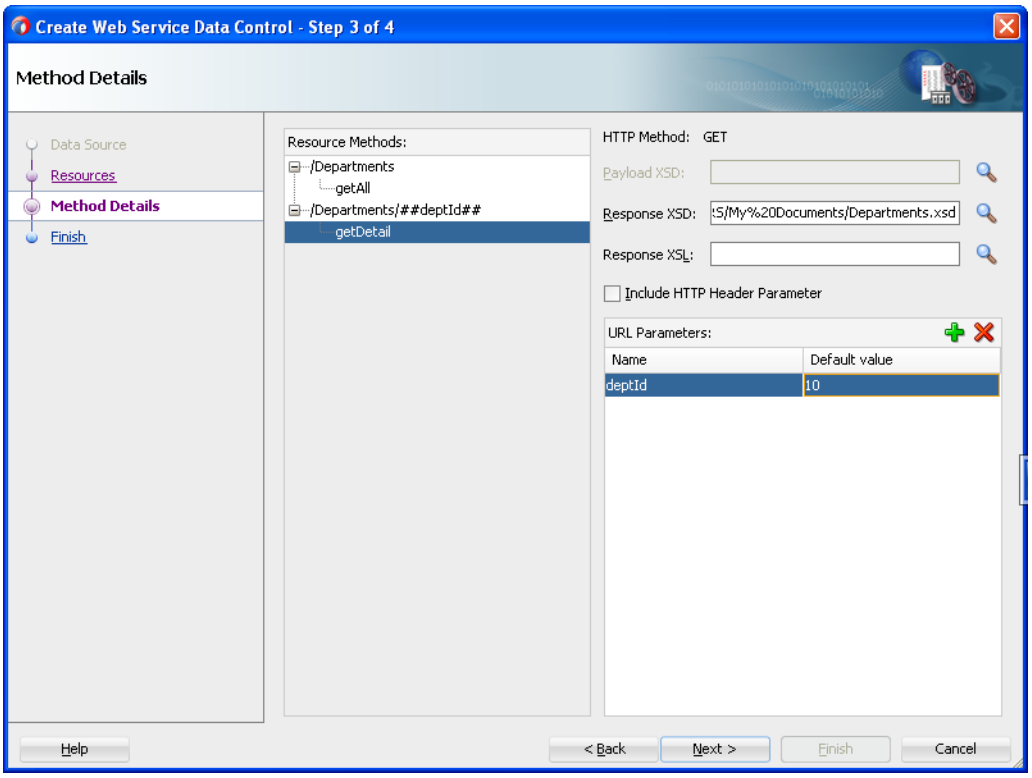
   For parameters that were included in the resource path, the parameter names are included in the URL Parameters list. For these parameters, you need to fill in a default value.

   For example, from the sample parameter shown in step 5, `ticker` would be the parameter name and `ACOMPANY` could be the default value.

   For path parameters of GET methods, it is not necessary to provide a default value if you have provided a response XSD for the method in step 7.
10. Optionally, select the Include HTTP Header Parameter checkbox. For more information, see Section 5.2.3, "How to Include a Header Parameter for a Web Service Data Control."
11. Repeat steps 7 through 10 for each method.

    Figure 5–2 shows the Method Details page with the `getDetails` method selected, a response XSD specified, and default value set for its `deptId` parameter.

Figure 5–2 Method Details Page of the Create Web Service Data Control Wizard



12. On the Finish page, review the details of the data control to be generated, and click **Finish**.

5.2.3 How to Include a Header Parameter for a Web Service Data Control

When using a web service data control, you may want to add a custom parameter to the HTTP header when invoking the HTTP request. Such a parameter can be useful for a variety of purposes, including for security or for notifications. For example, you may want to add an enterprise ID to the HTTP header when invoking the request. This enterprise ID in the request allows the web service data control to specify which cloud service the request will be directed to.

To configure the web service data control to use a header parameter, you select **Include Http Header Parameter** in the Create Web Service Data Control wizard. For SOAP-based web service data controls, this is on the Data Control Operations page. For REST-based web service data controls, it is on the Method Details page.

After creating the data control, you will be able to see the **HttpHeader** parameter in the Data Controls panel under the **Parameters** node of the web service data control's methods. In addition, the `AdapterDataControl` element for the web service data control (in the `.dcx` file) contains an `<httpHeaders paramName="HttpHeader"/>` element.

To use the `HttpHeader` parameter, you will need to create a backing bean in the user interface project for the web service data control. The value for the `HttpHeader` parameter is provided through the backing bean. The backing bean must have a property of the type `Map` and the name/value pairs for the HTTP headers should be added to that property. Additionally, the `Map` must be of type `<String, List<String>>` or `<String,String>`, and you should expose the property with getter and setter methods, as shown Example 5–1.

Example 5–1 Backing Bean to Support Http Header Parameters in a Web Service Data Control

```
public class BackingBean {
  private Map<String,Object> httpHeadersMap = new HashMap<String,Object>();
  public BackingBean() {
    List<String> headersList = new ArrayList<String>();
    headersList.add("Oracle");
    httpHeadersMap.put("enterpriseID",headersList);
  }
  public void setHttpHeadersMap(Map<String,Object> httpHeadersMap) {
    this.httpHeadersMap = httpHeadersMap;
  }
  public Map<String,Object> getHttpHeadersMap() {
    return httpHeadersMap;
  }
}
```

When you drag and drop the operation from the Data Controls panel onto a page as an ADF Parameter Form, remove the **HttpHeader** from the list of fields. Then, in the Edit Action Binding dialog, under the **Parameters** section specify the value for **HttpHeader** parameter by providing an expression that points to the backing bean `Map` property.

5.2.4 How to Adjust the Endpoint for a SOAP Web Service Data Control

After developing a web service data control, you can modify the endpoint. This is useful, for example, when you migrate the application from a test environment to production.

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 Section 5.2, "Creating Web Service Data Controls."

You may also find it helpful to understand additional functionality that can be added using other web services features. For more information, see Section 5.1.2, "Additional Functionality for Web Service Data Controls in ADF Applications."

To change the endpoint for a web service data control:

1. In the Applications window, select the `DataControls.dcx` file for the web service data control.
2. In the Structure window, right-click the web service data control and choose **Edit Web Service Connection**.
3. In the Edit Web Service Connection dialog, make the necessary changes to the endpoint URL and port name.
4. Click **OK**.

5.2.5 How to Refresh a SOAP Web Service Data Control

After updating a SOAP-based web service data control, you might find that a web service operation has changed in its method signature, return type, or structure. When this happens, you can update the data control without having to re-create it.

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 Section 5.2, "Creating Web Service Data Controls."

You may also find it helpful to understand additional functionality that can be added using other web services features. For more information, see Section 5.1.2, "Additional Functionality for Web Service Data Controls in ADF Applications."

To refresh an operation in a SOAP-based web service data control:

1. In the Applications window, select the `DataControls.dcx` file for the web service data control.
2. In the Structure window, right-click the desired web service operation and choose **Update**.

JDeveloper queries the web service and updates the web service data control to reflect the current state of the selected operation.

5.2.6 What You May Need to Know About Primary Keys in SOAP Web Service Data Controls

When you create a data control on a SOAP-based web service, the data control supports primary key operations on any exposed collection.

If the web service definition references a schema that defines an element or attribute as type `xsd:ID`, the data control will expose the attribute as a key attribute and make the `setCurrentRowWithKey` and `setCurrentRowWithKeyValue` data control operations available for the collection.

For example, your schema could set the `deptno` attribute as the primary key using the `<xsd:attribute>` element as shown below:

```
<xsd:attribute name="deptno" type="xsd:ID" use="required"/>
```

Or the schema could set the `deptno` attribute as the primary key using the `<xsd:element>` element as shown below:

```
xsd:element name="deptno" type="xsd:ID"/>
```

> **Note:** The XSD entries shown above are generated at runtime if you have created the web service from a Java class and added the JAXB `@XmlID` annotation and either `@XmlAttribute(required=true)` or `@XmlElement(required=true)` to the field or the getter method representing the key.

If no `ID` is defined for a collection in one of the above ways, the data control creates a hidden attribute for the collection that serves as an index-based primary key. You can then use the `setCurrentRowWithKey` or `setCurrentRowWithKeyValue` data control operation to pass the index of the row.

Note: If you do not see the `setCurrentRowWithKey` or `setCurrentRowWithKeyValue` operations for a collection in the Data Controls panel, you may need to manually update the `DataControls.dcx` file to have those operations exposed. To do so, open the Source view of the `DataControls.dcx` and change the value of the service's `ensureKeyAttribute` property to `true`. Then, in the Data Controls panel, click the **Refresh** icon to refresh the list of operations.

5.2.7 How to Add Custom Attributes to a REST Web Service Data Control

Before you create a data control based on a RESTful web service, you can create an XSL style sheet to transform the content of the REST service into a custom XML format to incorporate into the data control. For example, you can create elements that are calculated from other elements returned by the service and have those elements exposed in the data control as attributes on which you can base UI components.

Example 5–2 shows a style sheet that defines `NAME` and `DEPTID` elements to be derived directly from the response XML. In addition, the style sheet defines the `CUSTOMNAME` and `UNIQUEID` elements, the values of which are calculated from the `NAME` and `DEPTID` elements. A data control created on the basis of this style sheet would expose all four of these elements as attributes.

Example 5–2 XSL Style Sheet for Customizing Format of a REST Response

```
<?xml version="1.0" encoding="windows-1252" ?>
<xsl:stylesheet version="2.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="DEPARTMENTS">
    <DEPARTMENTS>
      <xsl:for-each select="DEPT">
        <DEPT>
          <NAME><xsl:value-of select="NAME"/></NAME>
          <DEPTID><xsl:value-of select="DEPTID"/></DEPTID>
          <CUSTOMNAME><xsl:value-of select="concat('CUST_',NAME)"/></CUSTOMNAME>
          <UNIQUEID><xsl:value-of select="DEPTID * 2"/></UNIQUEID>
        </DEPT>
      </xsl:for-each>
    </DEPARTMENTS>
  </xsl:template>
</xsl:stylesheet>
```

After you create the style sheet, you need to create an XML schema that incorporates the custom elements. One way to simplify this process is to create an XML file based on the response format that includes the custom elements and then use JDeveloper's XML tools to generate a schema based on that XML file.

Example 5–3 shows an XML file that contains both the base and custom elements that are defined in the style sheet shown in Example 5–2.

Example 5–3 XML Response That Is Modified with Custom Elements

```
<?xml version="1.0" encoding="UTF-8"?>
    <DEPARTMENTS>
      <DEPT>
        <NAME>Marketing</NAME>
        <DEPTID>20</DEPTID>
        <CUSTOMNAME>CUST_Marketing</CUSTOMNAME>
        <UNIQUEID>40</UNIQUEID>
```

```
    </DEPT>
    <DEPT>
      <NAME>Marketing</NAME>
      <DEPTID>20</DEPTID>
      <CUSTOMNAME>CUST_Marketing</CUSTOMNAME>
      <UNIQUEID>40</UNIQUEID>
    </DEPT>
  </DEPARTMENTS>
```

Example 5–4 shows the schema based on the modified XML response.

Example 5–4 XML Schema File Containing Custom Elements

```
<?xml version="1.0" encoding="windows-1252" ?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns="http://www.example.org"
            targetNamespace="http://www.example.org"
elementFormDefault="qualified">
  <xsd:element name="DEPARTMENTS">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="DEPT" maxOccurs="unbounded">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="NAME" type="xsd:string"/>
              <xsd:element name="DEPTID" type="xsd:integer"/>
              <xsd:element name="CUSTOMNAME" type="xsd:string"/>
              <xsd:element name="UNIQUEID" type="xsd:integer"/>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

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 Section 5.2, "Creating Web Service Data Controls."

You may also find it helpful to understand additional functionality that can be added using other web services features. For more information, see Section 5.1.2, "Additional Functionality for Web Service Data Controls in ADF Applications."

To add custom attributes to a REST data control:

1. Create an XSL style sheet based on the structure of the data returned by the web service and including any custom elements that you want to incorporate.
2. Based on the format for the method's response XML, create an XML document that also includes the custom elements.

> **Note:** If a resource can nest multiple resources of a given type, make sure that the XML file includes two or more of those nested resources so that the schema generated in the next step reflects the one-to-many cardinality.

3. Create an XML schema based on the XML document created in the previous step.

   You can do this in JDeveloper by opening the New Gallery, and selecting **General > XML > XML Schema from XML Document**.

4. Create a data control as described in Section 5.2.2, "How to Create a Data Control for a RESTful Web Service."

   In step 7 of the procedure, specify both the Response XSL (the XSL style sheet you have created) and the Response XSD (the XML schema you have created).

5.2.8 What You May Need to Know About Web Service Data Controls

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 more information, see Section 2.3.1, "How to Use the Data Controls Panel."

In the Data Controls panel, each data control object is represented by an icon. Table 5–1 describes what each icon represents, where it appears in the Data Controls panel hierarchy, and what components it can be used to create.

Table 5–1 Data Controls Panel Icons and Object Hierarchy for Web Services

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. Typically, there is one data control for each web service.	Serves as a container for other objects and is not used to create anything.
	Collection	Represents a data collection returned by an operation on the service. Collections also 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. For more information, see "Creating a Basic Databound Page," "Creating ADF Databound Tables," "Displaying Master-Detail Data," and "Creating Databound Chart and Gauge Components" in *Developing Fusion Web Applications with Oracle Application Development Framework*.
	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, date, list of values, and selection list components. For more information, see "Creating Text Fields Using Data Control Attributes" in *Developing Fusion Web Applications with Oracle Application Development Framework*.

Table 5–1 (Cont.) Data Controls Panel Icons and Object Hierarchy for Web Services

Icon	Name	Description	Used to Create...
	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.	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" in *Developing Fusion Web Applications with Oracle Application Development Framework*.
	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 single value, a structure or a collection of those.	Command components. For methods that accept parameters: command components and parameterized forms. For more information, see "Using Command Components to Invoke Functionality in the View Layer"

Table 5–1 (Cont.) Data Controls Panel Icons and Object Hierarchy for Web Services

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" in *Developing Fusion Web Applications with Oracle Application Development Framework*	The same components as for collections and attributes and for query forms. For more information on query forms, see "Creating ADF Databound Search Forms" in *Developing Fusion Web Applications with Oracle Application Development Framework*.
	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. If an operation requires one or more parameters, they are listed in a **Parameters** node under the operation. The following operations for navigation and setting the current row are supported: `First`, `Last`, `Next`, `Previous`, `setCurrentRowWithKey`, and `SetCurrentRowWithKeyValue`. `Execute` is supported for refreshing queries. `Create` and `Delete` are available as applicable, depending on the web service operation. Because the web service data controls are not transactional, `Commit` and `Rollback` are not supported.	UI command components, such as buttons, links, and menus. For more information, see "Creating Command Components Using Data Control Operations" and "Creating an Input Form" in *Developing Fusion Web Applications with Oracle Application Development Framework*.
	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. Array and structured parameters are exposed as updatable structured attributes and collections under the data control, which can be dropped as an ADF form or an updatable table on the UI. You can use the UI to build a parameter that is an array or a complex object (not a standard Java type).	Label, text, and selection list components.

5.2.9 What You May Need to Know About Making an XML Schema Available to a REST Data Control

Typically, when you include an XML schema as part of a REST data control, the data control definition stores the reference as an absolute path, which works well if the XSD that you specify is a network resource. If the XSD that you specify is a local resource, you can place the XSD in your project's resource path so that the XSD is packaged with your application and the reference to it is stored as a relative path.

If you provide a reference to a local copy of an XSD file that is not in the project's resource path, the reference to the resource will be stored as an absolute path. If you then run the application on a local instance of the application server, the XSD will be available to your application. However, if you deploy the application to a remote server, the application will not be able to access the XSD.

In order to make sure a local copy of an XSD is available to your application when deployed to a remote server instance, you need to manually copy the XSD to the project's resource path before creating the data control.

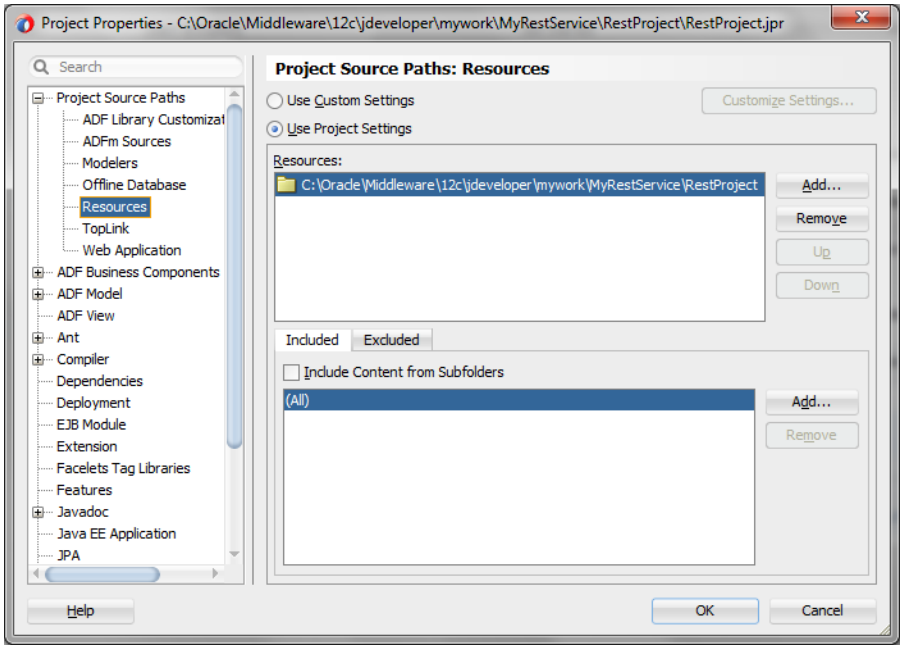
You can identify the project's resource path in the Project Properties dialog.

To identify the resource path for a project:

1. Right-click the project's node and choose **Project Properties**.
2. In the Project Properties dialog, expand the **Project Source Paths** node and select the **Resources** node, as shown in Figure 5–3.

   The folders listed in the Resources field are the project's resource path.

Figure 5–3 Project Resource Path



5.3 Securing Web Service Data Controls

Web services allow applications to exchange data and information through defined application programming interfaces. SSL (Secure Sockets Layer) provides secure data transfer over unreliable networks, but SSL only works point to point. Once the data reaches the other end, the SSL security is removed and the data becomes accessible in its raw format. A complex web service transaction can have data in multiple messages being sent to different systems, and SSL cannot provide the end-to-end security that would keep the data invulnerable to eavesdropping.

Any form of security for web services has to address the following issues:

- The authenticity and integrity of data
- Data privacy and confidentiality
- Authentication and authorization
- Non-repudiation
- Denial of service attacks

Throughout this section the "client" is the web service data control, which sends SOAP or REST messages to a deployed web service. The deployed web service may be:

- A web service running on Oracle WebLogic Server
- A web service running anywhere in the world that is accessible through the Internet

5.3.1 Oracle WSM Policy Framework

You can use Oracle Web Services Manager (WSM) policy framework to manage and secure web services consistently across your organization. The policy framework is built using the WS-Policy standard, which unifies multiple technologies to make secure web services interoperable between systems and platforms.

Among others, the Oracle WSM Policy Framework addresses the following aspects of web services security issues:

- Authentication and authorization

  The identity of the sender of the data is verified, and the security system ensures that the sender has privileges to perform the data transaction.

  The type of authentication can be a basic username/password pair transmitted in plain text, or trusted X509 certificate chains. SAML assertion tokens can also be used to allow the client to authenticate against the service, or allow it to participate in a federated SSO environment, where authenticated details are shared between domains in a vendor-independent manner.

- Data authenticity, integrity, and non-repudiation

  XML digital signatures, which use industry-standard messages, digest algorithms to digitally sign the SOAP message.

- Data privacy

  XML encryption that uses industry-standard encryption algorithms to encrypt the message.

- Denial of service attacks

  Defines XML structures to time-stamp the SOAP message. The server uses the time stamp to invalidate the SOAP message after a defined interval.

For more information on web service security, see *Securing Web Services and Managing Policies with Oracle Web Services Manager*

5.3.2 Using Key Stores

A web service can be configured for message-level security using key stores. For more information about creating and using key stores for message protection, see "Configuring Message Protection" in *Securing Web Services and Managing Policies with Oracle Web Services Manager*.

5.3.3 How to Define SOAP Web Service Data Control Security

After you create a SOAP-based web service data control in a JDeveloper project, you can define security for the data control using the Edit Data Control Policies dialog.

Before you begin:

It may be helpful to have an understanding of how security is used in web service data controls. For more information, see Section 5.3, "Securing Web Service Data Controls."

You may also find it helpful to understand additional functionality that can be added using other web services features. For more information, see Section 5.1.2, "Additional Functionality for Web Service Data Controls in ADF Applications."

To define security for a SOAP web service data control:

1. In the Applications window, select the web service data control `DataControls.dcx` file.
2. In the Structure window, right-click the web service data control and choose **Define Web Service Security**.

   JDeveloper displays the Edit Data Control Policies dialog, which shows the Policy Store location.

   > **Note:** If you want to use an alternative policy store, you must first specify it in the **WS Policy Store** page of the Preferences dialog. To do so, from the main menu, choose **Tools > Preferences** and select the **WS Policy Store** page.

3. From the **Ports** dropdown list, select the port to which you want then specified policies applied.
4. From the **MTOM** dropdown list, select the MTOM (message transmission optimization mechanism) policy you want to use. If you leave this field blank, no MTOM policy is used.
5. From the **Reliability** dropdown list, select the reliability policy you want to use. If you leave this field blank, no reliability policy is used.
6. From the **Addressing** dropdown list, select the addressing policy you want to use. If you leave this field blank, no addressing policy is used.
7. In the **Security** list, you can optionally specify additional security policies to apply. To add a policy, select its checkbox.
8. In the **WS Config** list, you can optionally specify additional web service configuration policies to apply. To add a policy, select its checkbox.

   For more information on the configuration policies, see "Configuration Policies" in *Securing Web Services and Managing Policies with Oracle Web Services Manager*.
9. In the **Management** list, you can optionally specify additional management policies to apply. To add a policy, select its checkbox.
10. If necessary, you can also remove policies from the **Security** list and the **Management** lists by deselecting their corresponding checkboxes.
11. You can optionally override properties for the policies in the **Security** list and the **Management** list by clicking **Override Properties**.
12. Click **OK**.

For more information about predefined policies and configuring policies and their properties, see "Configuration Policies" in *Securing Web Services and Managing Policies with Oracle Web Services Manager*.

6

Exposing URL Services Using the ADF Model Layer

This chapter describes how to create data controls for URL services.

This chapter includes the following sections:

- Section 6.1, "About Using ADF Model with URL Services"
- Section 6.2, "Exposing URL Services with ADF Data Controls"
- Section 6.3, "Using URL Service Data Controls"

6.1 About Using ADF Model with URL Services

A URL service can be simply a URL against which a query is posted, so that the URL can be exposed as an ADF form. For example, say you have a URL service that allows you to access employee data for your company. This data can be retrieved using a URL as shown in Example 6–1.

Example 6–1 Sample URL that Accesses a URL Service

```
http://example.com/getEmployee?empId=20+deptId=10
```

This simple URL that accesses employee data can become an ADF **data control** with a method (`getEmployee`) and two parameters (`empId` and `deptId`), that can then be dropped on a page as a form.

URL services can also utilize representational state transfer (REST) actions. REST services are web services that can be accessed using a simple HTTP URL, rather than a more complex SOAP protocol. The HTTP actions (`GET`, `PUT`, `POST`, `DELETE`) are mapped to service operations that access and manipulate data at the service implementation. The response data can be returned in delimiter separated value and XML formats, and you can specify an XSD to define the input format for the `PUT` and `POST` actions.

Note: This chapter focuses on using the URL Service data control type to create a data control based on a single URL service. To create a data control based on REST web services, you might be better served by using the REST Web Services data control type, which enables you to incorporate multiple REST methods in one data control with a single pass through the wizard. For more information, see Section 5.2.2, "How to Create a Data Control for a RESTful Web Service."

6.1.1 URL Services Use Cases and Examples

The REST architecture simplifies web service invocation by representing a web service as an HTTP resource, so that web service methods and operations look like a resources on the server that can be accessed through an HTTP URL.

For example, say a web service has a method called `getEmployee(int EmpID)`. Using the REST architecture, this can become `http://mywebservice.com/myService/getEmployee?EmpID=20`. When represented as a plain HTTP URL, it is easy to use the URL service data control to quickly create a databound page that accesses this service.

6.1.2 Additional Functionality for URL Services

You may find it helpful to understand other data access features before you start working with URL services. Following are links to other functionality that may be of interest.

- 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 more information, see Section 2.3.1, "How to Use the Data Controls Panel."
- For more general information about creating data controls, see Chapter 2, "Using ADF Data Controls."
- For information about creating web service data controls, see Chapter 5, "Exposing Web Services Using the ADF Model Layer."

6.2 Exposing URL Services with ADF Data Controls

The URL service data control enables you to access and consume data streams from specified URLs. A URL service data control can represent multiple operations. For example, the `GET` operation and the `PUT` operation for a given URL service can be represented by the same data control.

6.2.1 How to Create a URL Connection

You use the Create URL Connection dialog to create a URL connection by supplying a name and the connection details required to access a URL endpoint. You can access this dialog from the Create URL Service Data Control wizard or separately. If you access the dialog from the Create URL Service Data Control wizard, the connection only applies to the data control's application. If you access the dialog separately, you can create a single IDE-level connection that can be copied by multiple data controls.

Before you begin:

It may be helpful to have a general understanding of URL service data controls. For more information, see Section 6.2, "Exposing URL Services with ADF Data Controls."

You may also find it helpful to understand additional functionality that can be added using other URL services features. For more information, see Section 6.1.2, "Additional Functionality for URL Services."

You need to complete this task:

Make sure you have access to the URL service that the data control will access.

To create an IDE-level URL connection:

1. From the main menu, choose **File > New > From Gallery**.

2. In the New Gallery, expand **General**, select **Connections** and then **URL Connection**, and click **OK**.
3. In the Create URL Connection dialog, select where to create the connection.

   Select **Application Resources** if you want the URL connection to be available only within the application. Select **IDE Connections** if you want the URL connection to be available from the Resources window for use in other applications.
4. Enter a name for the connection.
5. In the **URL Endpoint** field, enter the URL of the desired data stream.

   Typically, this includes just the host and port. Do not include any URL parameters. For example, you can enter something like `http://service.example.com:7101/`.
6. Select the level of authentication from the **Authentication Type** dropdown.

   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. If the URL is associated with a protected area of the overall web site, enter the authentication realm.
9. After you have entered the name and endpoint, you can click Test Connection to verify URL connection is valid.
10. Click **OK**.

6.2.2 How to Create a URL Service Data Control

You can create a URL Service data control using the Create URL Service Data Control wizard, which is available from the New Gallery. When you create URL service data controls, you use the wizard to create each operation, one at a time.

Before you begin:

It may be helpful to have a general understanding of URL service data controls. For more information, see Section 6.2, "Exposing URL Services with ADF Data Controls."

You may also find it helpful to understand additional functionality that can be added using other URL services features. For more information, see Section 6.1.2, "Additional Functionality for URL Services."

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*.
- Make sure you have access to the URL service that the data control will access.
- Optionally, create the URL connection you will use, as described in Section 6.2.1, "How to Create a URL Connection."

To create a URL service data control:

1. In the Applications window, right-click the project where you want to place the data control and choose **New > From Gallery**.
2. In the New Gallery, expand **Business Tier**, select **Data Controls** and then **URL Service Data Control**, and click **OK**.
3. In the Create URL Service Data Control wizard, on the Data Source page, provide a name for the data control.
4. Select the URL connection for the data control to use.
   - If you have not yet created a URL connection, click the **Create New Connection** icon and specify a URL and a name for the connection.
   - If you have already created a URL connection, select the connection you want to use from the **Connection** dropdown.
5. From the **HTTP Method** dropdown list, select the action for this data control.
6. Optionally, select the **Include HTTP Header Parameter** checkbox. For more information, see Section 6.2.5, "How to Include a Custom Header Parameter for a URL Service Data Control."
7. In the **Source** field, enter the source for the URL service (for example, *`servicepath/servicename`*), and click **Next**.

   If this operation requires a parameter, you can use the format `?symbol=##`*`ParamName`*`##` to specify it (for example, `servicepath/servicename?symbol=##id##`). You can also use a parameter to provide dynamic input for the source path (for example, `servicepath/##servicename##?symbol=##id##`).
8. On the Parameters page, supply default values for any parameters you specified, and click **Next**.

   For the `PUT` and `POST` operations, you must also provide an XML schema definition (XSD) that defines the format of the input.
9. On the Data Format page, select the data format of the data source and set the associated properties, then click **Next**.

   You can choose either XML format (for which you provide URLs for the XSD and XSL files) or delimiter separated value (for which you specify the delimiter, text qualifier, and encoding for the data).

   > **Note:** The XSL URL field is optional. You can use this field to specify an XSL style sheet to define custom elements, such as fields calculated on the basis of the values of other fields. These elements will then be revealed as attributes in the data control. For more information on preparing the XSL style sheet, see Section 5.2.7, "How to Add Custom Attributes to a REST Web Service Data Control."
10. On the Finish page, you can click **Test URL Connection** to verify that the URL data connection is valid, and click **Finish**.
11. Repeat the above steps for any other operation you would like to include in the data control. In order to include the additional operations in the same data control, fill in the **Name** field of the Data Source page of the wizard with the same name for the data control that you used when you ran the wizard for the first operation.

6.2.3 What Happens When You Create a URL Service Data Control

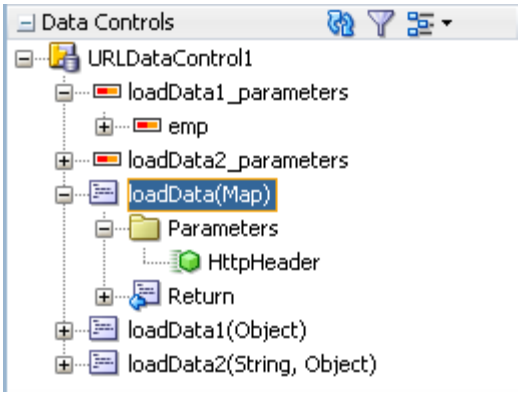
When you create a data control, JDeveloper creates the data control definition file (`DataControls.dcx`), opens the file in the overview editor, and displays the file's hierarchy in the Data Controls panel. For more information, see Section 2.2.2, "What Happens in Your Project When You Create a Data Control"

When you create a URL Service data control, the `DataControls.dcx` overview editor is populated with method nodes for each of your operations. Those method nodes may have **Return** subnodes, which in turn can contain subnodes for collections, scalar values, and attributes.

For operations on which you have selected the **Include HTTP Header Parameter** checkbox, an input parameter called `HttpHeader` and of type `Map` is specified for the method node.

For example, Figure 6–1 shows a data control with three methods, including a method with a custom header parameter (`loadData(Map)`).

Figure 6–1 URL Data Control in Data Controls Panel



For operations that take a parameter of a complex data type, a structured attribute node also appears.

See Table 6–1 for the full list of the nodes that appear for URL service data controls and information on how you can use them.

6.2.4 What You May Need to Know About Generating URL Data Controls without Schema

When you create a URL data control based on an HTTP GET method, a schema will be generated automatically if you do not specify one in the wizard. However, this auto-generated schema might not contain the information needed for the data control to work as you might expect. For example, the auto-generated schema has the following limitations:

- If there is no data returned from an element, the generated schema does not reflect the detail of the element's structure, and thus there are no corresponding data control objects shown in the Data Controls panel.
- If the returned data from a given element only contains one row, the element will not be treated as a table in the resulting data control. (You can fix this by adding the `maxOccurs="unbounded"` attribute to the given element in the schema.)
- If you set an attribute as a primary key, the generated schema might not reflect the correct data type. For more information, see Section 6.2.6, "What You May Need to

Know About Primary Keys in URL Service Data Controls."

The generated schema can be found under the project's Resources node in the Applications window, where you can inspect and edit it.

6.2.5 How to Include a Custom Header Parameter for a URL Service Data Control

When using a URL service data control, you may want to add a custom parameter to the HTTP header when invoking the HTTP request. Such a parameter can be useful for a variety of purposes, including for security or for notifications.

Before you begin:

It may be helpful to have a general understanding of URL service data controls. For more information, see Section 6.2, "Exposing URL Services with ADF Data Controls."

You may also find it helpful to understand additional functionality that can be added using other URL services features. For more information, see Section 6.1.2, "Additional Functionality for URL Services."

You need to complete these tasks:

- Create a URL Service data control and select the **Include HTTP Header Parameter** checkbox in the wizard. For more information, see Section 6.2.2, "How to Create a URL Service Data Control."
- Create a JSF page as described in "Creating a Web Page" in *Developing Web User Interfaces with Oracle ADF Faces*.
- Create a managed bean and register it in the view project's `adfc-config.xml` file. For more information, see "How to Use a Managed Bean to Store Information" in *Developing Fusion Web Applications with Oracle Application Development Framework*.

To pass a custom header to a URL service:

1. In the managed bean, create a bean property of type `java.util.Map` that provides the header parameter names and values.

   See Example 6–2 for an example of such a managed bean.

2. From the Data Controls panel, drag the method that includes the custom header parameter to a web page and drop it as an **ADF Button**.
3. In the Edit Action Binding dialog, perform the following steps to create the binding between the button and the managed bean:
   a. In the Data Collection tree, select the method that includes the HTTP header parameter.
   b. In the Parameters table, click in the Value cell for the parameter, click the drop-down button, and then choose **Show EL Expression Builder**.
   c. In the Variables dialog, type the expression by hand or generate the expression by navigating through the Variables tree and selecting the managed bean field that represents the HTTP parameter to which you are binding.

      For example, to bind to the `httpHeadersMap` property shown in Example 6–2, you would expand the **ADF Managed Beans** node, expand the node for the bean's scope, expand the node for name of the bean specified in the `adfc-config.xml` file (which is not necessarily the same as the class name), and select `httpHeadersMap`.

At runtime, the data control will obtain the entries from the map (if it is present) and add them as HTTP headers in the request.

Example 6–2 Managed Bean Containing Custom Parameters for URL Service Data Control

```
package view;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class BackingBean {

    private Map<String,String> httpHeadersMap = new HashMap<String,String>();

      public BackingBean() {
          httpHeadersMap.put("TenantID","OurCompany");
      }

      public void setHttpHeadersMap(Map<String,String> httpHeadersMap) {
          this.httpHeadersMap = httpHeadersMap;
      }

      public Map<String,String> getHttpHeadersMap() {
          return httpHeadersMap;
      }
}
```

6.2.6 What You May Need to Know About Primary Keys in URL Service Data Controls

When you create a data control on a URL service, the data control supports primary key operations on any exposed collection. However, to be sure the primary key functionality works correctly, on the Data Format page of the URL Service Data Control wizard, you must specify a schema that names the data type of the primary key field. (For URL data controls based on HTTP GET methods, you are not necessarily required to provide a schema.)

Otherwise, this feature is the same as the corresponding feature for web service data controls. For more information, see Section 5.2.6, "What You May Need to Know About Primary Keys in SOAP Web Service Data Controls."

6.2.7 What You May Need to Know About URL Service Data Controls

Because the URL Service data control is not updatable, there are limitations on some of the objects in the Data Controls panel. For example, the only built-in operations available under the Operations node are for retrieval and navigation. Also, in a URL Service data control, the parameter object is the parameter that a user passes in the URL For more information, see Table 6–1.

6.3 Using URL Service Data Controls

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 more information, see Section 2.3.1, "How to Use the Data Controls Panel."

In the Data Controls panel, each data control object is represented by an icon. Table 6–1 describes what each icon represents, where it appears in the Data Controls panel hierarchy, and what components it can be used to create.

Table 6–1 Data Controls Panel Icons and Object Hierarchy for the URL Service 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.
	Method	Represents the `loadData()` operation, which retrieves the contents of the URL. This operation may accept parameters, perform some action or business logic, and return data or data collections. If the method is a `get` method of a `Map` and returns a data collection, a method return icon appears as a child under it. If a method requires a parameter, a node appears under the method, which lists the required parameters.	Command components. For methods that accept parameters: command components and parameterized forms. For more information, see "Using Command Components to Invoke Functionality in the View Layer" in *Developing Fusion Web Applications with Oracle Application Development Framework*.
	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" in *Developing Fusion Web Applications with Oracle Application Development Framework*	The same components as for collections and attributes and for query forms. For more information on query forms, see "Creating ADF Databound Search Forms" in *Developing Fusion Web Applications with Oracle Application Development Framework*.
	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. For more information, see "Creating a Basic Databound Page," "Creating ADF Databound Tables," "Displaying Master-Detail Data," and "Creating Databound Chart and Gauge Components" in *Developing Fusion Web Applications with Oracle Application Development Framework*.

Table 6–1 (Cont.) Data Controls Panel Icons and Object Hierarchy for the URL Service Data Control

Icon	Name	Description	Used to Create...
	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.	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" in *Developing Fusion Web Applications with Oracle Application Development Framework*.
	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" in *Developing Fusion Web Applications with Oracle Application Development Framework*.
	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. If an operation requires one or more parameters, they are listed in a **Parameters** node under the operation. The following navigation operations are supported: `First`, `Last`, `Next`, and `Previous`. `Execute` is supported for refreshing queries. `Create` and `Delete` are available as applicable, depending on the URL method. Because the URL service data controls are not transactional, `Commit` and `Rollback` are not supported.	UI command components, such as buttons, links, and menus. For more information, see "Creating Command Components Using Data Control Operations" and "Creating an Input Form" in *Developing Fusion Web Applications with Oracle Application Development Framework*.
	Parameter	Represents a parameter value that is declared by the method or operation under which it appears. Parameters appear in a node under a method or operation. The parameter for a URL Service data control is the parameter that a user passes in the URL. These show up as a parameters to the `loadData()` method when the URL Service data control is created. For example, say you create a data control to the URL `http://www.example.org?id=##param##`. On the Data Controls panel, you would see that the `loadData()` method has one parameter with the name `param`. The value supplied to this parameter is substituted in the URL when the invocation occurs.	Label, text, and selection list components.

7

Adding Business Logic to Data Controls

This chapter describes how to configure your data controls with declarative metadata, such as default labels for attributes, new transient attributes, validation rules, and built-in search criteria. In addition, this chapter shows how you can test the data controls that you have configured before you use them in an application and provides a reference on using Groovy expressions in data controls.

This chapter includes the following sections:

7.1 Introduction to Adding Business Logic to Data Controls

When you generate **data controls**, you can use them without further modification to create bindings between your data model and the UI components in your application. In addition, you can configure the data controls to add business logic and other features to your data model so that those features are applied when you use the Data Controls panel to create UI components. For example, depending on the type of data control you are creating, you can do the following things:

- Configure default values for attributes
- Add labels and tooltips for attributes
- Add custom metadata (typically name-value pairs) that can be referenced via expression language (EL) from the ADF Faces UI
- Add calculated attributes
- Add attribute-level validation rules with custom error messages
- Define declarative search forms

7.2 Configuring Data Controls

When you create a data control, a standard set of values and behaviors are assumed for the data control. For example, the data control determines how the label for an attribute will display in a client. You can configure these values and behaviors by creating and modifying data control structure files that correspond to the elements of the data control. You first generate a data control structure file using the overview editor for the `.dcx` file.

7.2.1 How to Edit a 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 Section 7.2, "Configuring Data Controls."

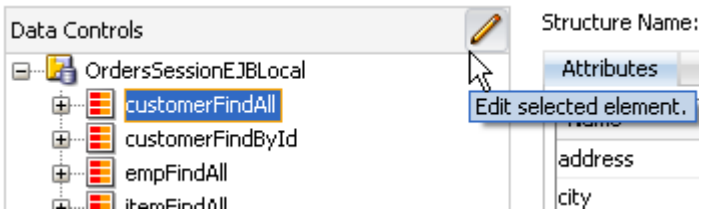
You will need to complete this task:

Create a data control, as described in Section 2.2.1, "How to Create ADF Data Controls."

To edit a 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 7–1.

Figure 7–1 Edit Button in Data Controls Registry



3. In the overview editor of the data control structure file, make the desired modifications.

7.2.2 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. This file stores configuration data for the data control that is specific to that collection, such as any UI hints or validators that you have specified for the data object.

A data control structure file has the same base name as the data object with which it corresponds. For example, if you click the **Edit** icon when you have a collection node selected that corresponds with the `Customer.java` entity bean, the data control structure file is named `Customer.xml`. The data control structure file is generated in a package that corresponds to the package of the bean class, but with `persdef` prepended to the package name. For example, if the `Customer.java` bean is in the

`model` package, the `Customer.xml` data control definition file is generated in the `persdef.model` package. Once a data control structure file has been generated, you can use the overview editor for that file to make further configurations.

A data control structure file contains the following information:

- Attributes: Describes all of the attributes on the service. For example, for entity beans, there is an attribute for each bean property that is mapped to a database column. You can also add transient attributes. You can set UI hints that define how these attributes will display in the UI. You can also set other properties, such as whether the attribute value is required, whether it must be unique, and whether it is visible. For more information, see Section 7.3, "Working with Attributes."

  You can also set validation for an attribute and create custom properties. For more information on validation, see Section 7.5, "Defining Validation Rules on Attributes Declaratively." For more information on custom properties, see the "How to Implement Generic Functionality Driven by Custom Properties" section of *Developing Fusion Web Applications with Oracle Application Development Framework*.

- Accessors: Describes data control elements that return result sets.

- Named Criteria: Enables you to create rules to filter the rows that are presented. Named criteria are available for JPA-based data controls.

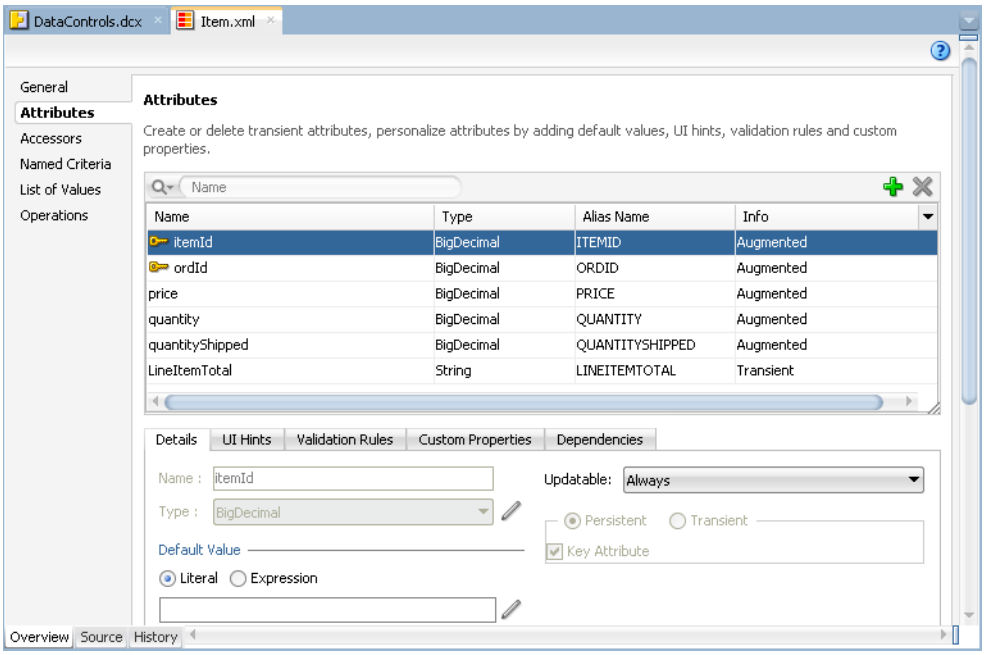
  For more information, see Section 7.6, "Filtering Result Sets with Named Criteria."

- List of Values: Enables you to create declarative associations between an attribute of the current data object and rows from another data object. You can then create components from these associations such as dropdown lists.

- Operations: Describes methods on the data object that are used by the data control's built-in operations, such as `add` and `remove` methods, which are used by the Create and Delete built-in operations, respectively.

Figure 7–2 shows the data control structure file for the `Item` bean.

Figure 7–2 Data Control Structure File in the Overview Editor



Note: The overview editor of a data control structure file shows all of the attributes, accessors, and operations that are associated with the data object. However, the data control structure file's XML source only contains definitions for elements that you have edited. The base elements are introspected from the data object. Also, when you make changes to the underlying data object, the data control inherits those changes.

7.2.3 How to Convert Data Controls from a Previous Release

The format of data control structure files changed in Release 11.1.2 to enable additional features, such as named criteria and list of values. In addition, data control structure files are now only generated on demand, such as when you need to add metadata to a data control object or if you have them generated when creating the data control. There is no longer metadata for data control objects that merely repeats information that is already available in the object on which the data control is based. For JPA-based data controls, `UpdatableSingleValue.xml` and `UpdateableCollection.xml` files are no longer generated.

If you have a data control from a previous release, you can continue using that data control in applications that you develop with later versions of JDeveloper. However, if you plan to make changes to that data control, you should first convert the data control to the new format. Modification of data controls in the old format is not supported.

Converting an old data control to use the current structure definition format involves a few changes to the `DataControls.dcx` file and generation of new data control structure files. You do not need to rebind existing UI components to the updated data control.

To migrate a data control from a previous release:

1. In the current version of JDeveloper, open the application workspace that contains the data control.
2. Open the `DataControls.dcx` file and make the following changes:
   - Change the value of the `ImplDef` attribute to the name of the appropriate data control implementation class, as shown in Table 7–1.
   - Optionally, for bean and EJB data controls, add a `DataControlHandler` attribute to the data control's `bean-definition` or `ejb-definition` element to enable full query and paging support. For JPA-based data controls, typically you would typically use `oracle.adf.model.adapter.bean.jpa.JPQLDataFilterHandler`. For non-JPA data controls you typically use `oracle.adf.model.adapter.bean.DataFilterHandler`.
   - Optionally, for JPA-based bean and EJB data controls, add an `accessMode` attribute to the data control's `bean-definition` or `ejb-definition` element to specify the type of paging for the data control. The value of the `accessMode` can be `scrollable` or `rangePaging`. For more information on access mode, see Section 3.4, "Paginated Fetching of Data in EJB Data Controls."
   - Optionally, for JPA-based bean and EJB data controls, add an `eagerPersist` attribute to the data control's `bean-definition` or `ejb-definition` element to specify whether new records are persisted eagerly. The value of `eagerPersist` can be `false` or `true`. For more information, see Section 3.3.8, "About Automatically Persisting New Rows."

3. For data controls to which you have added metadata (such as UI hints or validators), create new data control structure files as shown in Section 7.2.1, "How to Edit a Data Control."
4. Optionally, in the data control's project, delete the old data control structure files. These include:
   - `UpdatableSingleValue.xml`
   - `UpdateableCollection.xml`
   - Files ending in `.xml` that contain metadata for a data control object (not including the new files that you generated in step 3.

   For an example of a converted `DataControls.dcx` file, see Example 7–1.

Table 7–1 Data Control Implementation Classes

Data Control Type	Data Control Implementation Class
EJB	`oracle.adfinternal.model.adapter.ejb.EjbDCDefinition`
Bean	`oracle.adf.model.adapter.bean.BeanDCDefinition`
Web Service	`oracle.adfinternal.model.adapter.webservice.WSDefinition`
URL Service	`oracle.adfinternal.model.adapter.url.URLDCDefinition`

Example 7–1 DataControls.dcx File Converted to Use Sparse Metadata

```
<?xml version="1.0" encoding="UTF-8" ?>
<DataControlConfigs xmlns="http://xmlns.oracle.com/adfm/configuration"
                    version="11.1.1.61.92" id="DataControls" Package="model1">
  <AdapterDataControl id="BeanPagingDC1"
                      FactoryClass="oracle.adf.model.adapter.DataControlFactoryImpl"
                      ImplDef="oracle.adf.model.adapter.bean.BeanDCDefinition"
                      SupportsTransactions="false" SupportsSortCollection="true"
                      SupportsResetState="false" SupportsRangesize="false"
                      SupportsFindMode="false" SupportsUpdates="true"
                      Definition="model1.BeanPagingDC1"
                      BeanClass="model1.BeanPagingDC1"
                      xmlns="http://xmlns.oracle.com/adfm/datacontrol">
    <CreatableTypes>
      <TypeInfo FullName="model1.SampleData1"/>
      <TypeInfo FullName="model1.BeanPagingDC1"/>
    </CreatableTypes>
    <Source>
      <bean-definition BeanClass="model1.BeanPagingDC1"
        DataControlHandler="oracle.adf.model.adapter.bean.DataFilterHandler"
        AccessMode="scrollable"
        xmlns="http://xmlns.oracle.com/adfm/adapter/bean"/>
    </Source>
  </AdapterDataControl>
</DataControlConfigs>
```

7.2.4 What You May Need to Know About MDS Customization of Data Controls

If you wish for all of the objects that are encompassed by the data control to be available for Oracle Metadata Services (MDS) customization, the packaged application must contain data control structure files for those objects.

When you create a data control based on the adapter framework, data control structure files are not generated by default, since they are not needed by the data control if you do not add metadata to a given object. Typically, a data control structure

file is only generated for a data control object once you edit the data control to add declarative metadata (such as UI hints or validators) to that object, as described in Section 7.2.1, "How to Edit a Data Control." To create data control structure files for each data control object, you need to repeat that procedure for each data control object.

> **Note:** In the Create EJB Data Control and the Create Bean Data Control wizards, there is a **Generate Metadata** checkbox that you can select in order to have structure files for each data control object generated upon creation of the data control.

For more information on MDS, see "Customizing Applications with MDS" in *Developing Fusion Web Applications with Oracle Application Development Framework*.

7.3 Working with Attributes

When you create a data control for your business services, you can create a data control structure file for an individual data object in which you can declaratively augment the functionality of the data object's persistent attributes. For example, you can create validation rules and set UI hints to control the default presentation of attributes in UI components. In addition, you can create transient attributes.

In all cases, you set these properties on the Attributes page of the overview editor of the data control structure file. For information on creating a data control structure file, see Section 7.2.1, "How to Edit a Data Control."

7.3.1 How to Designate an Attribute as Primary Key

In the overview editor for a data object's data control structure file, you can designate an attribute as a primary key for that data object if you have not already done so in the data object's underlying class.

Before you begin:

It may be helpful to have an understanding of how you set attribute properties. For more information, see Section 7.3, "Working with Attributes."

You will need to complete this task:

Create the desired data control structure files as described in Section 7.2.1, "How to Edit a Data Control."

To set an attribute as a primary key:

1. In the Applications window, double-click the desired data control structure file.
2. In the overview editor, click the **Attributes** navigation tab.
3. On the Attributes page, select the attribute you want to designate as the primary key and then click the **Details** tab.
4. On the Details page, set the **Key Attribute** property.

> **Note:** If the attribute has already been designated as the primary key in the class (such as through the JPA @ID annotation), the data control inherits that setting and the **Key Attribute** checkbox will be selected. However, in this case, you can not deselect the **Key Attribute** option.

7.3.2 How to Control the Updatability of an Attribute

The `Updatable` property controls whether the value of a given attribute can be updated. You can select the following values for the `Updatable` property:

- **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 row for the first time, but after being successfully committed to the database the attribute is read-only

Before you begin:

It may be helpful to have an understanding of how you set attribute properties. For more information, see Section 7.3, "Working with Attributes."

You will need to complete this task:

Create the desired data control structure files as described in Section 7.2.1, "How to Edit a Data Control."

To set the updatability of an attribute:

1. In the Applications window, double-click the desired data control structure file.
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. On the Details page, select a value from the **Updatable** dropdown list.

7.3.3 How to Define a Static Default Value for an Attribute

The **Value** field in the **Details** section allows you to specify a static default value for the attribute when the value type is set to **Literal**. For example, you might set the default value of a `ServiceRequest` entity bean's `Status` attribute to `Open`, or set the default value of a `User` bean's `UserRole` attribute to `user`.

Before you begin:

It may be helpful to have an understanding of how you set attribute properties. For more information, see Section 7.3, "Working with Attributes."

To define a static default value for an attribute:

1. In the Applications window, double-click the desired data control structure file.
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. On the Details page, select the **Literal** option.
5. In the text field below the **Literal** option, enter the default value for the attribute.

7.3.4 How to Define a Default Value Using a Groovy Expression

You can use a Groovy expression to define a default value for an attribute. This approach is useful if you want to be able to change default values at runtime. However, if the default value is always the same, the value is easier to see and

maintain using value field with the **Literal** type (on the **Details** tab). For general information about using Groovy, see Section 7.9, "Groovy Language Support."

Before you begin:

It may be helpful to have an understanding of how you set attribute properties. For more information, see Section 7.3, "Working with Attributes."

You will need to complete this task:

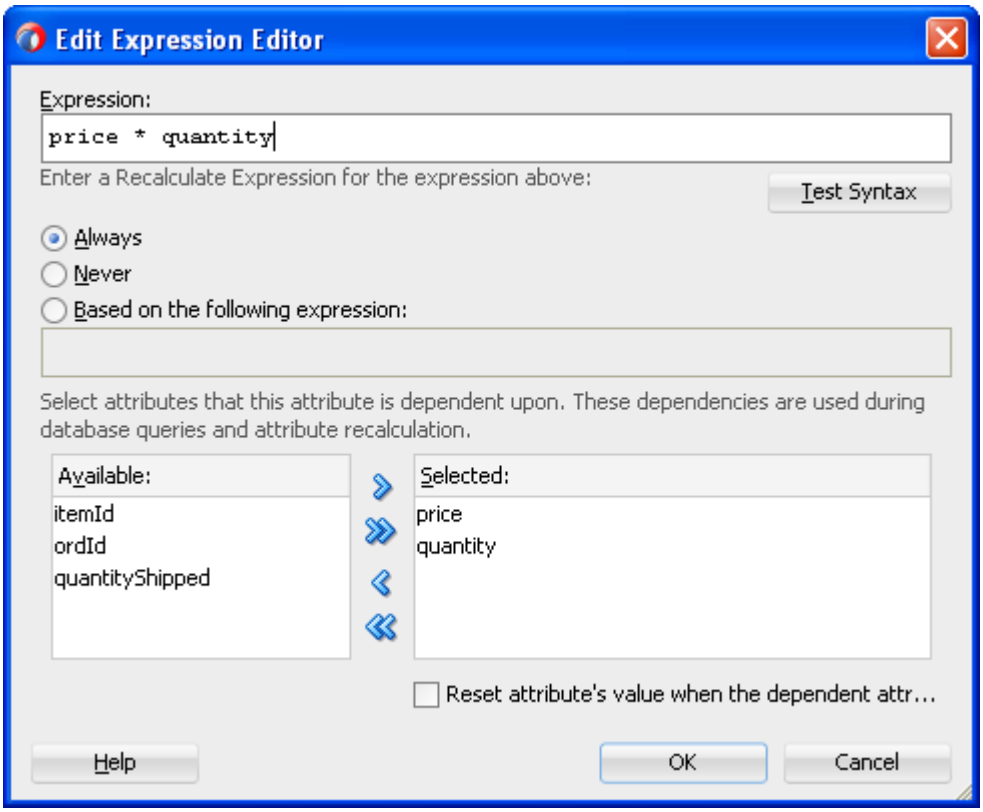
Create the desired data control structure files as described in Section 7.2.1, "How to Edit a Data Control."

To define a default value using a Groovy expression:

1. In the Applications window, double-click the desired data control structure file.
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. On the Details page, select **Expression** for the default value type, and click the **Edit** button next to the adjoining text field.
5. In the Edit Expression dialog, enter an expression in the field provided.

   Attributes that you reference can include any attribute that is defined for the data object.
6. If the attribute is transient, select the appropriate recalculate setting, as shown in Figure 7–3.

Figure 7–3 Edit Expression Editor



If you select **Always** (default), the expression is evaluated each time any attribute in the row changes.

If you select **Never**, the expression is evaluated only when the row is created.

If you select **Based on the following expression**, you can provide an expression that determines when to recalculate the attribute value.

For example, the following expression in the **Based on the following expression** field causes the attribute to be recalculated when either the `Quantity` attribute or the `UnitPrice` attribute are changed:

```
return (adf.object.dataProvider.isAttributeChanged("Quantity") ||
adf.object.dataProvider.isAttributeChanged("UnitPrice"));
```

7. In the **Available** list at the bottom of the dialog, select any attributes upon which the value expression or the optional recalculate expression is based and shuttle each to the **Selected** list.
8. Click **OK** to save the expression.

7.3.5 What Happens When You Create a Default Value Using a Groovy Expression

When you define a default value using a Groovy expression, `<RecalcCondition>` and `<TransientExpression>` tags are added within the tag for the corresponding attribute in the data control structure file. In addition, a `<Dependencies>` tag is added if you have specified any dependencies for the expression. Example 7–2 shows the code that is added to the `LineItemTotal` attribute when an expression is written to define the default value as the product of the `price` and `quantity` attributes and the recalculate condition is specified as **Always**.

Example 7–2 Default Value Expression

```
<RecalcCondition><![CDATA[true]]></RecalcCondition>
<TransientExpression><![CDATA[price * quantity]]></TransientExpression>
<Dependencies>
  <Item
    Value="price"/>
  <Item
    Value="quantity"/>
</Dependencies>
```

7.3.6 How to Set UI Hints on Attributes

You can set UI hints on attributes so that those attributes are displayed and labeled in a consistent and localizable way by any UI components that use those attributes. To create UI hints for attributes, use the overview editor for the data object's data control structure file, which is accessible from the Applications window.

> **Note:** As an alternative to using data control structure files to set UI hints for EJB and bean data controls, you can add annotations directly to the bean classes to create much of the same functionality. For more information, see Section 3.5, "Providing UI Hints for Attributes Using Annotations."

Before you begin:

It may be helpful to have an understanding of how you set attribute properties. For more information, see Section 7.3, "Working with Attributes."

You will need to complete this task:

Create the desired data control structure files as described in Section 7.2.1, "How to Edit a Data Control."

To set a UI hint:

1. In the Applications window, double-click the desired data control structure file.
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 **UI Hints** tab.
4. In the **UI Hints** section, set the desired UI hints.

For information about the various UI hints, see the "Defining UI Hints for View Objects" section of *Developing Fusion Web Applications with Oracle Application Development Framework*.

Note: The view objects referenced in *Developing Fusion Web Applications with Oracle Application Development Framework* are ADF Business Components used to encapsulate SQL queries and to simplify working with the results. Data control structure files function much like view objects in adapter data controls and provide many of the same configuration options. For more information on view objects, see the "About View Objects" section of *Developing Fusion Web Applications with Oracle Application Development Framework*.

7.3.7 What Happens When You Set UI Hints on Attributes

When you set UI hints on an attribute, those hints are stored as properties. Tags for the properties are added to the data object's data control structure file and the values for the properties are stored in a resource bundle file. If the resource bundle file does not already exist, it is generated in the data control's package and named according to the project name when you first set a UI hint.

Example 7–3 shows the code for the `price` attribute in the `Item.xml` data control structure file, including tags for the Label and Format Type hints which have been set for the attribute.

Example 7–3 XML Code for UI Hints

```
<PDefAttribute
  Name="price">
  <Properties>
    <SchemaBasedProperties>
      <LABEL
        ResId="${adfBundle['model.ModelBundle']['model.Item.price_LABEL']}"/>
      <FMT_FORMATTER
        ResId="${adfBundle['model.ModelBundle']['model.Item.price_FMT_
                                                FORMATTER']}"/>
    </SchemaBasedProperties>
  </Properties>
</PDefAttribute>
```

Example 7–4 shows the corresponding entries for the Label and Format Type hints in the `ModelBundle.properties` resource bundle file, which contains the values for all of the project's localizable properties.

Example 7–4 Resource Bundle Code for UI Hints

```
model.Item.price_LABEL=Price
. . .
model.Item.price_FMT_FORMATTER=oracle.jbo.format.DefaultCurrencyFormatter
```

7.4 Adding Transient Attributes to a Data Object

In addition to having attributes that map to columns in an underlying table, your data control structure files can include transient attributes that display calculated values.

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 use a Groovy expression in the attribute definition to specify a default value.

7.4.1 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 transient and calculated attributes. For more information, see Section 7.4, "Adding Transient Attributes to a Data Object."

You will need to complete this task:

Create the desired data control structure files as described in Section 7.2.1, "How to Edit a Data Control."

To add a transient attribute to a data object's data control structure file:

1. In the Applications window, double-click the data object's data control structure file.
2. In the overview editor, click the **Attributes** navigation tab, and then click the **Create new attribute** icon.
3. In the New View Attribute dialog, enter a name for the attribute and click **OK**.
4. On the Attributes page of the overview editor, click the **Details** tab and select an object type from the **Type** dropdown list.
5. Optionally, in the **Default Value** section, set a default value or enter an expression to calculate the default value.

   For information on setting an expression to calculate the default value, see Section 7.3.4, "How to Define a Default Value Using a Groovy Expression."
6. If the value will be calculated with an expression, set **Updatable** to **Never**.

7.4.2 What Happens When You Add a Transient Attribute

When you add a transient attribute, JDeveloper adds a `<ViewAttribute>` tag to the data object's data control structure file to reflect the new attribute. Example 7–5 shows the XML code for a transient attribute named `LineItemTotal` that is based on an expression that multiplies the values of the `price` and `quantity` attributes.

Example 7–5 XML Code for a Transient Attribute

```
<ViewAttribute
```

```
  Name="LineItemTotal"
  IsUpdateable="false"
  IsSelected="false"
  IsPersistent="false"
  PrecisionRule="true"
  Type="java.lang.String"
  ColumnType="$none$">
  SQLType="VARCHAR">
  <RecalcCondition><![CDATA[true]]></RecalcCondition>
  <TransientExpression><![CDATA[price * quantity]]></TransientExpression>
  <Dependencies>
    <Item
      Value="price"/>
    <Item
      Value="quantity"/>
  </Dependencies>
</ViewAttribute>
```

7.5 Defining Validation Rules on Attributes Declaratively

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 data object's data control structure file. Encapsulating the business logic this way ensures that your business information is validated consistently in every client that accesses it, and it simplifies maintenance by centralizing where the validation is stored.

Oracle ADF provides built-in declarative validation rules for many common business needs. You can also base validation on a Groovy expression, as described in Section 7.5.4, "How to Use Groovy Expressions For Validation Rules."

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 set the severity level.

Many of the declarative validation features available for data objects are also available at the page definition level, should your application warrant the use of page-level validation in addition to business-layer validation. For more information, see the "Using Validation in the ADF Model Layer" chapter of *Developing Fusion Web Applications with Oracle Application Development Framework*.

Note: You can also add validation rules by implementing a custom validation class. This approach is particularly useful if you need to define complex parameterized validation rules (such as for checking credit card numbers) that you will need to use multiple times in your application. For more information, see the "Implementing Custom Validation Rules" section of *Developing Fusion Web Applications with Oracle Application Development Framework*.

7.5.1 How to Add Validation Rules to Attributes

The process for adding a validation rule to a data object is similar for most of the validation rules, and is done using the Add Validation Rule dialog. You can open this dialog by opening the data object's data control structure file, selecting an attribute on Attributes page, clicking the **Validation Rules** tab, and then clicking the **Add** icon

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. 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.

To add a declarative validation rule to a data control structure file, use the Attributes page of the data object's overview editor.

Before you begin:

It may be helpful to have an understanding of the use of validation rules in data control structure files. For more information, see Section 7.5, "Defining Validation Rules on Attributes Declaratively."

You will need to complete this task:

Create the desired data control structure files as described in Section 7.2.1, "How to Edit a Data Control."

To add a validation rule:

1. In the Applications window, double-click the desired data control structure file.
2. In the overview editor, click the **Attributes** navigation tab.
3. On the Attributes page, select the attribute for which you want to add the validation rule, and then click the **Validation Rules** tab.
4. In the **Validation Rules** page, click the **Add Validation Rule** icon.
5. In the Add Validation Rule dialog, select the type of validation rule desired from the **Rule Type** dropdown list.

   Note that the subordinate fields change depending on your choices.
6. Use the dialog settings to configure the new rule.

   For more information about the different validation rules, see Section 7.5.3, "How to Use 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 Section 7.5.5, "How to Create Validation Error Messages."
8. Click **OK**.

7.5.2 What Happens When You Add a Validation Rule

When you add a validation rule to a data object, JDeveloper updates the data object's data control structure file to include an entry describing what rule you've used and what rule properties you've entered.

For example, if you add a compare validation rule to the `dateShipped` attribute to ensure that the shipping date does not precede the date in the `dateOrdered` attribute, this results in a `<validation:CompareValidationBean>` entry in the XML file, as shown in Example 7–6.

Example 7–6 Compare Validator

```
<validation:CompareValidationBean
  Name="dateShipped_Rule_0"
```

```
  ResId="${adfBundle['model.ModelBundle']['model.Ord.dateShipped_Rule_0']}"
  OnAttribute="dateShipped"
  OperandType="EXPR"
  Inverse="false"
  CompareType="GREATERTHANEQUALTO">
  <validation:TransientExpression><![CDATA[dateOrdered]]>
  </validation:TransientExpression>
</validation:CompareValidationBean>
```

7.5.3 How to Use the Built-in Declarative Validation Rules

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:

- Make a comparison between an attribute and literal value or expression.
- Validate against a list of values that you provide manually.
- 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.

7.5.3.1 Validating Based on a Comparison

The Compare validator performs a logical comparison between an 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 attribute to which you are applying the rule. In all cases, the type corresponds to the type mapping for the 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.

- Expression

  For information on the expression option, see Section 7.5.4, "How to Use Groovy Expressions For Validation Rules."

Before you begin:

It may be helpful to have an understanding of the use of validation rules in data control structure files. For more information, see Section 7.5, "Defining Validation Rules on Attributes Declaratively."

You will need to complete this task:

Create the desired data control structure files as described in Section 7.2.1, "How to Edit a Data Control."

To validate based on a comparison:

1. In the Applications window, double-click the desired data control structure file.
2. In the overview editor, click the **Attributes** navigation tab.
3. On the Attributes page, select the attribute for which you want to add the validation rule, and then click the **Validation Rules** tab.
4. In the **Validation Rules** page, click the **Add Validation Rule** icon.
5. In the Add Validation Rule dialog, in the **Rule Type** dropdown list, select **Compare**.
6. Select the appropriate operator.
7. Select an item in the **Compare With** list, and based on your selection provide the appropriate comparison value.
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 Section 7.5.5, "How to Create Validation Error Messages."
9. Click **OK**.

7.5.3.2 What Happens When You Validate Based on a Comparison

When you create a Compare validator, a `<validation:CompareValidationBean>` tag is added to the data object's data control structure file.

Example 7–6 shows the XML code for the validator on the `dateShipped` attribute.

7.5.3.3 Validating Using a List of Values

The List validator compares an attribute against a list of values that you provide manually. The validator ensures that the value of the data object attribute is in (or not in, if specified) that list.

Before you begin:

It may be helpful to have an understanding of the use of validation rules in data control structure files. For more information, see Section 7.5, "Defining Validation Rules on Attributes Declaratively."

You will need to complete this task:

Create the desired data control structure files as described in Section 7.2.1, "How to Edit a Data Control."

To validate using a list of values:

1. In the Applications window, double-click the desired data control structure file.
2. In the overview editor, click the **Attributes** navigation tab.
3. On the Attributes page, select the attribute for which you want to add the validation rule, and then click the **Validation Rules** tab.
4. In the **Validation Rules** page, click the **Add Validation Rule** icon.
5. In the Add Validation Rule dialog, in the **Rule Type** dropdown list, select **List**.
6. In the **Operator** field, select **In** or **NotIn**, depending on whether you want an inclusive list or an exclusive list.
7. In the **Enter List of Values** section, enter the values, one per line.

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 Section 7.5.5, "How to Create Validation Error Messages."
9. Click **OK**.

7.5.3.4 What Happens When You Validate Using a List of Values

When you validate using a list of values, a `<validation:ListValidationBean>` tag is added to the data object's data control structure file.

7.5.3.5 Ensuring That a Value Falls Within a Certain Range

The Range validator performs a logical comparison between an 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 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, use the Script Expression validator and provide a Groovy expression. For more information, see Section 7.9.1, "How to Reference ADF Objects in Groovy Expressions."

Before you begin:

It may be helpful to have an understanding of the use of validation rules in data control structure files. For more information, see Section 7.5, "Defining Validation Rules on Attributes Declaratively."

You will need to complete this task:

Create the desired data control structure files as described in Section 7.2.1, "How to Edit a Data Control."

To validate within a certain range:

1. In the Applications window, double-click the desired data control structure file.
2. In the overview editor, click the **Attributes** navigation tab.
3. On the Attributes page, select the attribute for which you want to add the validation rule, and then click the **Validation Rules** tab.
4. In the **Validation Rules** page, click the **Add Validation Rule** icon.
5. In the Add Validation Rule dialog, in the **Rule Type** dropdown list, select **Range**.
6. In the **Operator** field, select **Between** or **NotBetween**.
7. In the **Minimum Value** and **Maximum Value** fields, enter appropriate values.
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 Section 7.5.5, "How to Create Validation Error Messages."
9. Click **OK**.

7.5.3.6 What Happens When You Use a Range Validator

When you validate against a range, a `<validation:RangeValidationBean>` tag is added to the data control structure file.

Example 7–7 shows the `quantity` attribute with a minimum of zero and a maximum of 10.

Example 7–7 Range Validator XML Code

```
<PDefAttribute
  Name="quantity">
  <validation:RangeValidationBean
    Name="quantity_Rule_0"
    ResId="${adfBundle['model.ModelBundle']['QUANTITY_VALIDATOR']}"
    OnAttribute="quantity"
    OperandType="LITERAL"
    Inverse="false"
    MinValue="0"
    MaxValue="10"/>
. . .
</PDefAttribute>
```

7.5.3.7 Validating 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. For example, you might have a field where the user enters a password or PIN and you need to validate that it is at least 6 characters long, but not longer than 10.

Before you begin:

It may be helpful to have an understanding of the use of validation rules in data control structure files. For more information, see Section 7.5, "Defining Validation Rules on Attributes Declaratively."

You will need to complete this task:

Create the desired data control structure files as described in Section 7.2.1, "How to Edit a Data Control."

To validate against a number of bytes or characters:

1. In the Applications window, double-click the desired data control structure file.
2. In the overview editor, click the **Attributes** navigation tab.
3. On the Attributes page, select the attribute for which you want to add the validation rule, and then click the **Validation Rules** tab.
4. In the **Validation Rules** page, click the **Add Validation Rule** icon.
5. In the Add Validation Rule dialog, in the **Rule Type** dropdown list, select **Length**.
6. In the **Operator** field, select how to evaluate the value.
7. In the **Comparison Type** field, select **Byte** or **Character** and enter a length in the **Value** field.
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 Section 7.5.5, "How to Create Validation Error Messages."
9. Click **OK**.

7.5.3.8 What Happens When You Validate Against a Number of Bytes or Characters

When you validate using length, a `<validation:LengthValidationBean>` tag is added to the data control structure file. Example 7–8 shows the Between operator with a minimum value of 6 and a maximum value of 10.

Example 7–8 Validating the Length Between Two Values

```
<validation:LengthValidationBean
  OnAttribute="pin"
  CompareType="BETWEEN"
  DataType="CHARACTER"
  MinValue="6"
  MaxValue="10"
  Inverse="false"/>
```

7.5.3.9 Validating 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 Section 7.5.4, "How to Use Groovy Expressions For Validation Rules."

Before you begin:

It may be helpful to have an understanding of the use of validation rules in data control structure files. For more information, see Section 7.5, "Defining Validation Rules on Attributes Declaratively."

You will need to complete this task:

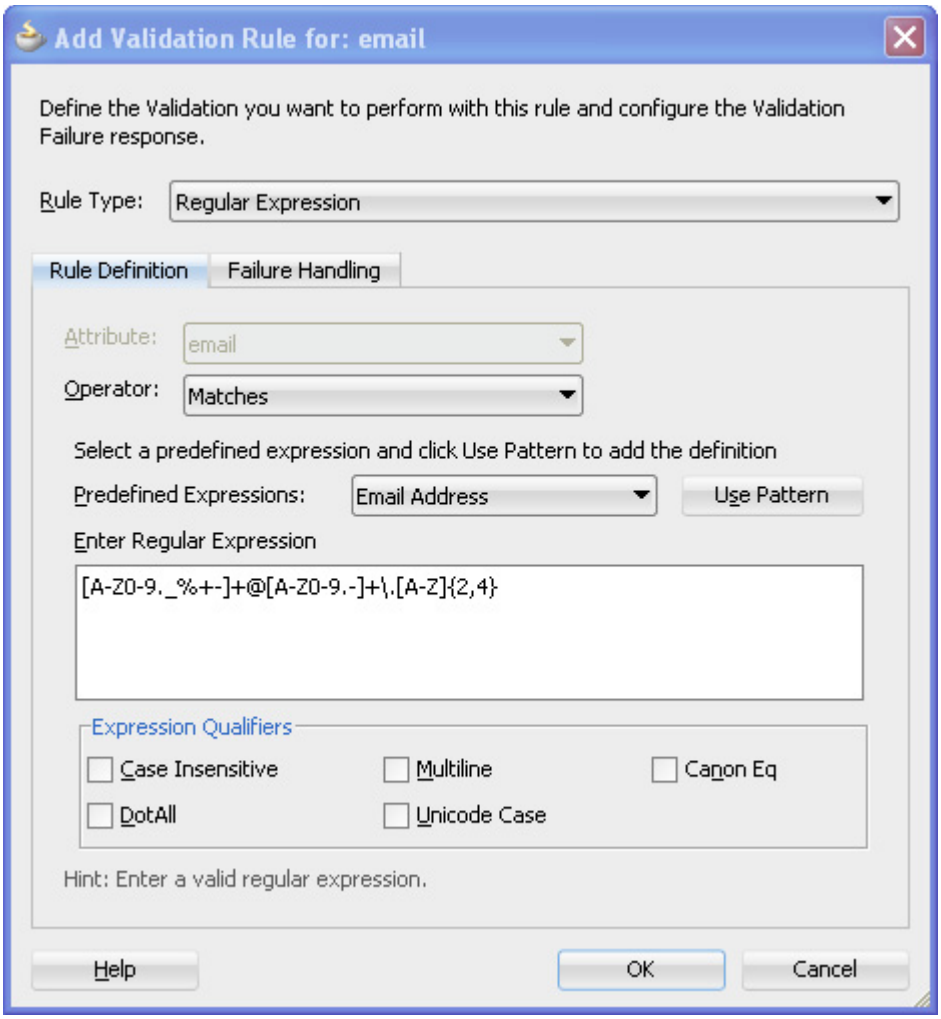
Create the desired data control structure files as described in Section 7.2.1, "How to Edit a Data Control."

To validate using a regular expression:

1. In the Applications window, double-click the desired data control structure file.
2. In the overview editor, click the **Attributes** navigation tab.
3. On the Attributes page, select the attribute for which you want to add the validation rule, and then click the **Validation Rules** tab.
4. In the **Validation Rules** page, click the **Add Validation Rule** icon.
5. In the Add Validation Rule dialog, in the **Rule Type** dropdown list, select **Regular Expression**.
6. In the **Operator** field, you can select **Matches** or **Not Matches**.
7. 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.
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 Section 7.5.5, "How to Create Validation Error Messages."
9. Click **OK**.

Figure 7–4 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 7–4 Regular Expression Validator Matching Email Address



7.5.3.10 What Happens When You Validate Using a Regular Expression

When you validate using a regular expression, a `<RegExpValidationBean>` tag is added to the data control structure file. Example 7–9 shows an `Email` attribute that must match a regular expression.

Example 7–9 Regular Expression Validator XML Code

```
<validation:RegExpValidationBean
  Name="Email_Rule_0"
  OnAttribute="Email"
  Pattern="[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}"
  Flags="CaseInsensitive"
  Inverse="false"/>
```

7.5.4 How to Use Groovy Expressions For Validation Rules

Groovy expressions are Java-like scripting code stored in the data control structure file. You can even change or specify values at runtime.

For more information about creating validation rules, see Chapter 7.5, "Defining Validation Rules on Attributes Declaratively."

For more information about using Groovy expressions in your business logic, see Section 7.9, "Groovy Language Support."

7.5.4.1 Validating Using a Groovy 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()` and `adf.error.warn()` methods (rather than simply returning `true` or `false`) allows you to define the message text to show to the user, and to associate a validator with a specific attribute. For more information, see Section 7.5.5.4, "Raising Error Message Conditionally Using Groovy."

Before you begin:

It may be helpful to have an understanding of validation in data control structure files. For more information, see Section 7.5, "Defining Validation Rules on Attributes Declaratively."

You may also find it helpful to understand the use of Groovy in validation rules. For more information, see Section 7.5.4, "How to Use Groovy Expressions For Validation Rules."

You will need to complete this task:

Create the desired data control structure files as described in Section 7.2.1, "How to Edit a Data Control."

To validate using a true/false expression:

1. In the Applications window, double-click the desired data control structure file.
2. In the overview editor, click the **Attributes** navigation tab.
3. On the Attributes page, select the attribute for which you want to add the validation rule, and then click the **Validation Rules** tab.
4. In the **Validation Rules** page, click the **Add Validation Rule** icon.
5. In the Add Validation Rule dialog, in the **Rule Type** dropdown list, select **Script Expression**.
6. Enter a validation expression in the field provided.
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 Section 7.5.5, "How to Create Validation Error Messages."
8. Click **OK**.

7.5.4.2 What Happens When You Validate Based on a Groovy Expression

When you create a Groovy expression, it is saved in the data object's data control structure file. The Groovy expression is wrapped by a `<TransientExpression>` tag. For some Groovy expressions, the `<TransientExpression>` tag is wrapped by a `<validation:ExpressionValidationBean>` tag as well.

7.5.4.3 Referencing Data Object Methods in Groovy Validation Expressions

You can call methods on the data object using the `adf.source.dataProvider` property of the current object. The `adf.source.dataProvider` property allows you to access the data object that is being validated.

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 example, the Groovy expression in Example 7–10 would call the `getXyzAbc()` method.

Example 7–10 Groovy Expression Calling Sample Methods

```
adf.source.dataProvider.XyzAbc
```

For a Boolean property, the same holds true but the JavaBeans component naming pattern for the getter method changes to recognize `isXyzAbc()` instead of `getXyzAbc()`. If the method on your data object does not match the JavaBeans getter method naming pattern, or if it takes one or more arguments, then you must call it like a method using its complete name.

7.5.5 How to Create Validation Error Messages

Validation error messages provide important information for the user: the message should convey what went wrong and how to fix it.

7.5.5.1 Creating Validation Error Messages

Before you begin:

It may be helpful to have an understanding of the use of validation rules in data control structure files. For more information, see Section 7.5, "Defining Validation Rules on Attributes Declaratively."

You will need to complete this task:

Create the desired data control structure files as described in Section 7.2.1, "How to Edit a Data Control."

To create validation error messages:

1. In the Applications window, double-click the desired data control structure file.
2. In the overview editor, click the **Attributes** navigation tab.
3. On the Attributes page, select the attribute for which you want to create the validation error message, and then click the **Validation Rules** tab.
4. In the **Validation Rules** page, select the validation rule that you want to edit and click the **Edit Validation Rule** icon.
5. In the Edit Validation Rule dialog, click the **Failure Handling** tab.
6. 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 in the top right corner of the Message Text field to open the Select Text Resource dialog.

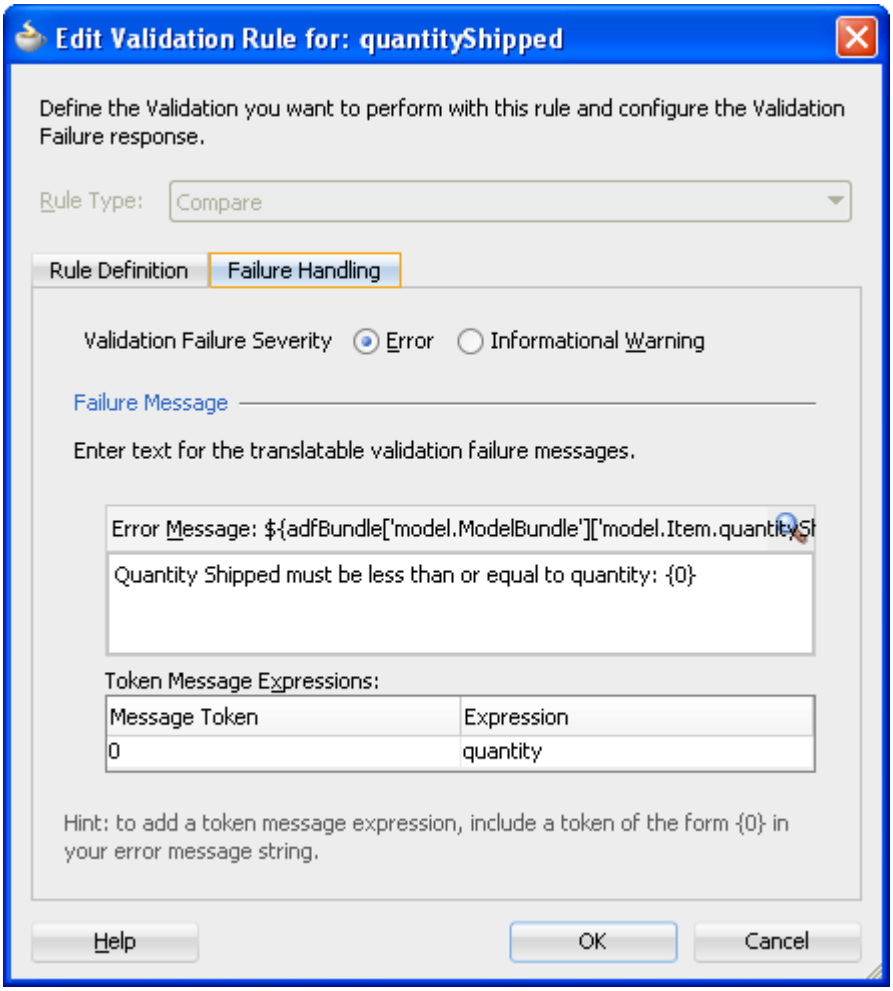
Note: The Script Expression validator allows you to enter more than one error message. This is useful if the validation script conditionally returns different error or warning messages. For more information, see Section 7.5.5.4, "Raising Error Message Conditionally Using Groovy."

7. Optionally, define a message token by entering the message token's name in curly braces ({}) within the text of the error message. An entry for the token will then appear in the **Token Message Expressions** list where you can then define an expression to determine the value of the message token.

   Figure 7–5 shows a failure message that contains a message token for a validation rule in the data control structure file. For more information on this feature, see Section 7.5.5.5, "Embedding a Groovy Expression in an Error Message."

8. Click **OK**.

Figure 7–5 Failure Handling Message for a Validation Rule



7.5.5.2 What Happens When You Create a Validation Error Message

When you create an error message for a validation rule, it is saved in a `.properties` file. The `.properties` file and the message string are referenced from the `ResId` tag attribute of the validator's tag in the data control structure file.

Example 7–11 shows the validation rule tag's `ResId` attribute that specifies the location and key of the error message. Example 7–12 shows the corresponding entry in the `.properties` file which contains the error message.

Example 7–11 Validation Error Message Reference

```
<validation:CompareValidationBean
  Name="dateShipped_Rule_0"
  ResId="${adfBundle['model.ModelBundle']['model.Ord.dateShipped_Rule_0']}"
```

Example 7–12 Validation Error String

```
model.Ord.dateShipped_Rule_0=Date Shipped cannot be before Date Ordered
```

7.5.5.3 Localizing Validation Messages

The error message is a translatable string and is managed in the same way as translatable UI hints in a message bundle file. To view the error message for the defined rule in the message bundle class, locate the key in the message bundle that corresponds to the `ResId` property in the data control structure file entry for the validator.

7.5.5.4 Raising Error Message Conditionally 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.

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.

You can use either of the `adf.error.raise()` and `adf.error.warn()` methods, depending on whether you want to throw an exception, or whether you want processing to continue, as described in Section 7.5.6, "How to Set the Severity Level for Validation Exceptions."

Example 7–13 shows a rule that will allow the input of value 5000 or under. If the value is above 5000, the error message tied to the `SALARY_TOO_HIGH_ERROR` resource bundle property is shown and validation fails. If the entered value is between 1001 and 5000, validation passes, but a warning is displayed to the user.

Example 7–13 Using Groovy to Raise an Error Message

```
if (newValue > 1000)
{
if (newValue > 5000)
{adf.error.raise("SALARY_TOO_HIGH_ERROR")
return false}
adf.error.warn("SALARY_LIMIT_WARNING")
return true
}
else
{
return true
}
```

7.5.5.5 Embedding 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 7–5 shows a failure message that contains a message token for a validation rule.

The expression shown in Figure 7–5 is a Groovy expression that returns the value of the `quantity` attribute. You can also use Groovy expressions to access attribute UI hints and other objects that are defined in the data control structure file.

You can use the Groovy expression `newValue` to return the entered value.

For more information about accessing ADF objects using Groovy, see Section 7.9, "Groovy Language Support."

7.5.6 How to Set the Severity Level for Validation Exceptions

You can set the severity level for validation exceptions to either Informational Warning or 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 an 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.

To set the severity level for validation exceptions, use the **Failure Handling** tab of the Add Validation Rule dialog.

Before you begin:

It may be helpful to have an understanding of the use of validation rules in data control structure files. For more information, see Section 7.5, "Defining Validation Rules on Attributes Declaratively."

You will need to complete this task:

Create the desired data control structure files as described in Section 7.2.1, "How to Edit a Data Control."

To set the severity level of a validation exception:

1. In the Applications window, double-click the desired data control structure file.
2. In the overview editor, click the **Attributes** navigation tab.
3. On the Attributes page, select the attribute for which you want to create the validation error message, and then click the **Validation Rules** tab.
4. In the **Validation Rules** page, select the validation rule that you want to edit and click the **Edit Validation Rule** icon.
5. In the Edit Validation Rule dialog, click the **Failure Handling** tab and select the option for either **Error** or **Informational Warning**.

6. Click **OK**.

7.6 Filtering Result Sets with Named Criteria

JDeveloper enables you to create named criteria for data control structure files for JPA-based data controls. Named criteria can be used in the application's data model and can be exposed to users as seeded queries in search forms.

You can specify named criteria in order to filter results to display. The named criteria object is a row set of one or more named criteria rows, whose attributes mirror those in the corresponding data object. The named criteria definition comprises query conditions that function like the `WHERE` clause of an SQL query.

In the result set of a named criteria, the data type of each attribute is `String`, which enables the use of Query-by-Example operators. For example, this allows the user to enter conditions such as "`OrderId > 304`".

You use the Named Criteria page of the overview editor to define named criteria for specific data control structure files.

7.6.1 Use Case for Named Criteria

You create named criteria definitions when you need to filter individual accessor results. Named criteria that you define at design time can be used for easy creation of Query-by-Example search forms that allow the end user to supply values for attributes of the target data control structure file.

For example, the end user might want to be able to input the value of a customer name and the date to filter the results in a web page that displays the rows of a `CustomerOrders` collection. To satisfy that case, the web page designer can easily create a search form based on named criteria that have been created for this purpose and made available in the Data Controls panel. For more information about the utilizing the named criteria in the Data Controls panel, see "Creating ADF Databound Search Forms" in *Developing Fusion Web Applications with Oracle Application Development Framework*.

7.6.2 How to Create Named Criteria Declaratively

To define named criteria for a JPA-based data control structure file, you open the data control structure file in the overview editor and use the Named Criteria page. A dedicated editor that you open from the Named Criteria page helps you to build the equivalent of a `WHERE` clause using attribute names (as opposed to SQL column names). You may define multiple named criteria for each data object.

Each named criteria definition consists of the following elements:

- One or more named criteria rows consisting of an arbitrary number of named criteria groups or an arbitrary number of references to another named criteria already defined for the current data control structure file.
- Optional named criteria groups consisting of an arbitrary number of named criteria items.
- Named criteria items 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 Section 7.9, "Groovy Language Support."

Named 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 named criteria item you add.
- `OR` conjunctions specify that the query results meet either or both joined conditions. This is the default for named criteria groups.

Additionally, you may create nested named criteria to have more control over the logical conjunctions among the various named criteria items. A nested named criteria group consists of an arbitrary number of nested named criteria items. The nested criteria place restrictions on the rows that satisfy the criteria under the nested criteria's parent named 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 named criteria with the first group with one item (`Salary > 3000`) and a nested named 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 named criteria. For more information, see Section 7.6, "Filtering Result Sets with Named Criteria."

You will need to complete these tasks:

- Create an EJB data control as described in Section 3.3.1, "How to Create EJB Data Controls" or a JPA-based bean data control as described in Section 4.3.1, "How to Create a JPA-Based Bean Data Control" and select the **Support Named Criteria** checkbox in the wizard when creating the data control.
- Create the desired data control structure files as described in Section 7.2.1, "How to Edit a Data Control."

To define a named criteria:

1. In the Applications window, double-click the data control structure file for which you want to create the named criteria.
2. In the overview editor, click the **Named Criteria** navigation tab.
3. In the Named Criteria page, expand the **Named Criteria** section and click the **Create new view criteria** button.
4. In the Create Named Criteria dialog, enter the name of the named criteria to identify its usage in your application.
5. Click one of these **Add** buttons to define the named criteria.
   - **Add Item** to add a single criteria item. The editor will add the item to the hierarchy beneath the current group or named 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 data control structure file 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 named criteria that you intend to define. This selection is an alternative to adding a named criteria that already exists in the data control structure file. When you add a new named criteria, the editor inserts the `AND` conjunction into the hierarchy. You can change the conjunction as desired. Each time you add another named criteria, the editor nests the new named criteria beneath the current named criteria selection in the hierarchy. The root node of the hierarchy defines the named criteria that you are currently editing.
- **Add Named Criteria** to add a named criteria that has already been defined in the data control structure file.

Note: Search forms that the UI designer will create from named criteria are not able to use named criteria that contain other named criteria.

---

6. Select a named criteria item node in the Named Criteria tree and define the added node in the **Criteria Item** section.
7. Select the desired **Attribute** for the criteria item. By default the editor adds the first one in the list.
8. Select the desired **Operator**.

   The list displays only the operators that are appropriate for the selected attribute. 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.
9. Select the desired **Operand** for the named 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 named 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 the wildcard characters `*` or `%`.
   - Select **Bind Variable** when you want the value to be determined at runtime using a bind variable. Click **New** to display the Bind Variable dialog that lets you create a new bind variable for the data control structure file. For more information about creating bind variables, see Section 7.6.4, "How to Use Bind Variables in Named Criteria."
10. For each item, group, or nested named criteria that you define, optionally change the default conjunction to specify how the selection should be joined.
    - **AND** specifies that the query results meet both joined conditions. This is the default for each named criteria item or view nested named criteria that you add.
    - **OR** specifies that the query results meet either or both joined conditions. This is the default for named criteria groups.
11. Optionally, to allow filtering based on the case of the runtime-supplied value, deselect the **Ignore Case** option. It is selected by default, preventing such filtering.

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.

12. In the **Validation** dropdown list, decide whether the named criteria item is a required or an optional part of the attribute value comparison in the generated `WHERE` clause.
    - **Selectively Required** means that the generated `WHERE` clause will ignore the named criteria item at runtime if no value is supplied and there exists at least one criteria item at the same level that has a criteria value. Otherwise, an exception is thrown.
    - **Optional** means the named criteria is added to the `WHERE` clause only if the value is non-`NULL`. The default **Optional** for each new named 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.
13. If the named criteria uses a bind variable as the operand, decide whether the `IS NULL` condition is 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 named criteria to return a result even when the bind variable value is not supplied at runtime. When validation is set to **Required** or **Optionally Required**, the named criteria expects to receive a value and thus this option to ignore null values is disabled.

      For example, leaving this option selected for a bind variable that is used in a user search form would enable a user to see results from a search without having to fill in a value for the field with that bind variable.
    - Deselect **Ignore Null Values** when you expect the named criteria to return a null result when the bind variable value is not supplied at runtime.

      Note that the validation settings **Required** or **Optionally Required** also remove the null value condition but support a different use case. They should be used in combination with **Ignore Null Values** feature to achieve the desired runtime behavior.
14. Click **OK** create the named criteria and return to the overview editor.

7.6.3 What Happens When You Create a Named Criteria

When you create a named criteria, the named criteria definition is added to the data control structure file and appears by name on the Named Criteria page of the overview editor.

To view XML code for the named criteria, open the source editor for the data control structure file. Each named criteria definition contains one or more `<ViewCriteriaRow>` elements corresponding to the number of groups that you define in the Create Named Criteria dialog.

7.6.4 How to Use Bind Variables in Named Criteria

Bind variables provide you with the means to supply attribute values that are calculated at runtime to the named criteria.

If the named criteria is to be used in a seeded search, 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 the search form.

Named criteria execution need not require the bind variable value if the named criteria item for which the bind variable is assigned is not required. To enforce this desired behavior, the Bind Variable dialog lets you specify whether a bind variable is required or not.

You can define a default value for the bind variable or write scripting expressions for the bind variable that include dot notation access to attribute property values. Expressions are based on the Groovy scripting language, as described in Section 7.9, "Groovy Language Support."

To add a named bind variable to a named criteria, use the Named Criteria page of the overview editor for the data control structure file. You can define as many bind variables as you need.

Before you begin:

It may be helpful to have an understanding of named criteria. For more information, see Section 7.6, "Filtering Result Sets with Named Criteria."

You will need to complete this task:

Create a named criteria as described in Section 7.6.2, "How to Create Named Criteria Declaratively."

To create a named bind variable

1. In the Applications window, double-click the data control structure file.
2. In the overview editor, click the **Named Criteria** navigation tab.
3. On the Named Criteria page, expand the **Bind Variables** section and click the **Create new bind variable** button.
4. In the New Variable dialog, enter the name and data type for the new bind variable.

   Because the bind variables share the same namespace as data control structure file attributes, specify names that don't conflict with existing attribute names.
5. 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 **Default Value** field.
   - When you want to define a default value, select the **Literal** value type and enter the literal value in the **Default Value** field.
6. Optionally, click the **UI Hints** tab and specify hints like **Label Text**, **Format Type**, **Format** mask, and others.

   The view layer will use bind variable control hints when you build user interfaces like search pages that allow the user to enter values for the named bind variables. The **Updatable** option 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, then its value can only be changed programmatically by the developer.
7. Click **OK**.

7.6.5 What Happens When You Use Bind Variables in Named Criteria

Once you've added one or more named bind variables to a data control structure file, 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 data object's data control structure file. If you have defined UI hints for the bind variables, this information is saved in the data model project's message bundle file along with other UI hints for the data control structure file.

7.6.6 What You May Need to Know About Nested Expressions

Search forms that the UI designer will create from named criteria are not able to work with all types of nested expressions. Specifically, search forms do not support expressions with directly nested named criteria. This type of nested expression defines one named criteria as a direct child of another named criteria. Query search forms do support nested expressions where you nest the named criteria as a child of a criteria item which is itself a child of a named criteria. For more information about using named criteria to create search forms, see "Creating ADF Databound Search Forms" in *Developing Fusion Web Applications with Oracle Application Development Framework*.

7.6.7 How to Set User Interface Hints on Named Criteria

Named criteria that you create for data control structure file collections can be used by the web page designer to create Query-by-Example search forms. Web page designers select your named criteria from the Data Controls panel to create search forms for the web application. In the web page, the search form utilizes an ADF Faces query search component that will be bound initially to the named criteria selected in the Data Controls panel. At runtime, the end user may select among all other named criteria that appear in the Data Controls panel. Named criteria that the end user can select in a search form are known as *developer-seeded searches*. The query component automatically displays these seeded searches in its **Saved Search** dropdown list. For more information about creating search forms and using the ADF query search component, see "Creating ADF Databound Search Forms" in *Developing Fusion Web Applications with Oracle Application Development Framework*.

Because developer-seeded 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 criteria. At runtime, the query component's behavior will conform to the selections you make for the following seeded search properties:

To create a seeded search for use by the ADF query search component, you select **Show In List** on 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 named criteria in their search form.

Before you begin:

It may be helpful to have an understanding of named criteria. For more information, see Section 7.6, "Filtering Result Sets with Named Criteria."

You will need to complete this task:

Create the named criteria, as described in Section 7.6.2, "How to Create Named Criteria Declaratively."

To configure a named criteria for the user interface:

1. In the Applications window, double-click the data control structure file that defines the named criteria you want to use as a seeded search.
2. In the overview editor, click the **Named Criteria** navigation tab.
3. On the Named Criteria page, select the named criteria that you want to allow in seeded searches and click the **Edit** icon.
4. In the 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 seeded search in its **Saved Search** dropdown list.
5. Enter a user-friendly display name for the seeded search to be added to the query component Saved Search dropdown list.

   When left empty, the named criteria name displayed in the Edit View Criteria dialog will be used by the query component.
6. Optionally, enable **Query Automatically** when you want the query component to automatically display the search results whenever the end user selects the seeded search from the Saved Search dropdown list.

   By default, no search results will be displayed.
7. Optionally, set the **Rendered Mode** property for each named criteria item in order to determine whether the item is displayed for the user in basic mode or advanced mode.

   Note: When your named 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 named 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 named 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.
8. Optionally, use the **Show Operators** dropdown list to configure whether the query component renders individual criteria items when the end user toggles the search from between basic and advanced mode.

   By default, all named criteria items defined by the seeded search will be displayed in either mode.
9. If a rendered criteria item is of type `Date`, you must also define UI hints for the corresponding data object attribute. Set the data 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. For more information, see Section 7.3.6, "How to Set UI Hints on Attributes."
10. Click **OK**.

7.6.8 How to Create a Named Criteria Based on Multiple JPA Entities

When you create a named criteria for an entity class on a JPA-based data control, only attributes related to that particular entity are available for filtering. However, it is also possible to create a named criteria based on a custom bean and a JPA query that aggregates data from multiple tables that are related by foreign keys.

Before you begin:

It may be helpful to have an understanding of named criteria. For more information, see Section 7.6, "Filtering Result Sets with Named Criteria."

To create a named criteria based on multiple JPA entities:

1. Create JPA entity classes for the tables that you want to filter and create an EJB session bean over those entities. For more information, see "How to Work with an EJB Business Services Layer" in *Developing Applications with Oracle JDeveloper*.
2. Create a custom bean and populate it with simple getters and setters for the various attributes (from multiple entities) that you want to filter. For example, the getter and setter corresponding to the `name` attribute might look like the following:

```
public void setName(String name) {
    this.name = name;
}

public String getName() {
    return name;
}
```

3. Add an accessor method to the session facade that creates a list of the custom beans based on a JPA query to the database.

   Example 7–14 shows a such method that you might add to a session bean in order to create a list where each object consists of the `id` and `name` columns from the `Customer` entity and the `dateOrdered` and `dateShipped` columns from the `Ord` entity.

Example 7–14 Custom JPA Query on Which to Base a Multi-Entity Named Criteria

```
public List<CustomBean> getCustomBeanFindAll() {
     String queryString =
            "select c.id, c.name, o.date_ordered, o.date_shipped \n" +
            "from s_customer c, s_ord o where o.customer_id = c.id";
     Query genericSearchQuery = em.createNativeQuery(queryString, "CustomQuery");
     List resultList = genericSearchQuery.getResultList();
     Iterator resultListIterator = resultList.iterator();
     List<CustomBean> customList = new ArrayList();
     while (resultListIterator.hasNext()) {
         Object col[] = (Object[])resultListIterator.next();
         CustomBean custom = new CustomBean();
         custom.setCustomerId((BigDecimal)col[0]);
         custom.setName((String)col[1]);
         custom.setDateOrdered((Date)col[2]);
         custom.setDateShipped((Date)col[3]);
     customList.add(custom);
     }
     return customList;
}
```

4. Create an EJB data control on the session bean as described in Section 3.3.1, "How to Create EJB Data Controls" and be sure to select the **Support Named Criteria** checkbox in the wizard when creating the data control.
5. Create a data control structure file for the custom bean as described in Section 7.2.1, "How to Edit a Data Control."
6. Create the named criteria on the custom bean as described in Section 7.6.2, "How to Create Named Criteria Declaratively."

7.7 Creating List of Values Objects

In database applications, some columns in a given database table might have a restricted set of possible values that are defined in another database table. For example, an `Orders` database table might contain a `Country` column that must be filled in only with values derived from a `Countries` database table. To make it easier to work with such data from other tables, JDeveloper enables you to create *list of value* objects (LOVs) to reference data from one object in another object. By establishing these list of values relationships in the data model, you can then easily create selection lists, combo boxes, and other list of value UI components by dragging the LOV attribute from the Data Controls panel on to a page.

For JPA-based data controls, you can also apply any named criteria that you have defined on the object that is the list source in order to limit which objects appear in the list at runtime. To create named criteria, see Section 7.6, "Filtering Result Sets with Named Criteria".

7.7.1 How to an Create LOV for an Attribute

You define LOV objects in data control structure files.

Before you begin:

It may be helpful to have an understanding of list of value components. For more information, see Section 7.7, "Creating List of Values Objects."

You will need to complete this task:

Create data control structure files for both the object that will contain the LOV definition and the object which will be the data source for the LOV, as described in Section 7.2.1, "How to Edit a Data Control."

To create an LOV for an attribute

1. In the overview editor for the data control structure file that will contain the LOV definition, click the **List of Values** navigation tab and click the **Add list of values object**.
2. In the Create List of Values dialog, perform the following steps to apply the named criteria:
   a. In the List of Values Name field, type a name for the LOV object.
   b. In the Base Object Field tree, select the data control object that will be used to access the list of values. For example, you might choose a structured attribute node that has a `OneToMany` relationship with a collection.
   c. In the List Data Source tree, select the data control object for the source of the values.

This accessor should not be otherwise used in the application. If this accessor is used elsewhere, create a new accessor method on the data control delegate that duplicates the functionality and use that as the list data source.

Note: When using an LOV to select a JPA entity, the JPA entities returned by the list data source accessor will be compared, using `equals()`, to the property bound to the LOV. If the JPA entities in the list data source are guaranteed to be in a managed, and not detached, state by their `EntityManager`, the comparison will work correctly. If they might be in a detached state, then the default `equals()` method comparison may not work as expected and the entity class should override the `equals()` and `hashCode()` methods to at least include the entity's ID field.

Similarly, you need to override the `equals()` method when the underlying session bean is stateless. Each method call in a stateless session bean employs its own persistence context, so the default `equals()` implementation will return `false` when comparing instances returned from separate method calls.

You can generate code to override these two methods by selecting the entity class in the Applications window, right-clicking the entity's node in the Structure window, and choosing **Generate equals() and hashCode()**. However, this approach is not recommended because JPA also has dependencies on these methods. If you do override these methods, make sure that your implementation of `equals()` compares all non-transient properties in the entities to minimize any unexpected behavior. The recommended approach is to avoid the use of detached entities in your ADF web application, by using a stateful session bean with an extended persistence context. Using this model type, you can avoid multiple instances of the same logical entity, and the `equals()` method does not need to be overridden to compare entities based on their immutable identity.

---

d. Click **OK**.

3. Optionally, for JPA-based data controls, click the **Edit View Attributes** icon and apply any named criteria from the data source that you would like to use to limit the displayed values. This icon is located on the right side of the page between the List Data Source and List Return Values panels.
4. In the overview editor, click List UI Hints tab and perform the following steps:
   a. In the Default List Type dropdown list, select the component to display the list of values. See the "List Component Types for List Type UI Hint" table in *Developing Fusion Web Applications with Oracle Application Development Framework* for a description of the types of components available.
   b. Optionally, in the Display Attributes section, change the attributes to be displayed in the list.
   c. Optionally, configure any of the other options that are available for the type of list you are using.
5. Optionally, click the Accessor UI Hints tab to set UI hints for things such as the attributes displayed in the list and whether a user's selection is automatically submitted.

7.7.2 What Happens When You Create an LOV

When you create an LOV object, the following things happen:

- If not already present, an attribute node for the LOV object appears as a subnode of its base object in the Data Controls panel. You can use this node to create a model-driven list UI component by dragging it to the page. You can also drag the attribute's parent collection to the page and create a form or table in which that attribute is represented by a list component.

  Note: This attribute node might already appear in the Data Controls panel before the creation of the LOV object if a mapping to the list source is not already included for the attribute in the object definition.

  If the object definition includes mapping to the list source (such as in the form of a `@JoinColumn` annotation in a JPA entity class), by default the Data Controls panel represents the attribute as a structured attribute. However, by default, UI components for this structured attribute are not generated when you drag the parent collection to a page to create an ADF Form or an ADF Table. Therefore, creating the LOV object for an attribute that is mapped to another table simplifies the subsequent creation of forms and tables that you want to include model-driven selection lists.

- The following elements are added to the data control structure file for the collection that contains the LOV attribute:
  - A `<PDefAttribute>` element, which specifies the attribute, points to the list binding element that defines the LOV behavior, and specifies the component type to display in the web page.
  - A `<ViewAccessor>` element, which is the mechanism that is used to obtain the list of possible values from the row set of the data source.
  - A `<ListBinding>` element which defines the behavior of the LOV, such as query limit and whether to display the list will include a blank entry.

7.8 Testing Data Object Metadata Using the Oracle ADF Model Tester

Before you start designing the views for your application, you can use the Oracle ADF Model Tester to test various aspects of your model for EJB and bean data controls. For example, if you have added validation rules or UI hints to your model, you can test them before binding those objects to a page. Even after you have your UI pages constructed, the Oracle ADF Model Tester can assist you in diagnosing problems when they arise. You can use the Oracle ADF Model Tester to narrow down whether a problem lies in the business service layer or not.

7.8.1 How to Run the Oracle ADF Model Tester

To test the metadata that you have defined in your data control, 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. For more information, see Section 7.8, "Testing Data Object Metadata Using the Oracle ADF Model Tester."

You need to complete this task:

Create an EJB or bean data control, as described in Section 3.3.1, "How to Create EJB Data Controls" or Section 4.3.1, "How to Create a JPA-Based Bean Data Control."

Note: For EJB data controls, the tester does not work for the local session bean interface. However, you can work around that limitation by creating a Java service facade class that is based on the same persistence unit as the session bean, creating a data control for that class, and using that data control in the tester. Since this Java service facade will use the same persistence unit as the EJB session bean, the two data controls will use the same data control structure files for the entities.

For more information on creating a Java service facade class, see Section 4.2.4, "How to Create a Service Facade for a JPA-Based Bean Data Control."

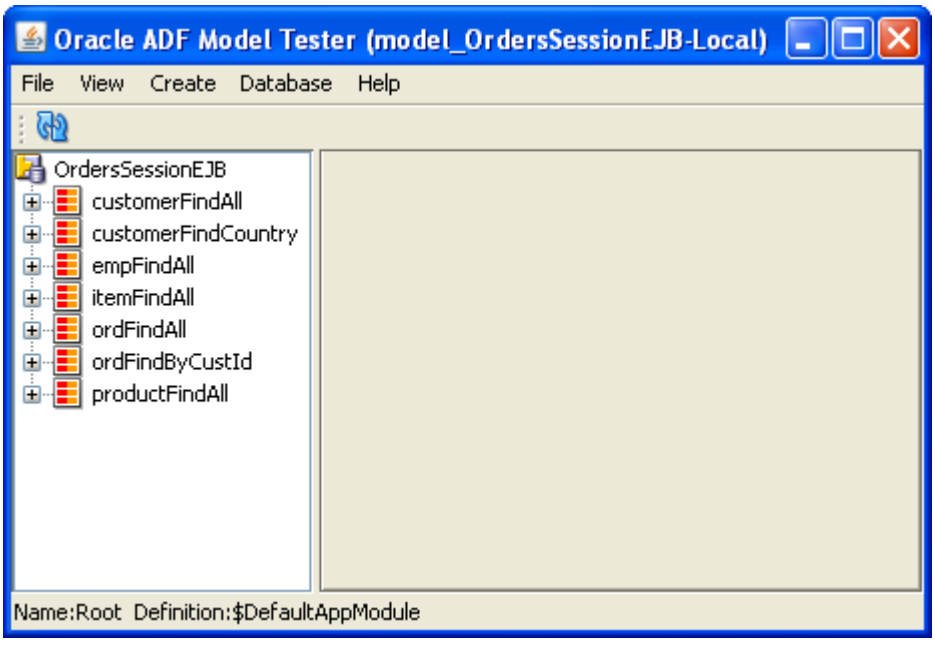
Using this approach provides the additional benefit of reducing the overhead of the integrated WebLogic Server instance. The instance has to run a process for the tester, but it does not have to run the session bean itself.

To test the bean metadata in your application:

1. In the Applications window, double-click the **DataControls.dcx** file.
2. In the overview editor, right-click the node for the data control that you want to test and choose **Run**.

   The Oracle ADF Model Tester opens, as shown in Figure 7–6

Figure 7–6 Oracle ADF Model Tester

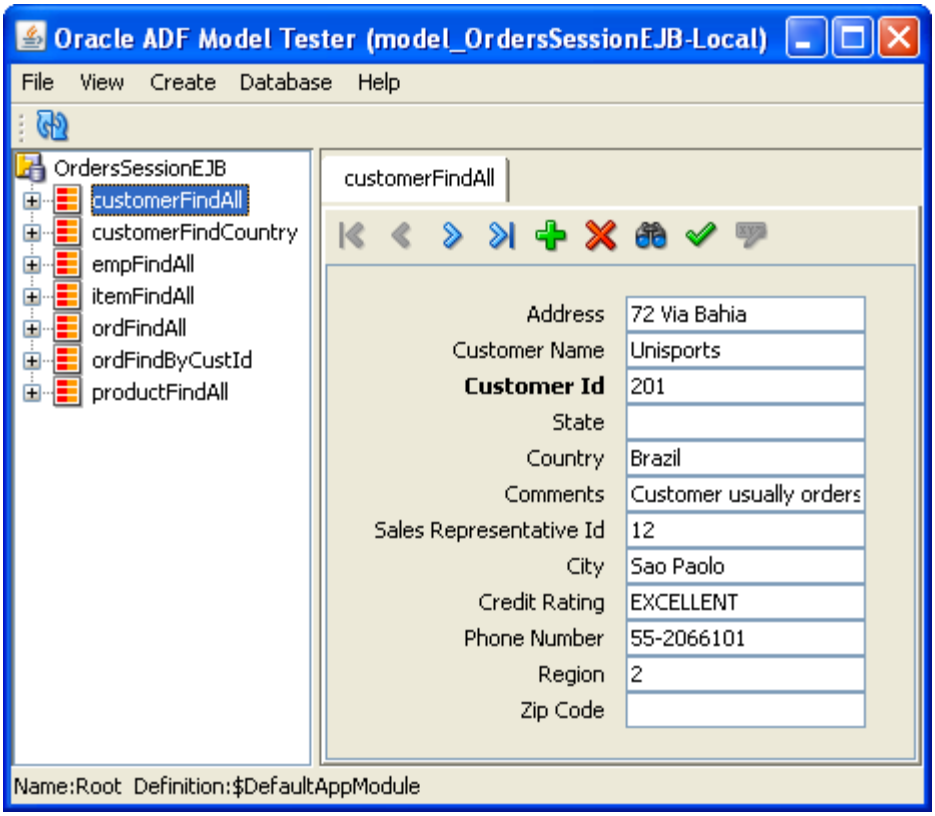


3. Double-click a collection object to start testing the metadata for that object.

7.8.2 What Happens When You Use the Oracle ADF Model Tester

When you launch the Oracle ADF Model Tester, JDeveloper starts the tester in a separate process and the Oracle ADF Model Tester window appears. The tree at the left of the window displays all of the collections in your data model. If the data model defines master-detail view instance relationships, the tree will display them as parent and child nodes. After you double-click the desired collection, the Oracle ADF Model Tester will display a data view page to inspect the query results. For example, Figure 7–7 shows the **customerFindAll** collection that has been double-clicked to display the first record for this collection in the data view page on the right.

Figure 7–7 Oracle ADF Model Tester with the customerFindAll Collection Detail



The following are some of the features of the Oracle ADF Model Tester:

- You can validate that the UI hints based on the Label Text hint and format masks are defined correctly. (For more information on setting UI hints, see Section 7.3.6, "How to Set UI Hints on Attributes.")
- 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, you can select a named criteria like `CustomerInfoCriteria` and enter a query criteria like "`H%`" for a `LastName` attribute and click **Find** to narrow the search to only those users with a last name that begins with the letter `H`.

Table 7–2 gives an overview of the operations that the Oracle ADF Model Tester toolbar buttons perform.

Table 7–2 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**	Runs the commit operation on any pending transactions. However, for JPA-based data controls, the changes are not actually committed to the database. This button is only available when transactional methods are implemented on the service.
	**Discard all changes since last save**	Discards any pending transactions and restores the original values. This button is only available when transactional methods are implemented on the service.
	**Specify view criteria**	Displays the View Criteria dialog that you can use to create and apply named criteria to the result set.
	**Validate row**	Validates the current row by applying validation rules defined in the data control structure file. Disabled unless at least one field is editable.

7.8.3 How to Test Business Layer Validation

Depending on the validation rules you have defined, you can try entering invalid values to trigger and verify validation exceptions.

Before you begin:

It may be helpful to have an understanding of the Oracle ADF Model Tester. For more information, see Section 7.8, "Testing Data Object Metadata Using the Oracle ADF Model Tester."

You will need to complete this task:

Start the tester as described in Section 7.8.1, "How to Run the Oracle ADF Model Tester."

To test business layer validation:

- In the tester, enter a value for an attribute and click the **Validate Row** icon.

  For example, if you have defined a range validation rule for an attribute, enter a value outside that range, and click the **Validate Row** icon. You should see an error similar to following:

  ```
  (oracle.jbo.AttrSetValException) Valid product codes are between 100 and 999
  ```

7.8.4 How to Test Row Creation and Default Value Generation

You can use the Oracle ADF Model Tester to verify that any default values for attributes are properly generated when you create a new row.

Before you begin:

It may be helpful to have an understanding of the Oracle ADF Model Tester. For more information, see Section 7.8, "Testing Data Object Metadata Using the Oracle ADF Model Tester."

You may also find it helpful to understand attributes in data control structure files. For more information, see Section 7.3, "Working with Attributes."

You will need to complete this task:

Start the tester as described in Section 7.8.1, "How to Run the Oracle ADF Model Tester."

To test row creation and default value generation:

1. In the Oracle ADF Model Tester toolbar, click the **Insert a new row** button to create the blank row.

   Any fields that have a declarative default value will appear with that value in the blank row.

2. In the tester, enter all required fields and click the **Commit** button.

7.8.5 How to Test Named Criteria Using the Oracle ADF Model Tester

The Oracle ADF Model Tester enables you to test your data model using existing named criteria and by querying with ad hoc criteria.

Before you begin:

It may be helpful to have an understanding of the Oracle ADF Model Tester. For more information, see Section 7.8, "Testing Data Object Metadata Using the Oracle ADF Model Tester."

You may also find it helpful to understand named criteria. For more information, see Section 7.6, "Filtering Result Sets with Named Criteria."

You will need to complete this task:

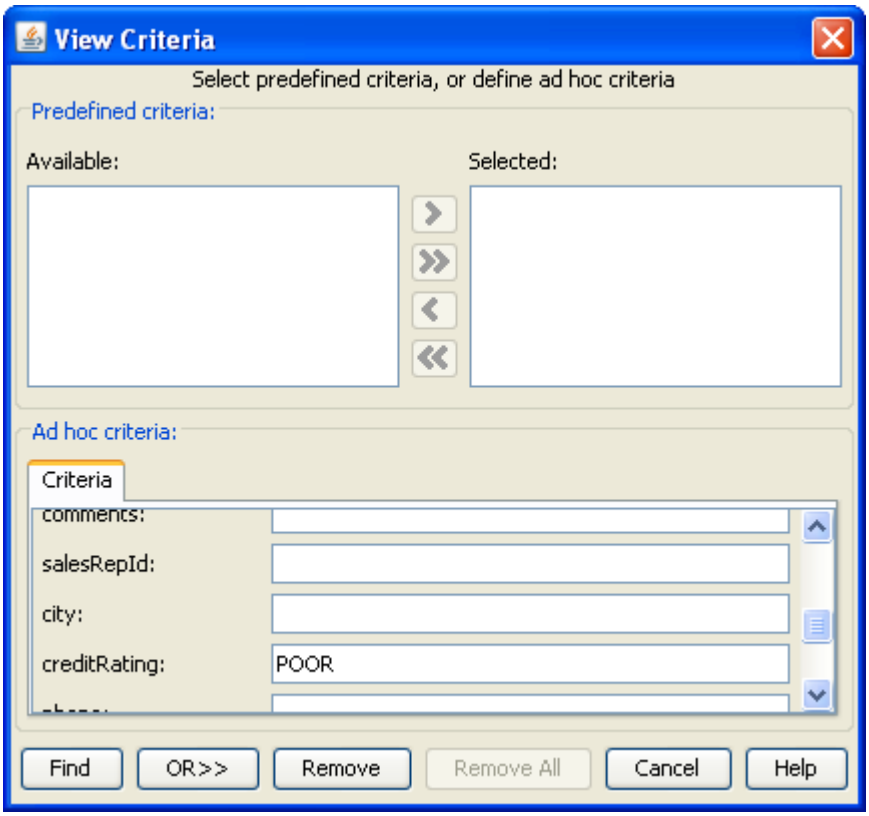
Start the tester as described in Section 7.8.1, "How to Run the Oracle ADF Model Tester."

To test named criteria and ad hoc query criteria using the Oracle ADF Model Tester:

1. In the Oracle ADF Model Tester, double-click the collection that you want to filter.
2. Click the **Specify View Criteria** toolbar button to test the named criteria.
3. In the View Criteria dialog, perform one of the following tasks:
   - To test a named criteria that you added to a data control structure file in your project, shuttle that criteria to the **Selected** list and click **Find**. Any additional criteria that you enter in the **Ad hoc criteria** section will be added to the filter.
   - To test ad hoc criteria attributes from a single named criteria row, enter the desired values for the named criteria and click **Find**.

For example, Figure 7–8 shows the filter to return all customers who possess a credit rating of `POOR`.

Figure 7–8 Oracle ADF Model Tester View Criteria Dialog



7.8.6 How to Update the Oracle ADF Model Tester to Display Project Changes

When you are testing your data control, you can iteratively modify the data control and retest it without redeploying your whole data model project. You can merely close and reopen the Oracle ADF Model Tester in order to reload changes that you have made to the data control structure files.

If you change Java code or any other files that are packaged in the data model project JAR file, you also need to rebuild and redeploy that JAR file to the internal application server.

Before you begin:

It may be helpful to have an understanding of the Oracle ADF Model Tester. For more information, see Section 7.8, "Testing Data Object Metadata Using the Oracle ADF Model Tester."

You will need to complete this task:

Start the tester as described in Section 7.8.1, "How to Run the Oracle ADF Model Tester."

To reload the data model metadata in the running Oracle ADF Model Tester:

1. In the Oracle ADF Model Tester, test the data model and determine any changes you want to make.

2. In JDeveloper, make the desired changes to your project.
3. If you have made any changes to Java classes or any other artifacts that are part of the session bean's JAR file, rebuild the project. (For example, you can right-click the data model project in the Applications window and choose **Rebuild *ProjectFileName*** The rebuilt JAR file is then deployed to the server.)

   If you have only made changes to data control structure files, you do not need to rebuild the project.
4. Close the Oracle ADF Model Tester.
5. Start the tester again.

7.8.7 How to Test Alternate 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.

Testing the language message bundles in the Oracle ADF Model Tester lets you verify that the translations of the data control 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 `04/12/2007` to `12/04/2007`).

Before you begin:

It may be helpful to have an understanding of the Oracle ADF Model Tester. For more information, see Section 7.8, "Testing Data Object Metadata Using the Oracle ADF Model Tester."

To specify a default language for the Oracle ADF Model Tester:

1. From the **Tools** menu, choose **Preferences**.
2. Expand **ADF Business Components** in the selection panel, 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.
4. Start the tester as shown in Section 7.8.1, "How to Run the Oracle ADF Model Tester."
5. In the tester's Locale menu, select a locale.

7.9 Groovy Language Support

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 supporting syntax to manipulate collections, Strings, and JavaBeans. Groovy language expressions are dynamically compiled and are executed at runtime. Any Groovy expressions that you create for an ADF application are stored in the data control structure files of the data objects for which they are defined.

Oracle ADF supports the use of the Groovy scripting language in places where access to data control objects is useful, including attribute validators, attribute default values, transient attribute value calculations, bind variable default values (in named criteria filters), and placeholders for error messages (in validation rules). Additionally, Oracle ADF provides a limited set of built-in keywords that can be used in Groovy expressions.

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

- Define a Script Expression validator (see Section 7.5.4.1, "Validating Using a Groovy Expression") or a Compare validator (see Section 7.5.3.1, "Validating Based on a Comparison").
- Define error message tokens for handling validation failure (see Section 7.5.5.5, "Embedding a Groovy Expression in an Error Message").
- Handle conditional execution of validators (see Section 7.5.5.4, "Raising Error Message Conditionally Using Groovy").
- Define the default value and optional recalculate condition for a data object attribute (see Section 7.3.4, "How to Define a Default Value Using a Groovy Expression").
- Determine the value of a transient attribute of a data object's data control structure file (see Section 7.4, "Adding Transient Attributes to a Data Object").

To perform these tasks in JDeveloper, you use expression editor dialogs that are specific to the task. For example, when you want to create a default value for a transient attribute, you use the attribute's Edit Expression Editor dialog to enter an expression that determines a runtime value for the attribute. The same dialog also lets you specify when the value should be calculated (known as a *recalculate condition*). Although expressions cannot be verified at design time, all expression editors let you test the syntax of the expression before you save it.

For more information about the Groovy language, refer to the following website:

- `http://groovy.codehaus.org/`

7.9.1 How to Reference ADF Objects 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. The accessible Oracle ADF objects consist of the following:

- `adf.context` - to reference the `ADFContext` object.
- `adf.object.dataProvider` - to reference the data object on which the expression is being applied. Other accessible member names come from the context in which the Groovy script is applied.
  - Data object attributes and methods: The context is the data object's data control structure file. Through this object you can reference any attributes defined in the data control structure file as well as any attributes and methods that are inherited from the data object.
  - Script validation rules: The context is the validator object (`JboValidatorContext`) merged with the data object on which the validator is applied. For details about keywords that you can use in this context, see Section 7.9.2, "How to Reference ADF Methods and Attributes in Groovy Expressions."
- `adf.error` - in validation rules, to access the error handler that allows the validation expression to generate exceptions or warnings

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

- `adf.currentDate`

- `adf.currentDateTime`

You can use the following built-in aggregate functions on rows of data:

- *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).

7.9.2 How to Reference ADF Methods and Attributes in Groovy Expressions

The simplest example of referencing data control members, including methods and attributes that the data object and the data object's data control structure file define, is to reference attributes that exist in the same data 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 a data 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 data control structure file 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, you can invoke methods as part of your expression. For example, to define an attribute default value:

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

A method reference requires the prefix `adf.object.dataProvider`, which allows you to reference the same object that defines the attribute on which the expression is applied.

Note that when you want to reference the method of a data object in a validation rule, you use `source` instead of `object`.

```
adf.source.dataProvider.getDefaultSalaryForGrade()
```

Use of the `source` prefix is necessary in validators because the `object` keyword implies the validation rule object instead of the data 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 < adf.source.dataProvider.getMaxSalaryForGrade(Job)
}
else
return true
```

A

Data Control Feature Comparison

This appendix provides a brief comparison of how data access features are implemented for each type of data control.

The type of **data control** that you choose to use will impact how you implement data access features. Table A–1 provides a comparison of how you implement some commonly used data access features for each type of data control.

Table A–1 Comparison of Feature Implementation in Data Controls

	ADF Business Components Data Control	Bean Data Control	EJB Data Control	Web Service Data Control	URL Service Data Control	Placeholder Data Control
af:Query	Declarative	Declarative	Declarative	Implemented programmatically	Not available	Not available
af:quickQuery	Declarative	Declarative	Declarative	Implemented programmatically	Not available	Not available
af:inputComboListOfValues	Declarative	Declarative	Declarative	Implemented programmatically	Not available	Declarative
af:Calendar	Declarative	Implemented programmatically	Implemented programmatically	Implemented programmatically	Not available	Not available
af:Media	Declarative	Implemented programmatically	Implemented programmatically	Implemented programmatically	Not available	Not available
Table filtering	Declarative	Declarative	Declarative	Not available	Not available	Not available
Range and scrollable paging	Declarative	Declarative	Declarative	Not available	Not available	Not available
UI Hints	Declarative	Declarative	Declarative	Declarative	Declarative	Declarative
Validation Rules	Declarative	Declarative	Declarative	Declarative	Declarative	Not available
Criteria-based fetching	Declarative	Declarative	Not available	Not available	Not available	Only implicit criteria available

Table A–1 (Cont.) Comparison of Feature Implementation in Data Controls

	ADF Business Components Data Control	Bean Data Control	EJB Data Control	Web Service Data Control	URL Service Data Control	Placeholder Data Control
List of Value (LOV) Components	Declarative	Declarative	Declarative	Declarative	Declarative	Declarative
Commit and Rollback Support	Declarative	Declarative, depends on implementation	Declarative, depends on implementation	Not available	Not available	Implemented programmatically
Failover Support	Declarative	Declarative, depends on implementation	Declarative, depends on implementation	Not available	Not available	Not available

Features that are listed as "declarative, depends on implementation" are available if the underlying business service provides an appropriate code pattern. Components that are listed as "implemented programmatically" in the table can be implemented using the necessary Java classes required to implement a business model that can be used by the specific data-entry component. For more information, refer to the Javadoc for the appropriate classes.