

Oracle® Fusion Middleware

Java EE Developer's Guide for Oracle Application Development Framework

11g Release 2 (11.1.2.2.0)

E17272-03

April 2012

Documentation for Oracle Application Development Framework (Oracle ADF) developers that describes how to develop and deploy web-based applications using Java EE, ADF Model, ADF Controller, and ADF Faces Rich Client components.

Oracle Fusion Middleware Java EE Developer's Guide for Oracle Application Development Framework 11g Release 2 (11.1.2.2.0)

E17272-03

Copyright © 2010, 2012 Oracle and/or its affiliates. All rights reserved.

Primary Author: Robin Whitmore, Peter Jew, Patrick Keegan

Contributing Author: Lynn Munsinger, Jeff Falk, Jim Pham

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, duplication, disclosure, modification, and adaptation shall be subject to the restrictions and license terms set forth in the applicable Government contract, and, to the extent applicable by the terms of the Government contract, the additional rights set forth in FAR 52.227-19, Commercial Computer Software License (December 2007). Oracle America, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information on content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.

Contents

Preface	xi
Audience	xi
Documentation Accessibility	xi
Related Documents	xi
Conventions	xii
What's New in This Guide for Release 11.1.2.2.0	xiii
1 Introduction to Building Java EE Web Applications with Oracle ADF	
1.1 Introduction to Oracle ADF	1-1
1.2 Developing with Oracle ADF	1-2
1.2.1 Core Development Steps	1-3
1.2.2 Supplemental Development Tasks	1-4
1.3 Introduction to the Summit Demo Application	1-5
1.3.1 How to Download the Application Resources	1-5
1.3.2 How to Install and Run the Summit Demo Application	1-6
1.3.3 Taking a Look at the Summit Demo Application Code	1-6
2 Using ADF Model Data Binding in a Java EE Web Application	
2.1 Introduction to ADF Model Data Binding	2-1
2.2 Exposing Business Services with ADF Data Controls	2-4
2.2.1 How to Create ADF Data Controls	2-4
2.2.2 What Happens in Your Project When You Create a Data Control	2-5
2.2.2.1 DataControls.dcx Overview Editor	2-6
2.2.2.2 Data Controls Panel	2-6
2.3 Using the Data Controls Panel	2-9
2.3.1 How to Use the Data Controls Panel	2-11
2.3.2 What Happens When You Use the Data Controls Panel to Create UI Components	2-12
2.3.3 What Happens at Runtime	2-18
2.3.4 What You May Need to Know About Iterator Result Caching	2-19
2.3.5 What You May Need to Know About Configuring Validation	2-20
2.3.6 What You May Need to Know About Custom Session Bean Accessor Methods ...	2-21

3 Adding Business Logic to Data Controls

3.1	Introduction to Adding Business Logic to Data Controls.....	3-1
3.2	Configuring Data Controls	3-2
3.2.1	How to Edit a Data Control.....	3-2
3.2.2	What Happens When You Edit a Data Control	3-2
3.3	Working with Attributes.....	3-4
3.3.1	How to Control the Updatability of an Attribute	3-4
3.3.2	How to Define a Static Default Value for an Attribute	3-4
3.3.3	How to Define a Default Value Using a Groovy Expression	3-5
3.3.4	What Happens When You Create a Default Value Using a Groovy Expression	3-6
3.3.5	How to Set UI Hints on Attributes.....	3-7
3.3.6	What Happens When You Set UI Hints on Attributes.....	3-7
3.4	Adding Transient Attributes to a Bean.....	3-8
3.4.1	How to Add a Transient Attribute	3-8
3.4.2	What Happens When You Add a Transient Attribute.....	3-9
3.5	Defining Validation Rules on Attributes Declaratively	3-9
3.5.1	How to Add Validation Rules to Attributes	3-10
3.5.2	What Happens When You Add a Validation Rule	3-11
3.5.3	How to Use the Built-in Declarative Validation Rules.....	3-11
3.5.3.1	Validating Based on a Comparison	3-11
3.5.3.2	What Happens When You Validate Based on a Comparison.....	3-12
3.5.3.3	Validating Using a List of Values.....	3-12
3.5.3.4	What Happens When You Validate Using a List of Values.....	3-13
3.5.3.5	Ensuring That a Value Falls Within a Certain Range.....	3-13
3.5.3.6	What Happens When You Use a Range Validator	3-14
3.5.3.7	Validating Against a Number of Bytes or Characters.....	3-14
3.5.3.8	What Happens When You Validate Against a Number of Bytes or Characters	3-15
3.5.3.9	Validating Using a Regular Expression	3-15
3.5.3.10	What Happens When You Validate Using a Regular Expression.....	3-16
3.5.4	How to Use Groovy Expressions For Validation and Business Rules	3-17
3.5.4.1	Referencing Bean Methods in Groovy Validation Expressions.....	3-17
3.5.4.2	Validating Using a True/False Expression.....	3-17
3.5.4.3	What Happens When You Add a True/False Expression	3-18
3.5.5	How to Create Validation Error Messages	3-18
3.5.5.1	Creating Validation Error Messages.....	3-18
3.5.5.2	Localizing Validation Messages	3-20
3.5.5.3	Raising Error Message Conditionally Using Groovy.....	3-20
3.5.5.4	Embedding a Groovy Expression in an Error Message	3-21
3.5.6	How to Set the Severity Level for Validation Exceptions	3-21
3.6	Filtering Result Sets with Named Criteria	3-22
3.6.1	Use Case for Named Criteria	3-22
3.6.2	How to Create Named Criteria Declaratively	3-22
3.6.3	What Happens When You Create a Named Criteria	3-26
3.6.4	How to Use Bind Variables in Named Criteria.....	3-26
3.6.5	What Happens When You Use Bind Variables in Named Criteria.....	3-27
3.6.6	What You May Need to Know About Nested Expressions.....	3-27

3.6.7	How to Set User Interface Hints on Named Criteria.....	3-27
3.7	Testing Bean Metadata Using the Oracle ADF Model Tester.....	3-29
3.7.1	How to Run the Oracle ADF Model Tester.....	3-29
3.7.2	How to Update the Oracle ADF Model Tester to Display Project Changes	3-32
3.7.3	What Happens When You Use the Oracle ADF Model Tester	3-32
3.7.4	How to Test Business Layer Validation	3-33
3.7.5	How to Test Alternate Language Message Bundles and UI Hints.....	3-34
3.7.6	How to Test Row Creation and Default Value Generation.....	3-34
3.7.7	How to Test Named Criteria Using the Oracle ADF Model Tester	3-35
3.8	Groovy Language Support	3-36
3.8.1	How to Reference ADF Objects in Groovy Expressions	3-37
3.8.2	How to Reference ADF Methods and Attributes in Groovy Expressions.....	3-38

4 Creating a Basic Databound Page

4.1	About Creating a Basic Databound Page	4-1
4.2	Using Attributes to Create Text Fields.....	4-2
4.2.1	How to Create a Text Field	4-2
4.2.2	What Happens When You Create a Text Field	4-4
4.2.2.1	Creating and Using Iterator Bindings	4-4
4.2.2.2	Creating and Using Value Bindings	4-5
4.2.2.3	Using EL Expressions to Bind UI Components	4-5
4.3	Creating a Basic Form.....	4-6
4.3.1	How to Create a Form	4-7
4.3.2	What Happens When You Create a Form	4-8
4.4	Incorporating Range Navigation into Forms.....	4-9
4.4.1	How to Insert Navigation Controls into a Form	4-10
4.4.2	What Happens When You Create Command Buttons.....	4-11
4.4.2.1	Action Bindings for Built-in Navigation Operations	4-11
4.4.2.2	Iterator RangeSize Attribute	4-11
4.4.2.3	EL Expressions Used to Bind to Navigation Operations.....	4-12
4.5	Creating a Form Using a Method That Takes Parameters.....	4-14
4.5.1	How to Create a Form or Table Using a Method That Takes Parameters.....	4-14
4.5.2	What Happens When You Create a Form Using a Method That Takes Parameters	4-15
4.5.3	What Happens at Runtime: Setting Parameters for a Method.....	4-15
4.5.4	What You May Need to Know About Setting Parameters with Methods	4-16
4.5.5	What You May Need to Know About Using Contextual Events Instead of Parameters..	4-16
4.6	Creating a Form to Edit an Existing Record	4-17
4.6.1	How to Create Edit Forms	4-17
4.6.2	What Happens When You Use Methods to Change Data.....	4-19
4.6.2.1	Method Bindings	4-19
4.6.2.2	Using EL Expressions to Bind to Methods	4-19
4.6.3	What You May Need to Know About the Difference Between the Merge and Persist Methods	4-20
4.6.4	What You May Need to Know About Overriding Declarative Methods.....	4-20
4.7	Creating an Input Form	4-21

4.7.1	How to Create an Input Form Using a Task Flow	4-21
4.7.2	What Happens When You Create an Input Form Using a Task Flow	4-23
4.7.3	What Happens at Runtime: Invoking the Create Action Binding from the Method Activity 4-24	
4.8	Modifying the UI Components and Bindings on a Form	4-24

5 Creating ADF Databound Tables

5.1	About Adding Tables	5-1
5.2	Creating a Basic Table	5-1
5.2.1	How to Create a Basic Table.....	5-2
5.2.2	What Happens When You Create a Table	5-4
5.2.2.1	Iterator and Value Bindings for Tables	5-4
5.2.2.2	Code on the JSF Page for an ADF Faces Table	5-5
5.2.3	What You May Need to Know About Setting the Current Row in a Table	5-8
5.3	Creating an Editable Table	5-10
5.3.1	How to Create an Editable Table.....	5-11
5.3.2	What Happens When You Create an Editable Table	5-14
5.4	Creating an Input Table	5-14
5.4.1	How to Create an Input Table.....	5-14
5.4.2	What Happens When You Create an Input Table	5-16
5.4.3	What Happens at Runtime: How Create and Partial Page Refresh Work	5-18
5.4.4	What You May Need to Know About Creating a Row and Sorting Columns.....	5-18
5.5	Modifying the Attributes Displayed in the Table	5-19

6 Displaying Master-Detail Data

6.1	Introduction to Displaying Master-Detail Data.....	6-1
6.2	Identifying Master-Detail Objects on the Data Controls Panel.....	6-2
6.3	Using Tables and Forms to Display Master-Detail Objects	6-3
6.3.1	How to Display Master-Detail Objects in Tables and Forms	6-4
6.3.2	What Happens When You Create Master-Detail Tables and Forms	6-5
6.3.2.1	Code Generated in the JSF Page	6-5
6.3.2.2	Binding Objects Defined in the Page Definition File.....	6-5
6.3.3	What Happens at Runtime: ADF Iterator for Master-Detail Tables and Forms.....	6-7
6.3.4	What You May Need to Know About Displaying Master-Detail Widgets on Separate Pages 6-7	
6.4	Using Trees to Display Master-Detail Objects	6-8
6.4.1	How to Display Master-Detail Objects in Trees	6-8
6.4.2	What Happens When You Create an ADF Databound Tree.....	6-10
6.4.2.1	Code Generated in the JSF Page	6-10
6.4.2.2	Binding Objects Defined in the Page Definition File.....	6-11
6.4.3	What Happens at Runtime: Displaying an ADF Databound Tree	6-12
6.5	Using Tree Tables to Display Master-Detail Objects	6-12
6.5.1	How to Display Master-Detail Objects in Tree Tables	6-13
6.5.2	What Happens When You Create a Databound Tree Table.....	6-13
6.5.2.1	Code Generated in the JSF Page	6-13
6.5.2.2	Binding Objects Defined in the Page Definition File.....	6-14
6.5.3	What Happens at Runtime: Events	6-14

6.5.4	Using the TargetIterator Property	6-15
6.6	Using Selection Events with Trees and Tables	6-16
6.6.1	How to Use Selection Events with Trees and Tables	6-16
6.6.2	What Happens at Runtime: RowKeySet Objects and SelectionEvent Events.....	6-17

7 Creating Databound Selection Lists

7.1	Introduction to Selection Lists	7-1
7.2	Creating a Single Selection List.....	7-1
7.2.1	How to Create a Single Selection List Containing Fixed Values	7-3
7.2.2	How to Create a Single Selection List Containing Dynamically Generated Values..	7-3
7.2.3	What Happens When You Create a Fixed Selection List.....	7-4
7.2.4	What Happens When You Create a Dynamic Selection List.....	7-5
7.3	Creating a List with Navigation List Binding.....	7-6

8 Creating Databound Search Forms

8.1	Introduction to Creating Search Forms	8-1
8.1.1	Query Search Forms	8-2
8.1.2	Quick Query Search Forms	8-7
8.1.3	Filtered Table and Query-by-Example Searches.....	8-7
8.2	Creating Query Search Forms.....	8-8
8.2.1	How to Create a Query Search Form with a Results Table or Tree Table.....	8-9
8.2.2	How to Create a Query Search Form and Add a Results Component Later	8-9
8.2.3	How to Persist Saved Searches into MDS	8-10
8.2.4	What Happens When You Create a Query Form	8-10
8.2.5	What Happens at Runtime: Search Forms.....	8-12
8.3	Setting Up Search Form Properties	8-12
8.3.1	How to Set Search Form Properties on the Query Component.....	8-12
8.4	Creating Quick Query Search Forms	8-13
8.4.1	How to Create a Quick Query Search Form with a Results Table or Tree Table	8-13
8.4.2	How to Create a Quick Query Search Form and Add a Results Component Later	8-14
8.4.3	How to Set the Quick Query Layout Format.....	8-14
8.4.4	What Happens When You Create a Quick Query Search Form.....	8-15
8.4.5	What Happens at Runtime: Quick Query	8-15
8.5	Creating Standalone Filtered Search Tables	8-15

9 Deploying an ADF Java EE Application

9.1	Introduction to Deploying ADF Java EE Web Applications	9-1
9.1.1	Developing Applications with Integrated WebLogic Server	9-3
9.1.2	Developing Applications for Standalone Application Server.....	9-3
9.2	Running a Java EE Application in Integrated WebLogic Server	9-5
9.2.1	How to Run an Application in Integrated WebLogic Server	9-6
9.2.2	How to Run an Application with Metadata in Integrated WebLogic Server	9-6
9.3	Preparing the Application	9-8
9.3.1	How to Create a Connection to the Target Application Server	9-8
9.3.2	How to Create Deployment Profiles.....	9-10
9.3.2.1	Creating a WAR Deployment Profile	9-11

9.3.2.2	Creating a MAR Deployment Profile	9-12
9.3.2.3	Creating an EJB JAR Deployment Profile	9-15
9.3.2.4	Creating an Application-Level EAR Deployment Profile	9-15
9.3.2.5	Delivering Customization Classes as a Shared Library	9-16
9.3.2.6	Viewing and Changing Deployment Profile Properties.....	9-17
9.3.2.7	Adding Customization Classes into a JAR.....	9-17
9.3.3	How to Create and Edit Deployment Descriptors.....	9-18
9.3.3.1	Creating Deployment Descriptors	9-19
9.3.3.2	Viewing or Modifying Deployment Descriptor Properties.....	9-20
9.3.3.3	Configuring the application.xml File for WebLogic Compatibility	9-20
9.3.3.4	Configuring the web.xml File for WebLogic Compatibility	9-21
9.3.3.5	Enabling the Application for Real User Experience Insight.....	9-21
9.3.4	How to Deploy Applications with ADF Security Enabled.....	9-22
9.3.4.1	Applications That Will Run Using Oracle Single Sign-On (SSO).....	9-22
9.3.4.2	Configuring Security for WebLogic Server	9-23
9.3.4.2.1	Applications with JDBC Data Source for WebLogic.....	9-24
9.3.4.3	Configuring Security for WebSphere Server	9-24
9.3.4.3.1	Applications with JDBC Data Source for WebSphere	9-25
9.3.4.3.2	Editing the web.xml File to Protect the Application Root for WebSphere	9-25
9.3.5	How to Replicate Memory Scopes in a Clustered Environment	9-25
9.3.6	How to Enable the Application for ADF MBeans.....	9-25
9.3.7	What You May Need to Know About JDBC Data Source for Oracle WebLogic Server ...	9-27
9.4	Deploying the Application	9-27
9.4.1	How to Deploy to WebLogic Server from JDeveloper.....	9-30
9.4.2	How to Create an EAR File for Deployment	9-31
9.4.3	What You May Need to Know About ADF Libraries	9-32
9.4.4	How to Deploy New Customizations Applied to ADF Library.....	9-32
9.4.4.1	Exporting Customization to a Deployed Application	9-33
9.4.4.2	Deploying Customizations to a JAR.....	9-33
9.4.5	What You May Need to Know About ADF Libraries	9-34
9.4.6	What You May Need to Know About EAR Files and Packaging.....	9-34
9.4.7	How to Deploy the Application Using Scripts and Ant	9-34
9.4.8	What You May Need to Know About JDeveloper Runtime Libraries	9-34
9.5	Postdeployment Configuration	9-34
9.5.1	How to Migrate an Application.....	9-35
9.5.2	How to Configure the Application Using ADF MBeans	9-35
9.5.3	How to Configure WebSphere for Result Sets Reuse.....	9-35
9.6	Testing the Application and Verifying Deployment	9-35

Preface

Welcome to the *Java EE Developer's Guide for Oracle Application Development Framework*.

Audience

This document is intended for enterprise developers who need to create and deploy database-centric Java EE applications using the Oracle Application Development Framework (Oracle ADF). This guide explains how to build web applications using the Enterprise JavaBeans (EJB), ADF Model, ADF Controller, and ADF Faces technologies.

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

- *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*
- *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*
- *Oracle Fusion Middleware User Guide for Oracle JDeveloper*
- *Oracle Fusion Middleware Skin Editor User's Guide for Oracle Application Development Framework*
- *Oracle Fusion Middleware Administrator's Guide for Oracle Application Development Framework*
- *Oracle Fusion Middleware Desktop Integration Developer's Guide for Oracle Application Development Framework*

- *Oracle Fusion Middleware Mobile Browser Developer's Guide for Oracle Application Development Framework*
- *Oracle Fusion Middleware Performance and Tuning Guide*
- *Oracle Fusion Middleware High Availability Guide*
- *Oracle Fusion Middleware Installation Guide for Oracle JDeveloper*
- *Oracle Fusion Middleware Installation Guide for Oracle Application Development Framework Skin Editor*
- *Oracle JDeveloper Online Help*
- *Oracle JDeveloper 11g Release Notes*, included with your JDeveloper installation, and on Oracle Technology Network
- *Oracle Fusion Middleware Java API Reference for Oracle ADF Model*
- *Oracle Fusion Middleware Java API Reference for Oracle ADF Controller*
- *Oracle Fusion Middleware Java API Reference for Oracle ADF Lifecycle*
- *Oracle Fusion Middleware Java API Reference for Oracle ADF Faces*
- *Oracle Fusion Middleware Javascript API Reference for Oracle ADF Faces*
- *Oracle Fusion Middleware Java API Reference for Oracle ADF Data Visualization Components*
- *Oracle Fusion Middleware Java API Reference for Oracle ADF Share*
- *Oracle Fusion Middleware Java API Reference for Oracle Business Component Browser*
- *Oracle Fusion Middleware Generic Domains Generic Domains Java API Reference for Oracle ADF Business Components*
- *Oracle Fusion Middleware interMedia Domains interMedia Domains Java API Reference for Oracle ADF Business Components*
- *Oracle Fusion Middleware Java API Reference for Oracle Metadata Service (MDS)*
- *Oracle Fusion Middleware Tag Reference for Oracle ADF Faces*
- *Oracle Fusion Middleware Tag Reference for Oracle ADF Faces Skin Selectors*
- *Oracle Fusion Middleware Data Visualization Tools Tag Reference for Oracle ADF Faces*
- *Oracle Fusion Middleware Data Visualization Tools Tag Reference for Oracle ADF 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.
<i>italic</i>	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 for Release 11.1.2.2.0

For Release 11.1.2.2.0, this guide has been updated. The following table lists the section that has been added.

For 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>.

Sections	Changes Made
Chapter 9 Deploying an ADF Java EE Application	
Section 9.5.3, "How to Configure WebSphere for Result Sets Reuse"	New section that describes how to configure WebSphere Application Server to enable reuse of result sets across requests.

Introduction to Building Java EE Web Applications with Oracle ADF

This chapter describes the architecture and key functionality of the Oracle Application Development Framework (Oracle ADF) when used to build a web application with session and entity beans that use EJB 3.0 annotations and the Java Persistence API (JPA), along with ADF Model, ADF Controller, and ADF Faces rich client.

This chapter includes the following sections:

- Section 1.1, "Introduction to Oracle ADF"
- Section 1.2, "Developing with Oracle ADF"
- Section 1.3, "Introduction to the Summit Demo Application"

1.1 Introduction to Oracle ADF

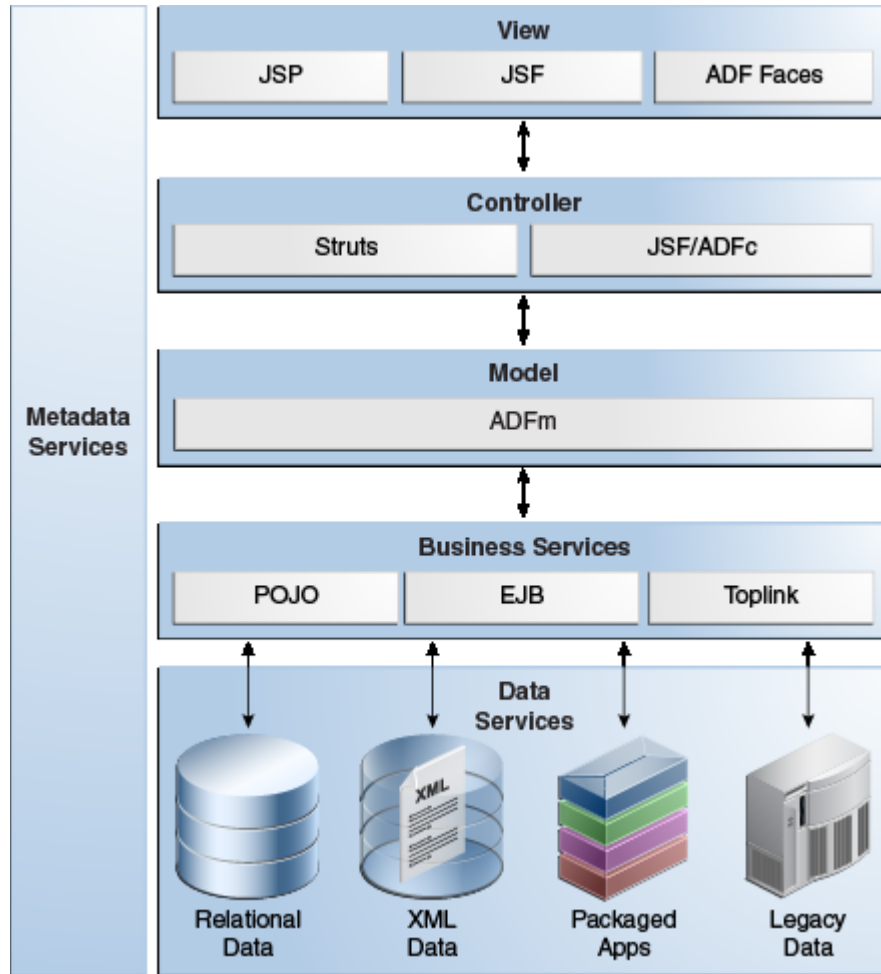
The Oracle Application Development Framework (Oracle ADF) is an end-to-end application framework that builds on Java Platform, Enterprise Edition (Java EE) standards and open-source technologies to simplify and accelerate implementing enterprise applications. If you develop enterprise solutions that search, display, create, modify, and validate data using web, wireless, desktop, or web services interfaces, Oracle ADF can simplify your job. Used in tandem, Oracle JDeveloper and Oracle ADF give you an environment that covers the full development lifecycle from design to deployment, with drag and drop data binding, visual UI design, and team development features built in.

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

- ADF Faces rich client, which offers a rich library of AJAX-enabled UI components for web applications built with JavaServer Faces (JSF). For more information about ADF Faces, refer to the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.
- ADF Controller, which integrates JSF with ADF Model. The ADF Controller extends the standard JSF controller by providing additional functionality, such as reusable task flows that pass control not only between JSF pages, but also between other activities, for instance method calls or other task flows. For more information about ADF Controller, see the "Getting Started with ADF Task Flows" section of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

Note: In addition to ADF Faces, Oracle ADF also supports using the Swing, JSP, and standard JSF view technologies. For more information about these technologies, refer to the *Oracle Fusion Middleware User Guide for Oracle JDeveloper*. Oracle ADF also provides support for using Microsoft Excel as a view layer for your application. For more information, see the *Oracle Fusion Middleware Desktop Integration Developer's Guide for Oracle Application Development Framework*.

Figure 1-1 Simple Oracle ADF Architecture



1.2 Developing with Oracle ADF

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

1.2.1 Core Development Steps

At a high level, the declarative development process for a Java EE web application usually involves the following core steps:

- Creating an application workspace: Using a wizard, JDeveloper automatically adds the libraries and configuration needed for the technologies you select, and structures your application into projects with packages and directories. For more information, see the "How to Create an Application" section of the *Oracle Fusion Middleware User Guide for Oracle JDeveloper*.
- Creating the persistence model: From your database tables, you create EJB 3.0 entity beans using wizards or dialogs. From those beans, you create the session bean as the facade that will be used by the pages in your application. For more information about using JDeveloper with EJBs, see the "Developing with EJB and JPA Components" section of the *Oracle Fusion Middleware User Guide for Oracle JDeveloper*.
- Creating data controls for your services: Once you've created your entity and session beans, you create the data controls that use metadata interfaces to abstract the implementation of your EJBs, 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 Chapter 2, "Using ADF Model Data Binding in a Java EE Web Application."
- Adding declarative metadata to your data controls: You can augment your data controls with UI control hints, validation rules, criteria for use in search forms, and other features. For more information, see Chapter 3, "Adding Business Logic to Data Controls".
- Implementing the user interface with JSF: JDeveloper's Data Controls panel contains a representation of the beans for your application. Creating a user interface is as simple as dragging an object onto a page and selecting the UI component you want to display the underlying data. For UI components that are not databound, you use the Component Palette to drag and drop components. JDeveloper creates all the page code for you. For more information, see the "Implementing the User Interface with JSF" section in the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

For information about creating specific types of web pages, see the following in this guide:

- Chapter 4, "Creating a Basic Databound Page"
 - Chapter 5, "Creating ADF Databound Tables"
 - Chapter 6, "Displaying Master-Detail Data"
 - Chapter 7, "Creating Databound Selection Lists"
 - Chapter 8, "Creating Databound Search Forms"
- Deploying the application: You use wizards and editors to create and edit deployment descriptors, JAR files, and application server connections. For more information, see Chapter 9, "Deploying an ADF Java EE Application."

Note: The process of developing a Java EE application with Oracle ADF modules is very similar to developing a Fusion web application. The main difference is that a Fusion web application uses ADF Business Components for the back-end services. When the development process and procedures are the same for both application types, this guide refers you to the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework* for that information.

Please disregard any information in the Fusion Developer's guide regarding ADF Business Components (such as entity objects and view objects). For similar information for EJB/JPA, refer to the "Developing with EJB and JPA Components" section of the *Oracle Fusion Middleware User Guide for Oracle JDeveloper*.

1.2.2 Supplemental Development Tasks

In addition to the above core steps for developing Java EE applications with Oracle ADF, JDeveloper provides tools for the tasks in the list below. These tasks are not covered in detail in this guide.

- Modeling the database objects: You can create an offline replica of any database, and use JDeveloper editors and diagrammers to edit definitions and update schemas. For more information, see the "Modeling with Database Object Definitions" section of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.
- Creating use cases: Using the UML modeler, you can create use cases for your application. For more information, see the "Creating Use Cases" section of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.
- Designing application control and navigation: You use diagrammers to visually determine the flow of application control and navigation. JDeveloper creates the underlying XML for you. For more information, see the "Designing Application Control and Navigation using ADF Task Flows" section of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.
- Identifying shared resources: You use a resource library that allows you to view and use imported libraries by simply dragging and dropping them into your application. For more information, see the "Identifying Shared Resources" section of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.
- Securing the application: You use editors to create roles and populate these with test users. You then use a flat file editor to define security policies for these roles and assign them to specific resources in your application. For more information, see the "Enabling ADF Security in a Fusion Web Application" chapter in the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.
- Developing pages and applications to allow customization: Using the customization features provided by the Oracle Metadata Services (MDS), you can create applications that are customizable by customers yet still easily upgradable, create pages that allow end users to change the application UI at runtime, and create applications that are completely customizable at runtime. For more information, see the "Customizing Applications with MDS" and "Allowing User

Customizations at Runtime" chapters of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

- Testing and debugging: JDeveloper includes an integrated application server that allows you to fully test your application without needing to package it up and deploy it. JDeveloper also includes the ADF Declarative Debugger, a tool that allows you to set breakpoints and examine the data. For more information, see the "Testing and Debugging ADF Components" chapter of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

1.3 Introduction to the Summit Demo Application

As a companion to this guide, the Summit demo application was created to demonstrate the use of the Java EE and Oracle ADF web application technology stack to create transaction-based web applications. The demonstration application is used to illustrate points and provide code samples.

1.3.1 How to Download the Application Resources

The Summit application requires an existing Oracle database. You run the Summit application using Oracle JDeveloper 11g Release 2.

Do the following before installing the Summit application:

- Install Oracle JDeveloper. You need the Studio configuration of Oracle JDeveloper 11g Release 2 to view the application's projects and run the application using the JDeveloper integrated server. You can download Oracle JDeveloper from:

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

- Download the Summit application ZIP file (`Summit_JPA.zip`). You can download the ZIP file from:

http://www.oracle.com/webfolder/technetwork/jdeveloper/downloads/Summit_JPA/Summit_JPA.zip

- Download the Summit database schema (`Summit_Schema.zip`). You can download the ZIP file from:

http://www.oracle.com/webfolder/technetwork/jdeveloper/downloads/Summit_JPA/Summit_Schema.zip

- Install an Oracle database. The Summit application requires a database for its data.

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

<http://www.oracle.com/technetwork/index.html>

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

<http://www.oracle.com/technetwork/database/express-edition/overview/index.html>

1.3.2 How to Install and Run the Summit Demo Application

To install and run the summit demo application, follow the instructions that are posted in the following document:

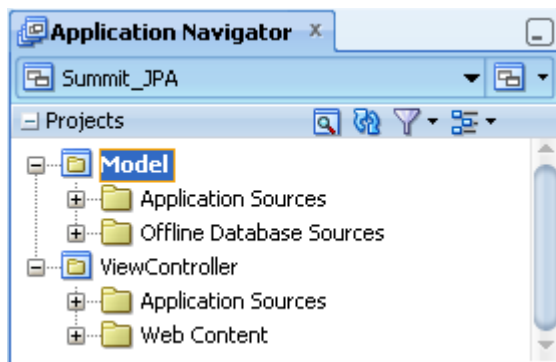
<http://www.oracle.com/technetwork/developer-tools/jdev/learnmore/summitjpainstructions-364887.pdf>

1.3.3 Taking a Look at the Summit Demo Application Code

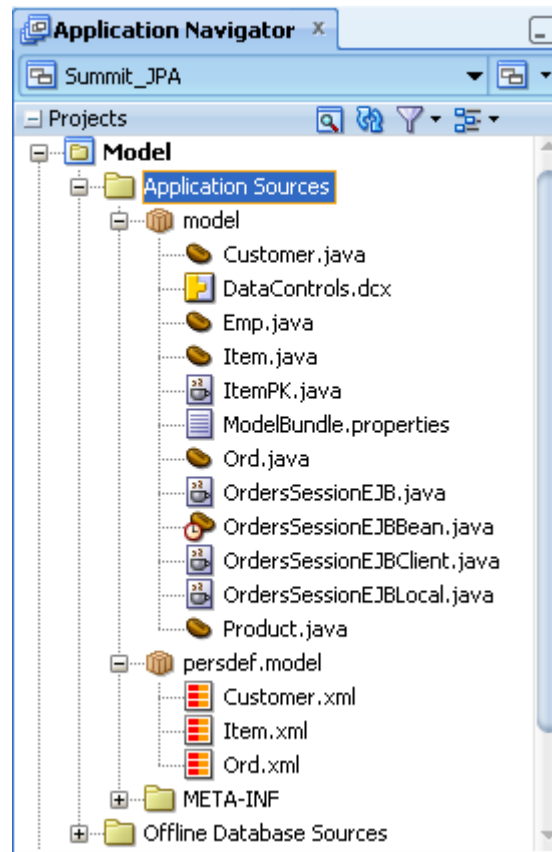
The Summit application consists of a business services project named `Model` and a web user interface project named `ViewController`. You run the Summit application by running the `ViewController` project. The `ViewController` project uses JavaServer Faces (JSF) as the view technology, and relies on the ADF Model layer to interact with the EJBs in the `Model` project.

Figure 1–2 shows the Application Navigator after you open the file for the application workspace.

Figure 1–2 The Summit Demo Application Projects in Oracle JDeveloper

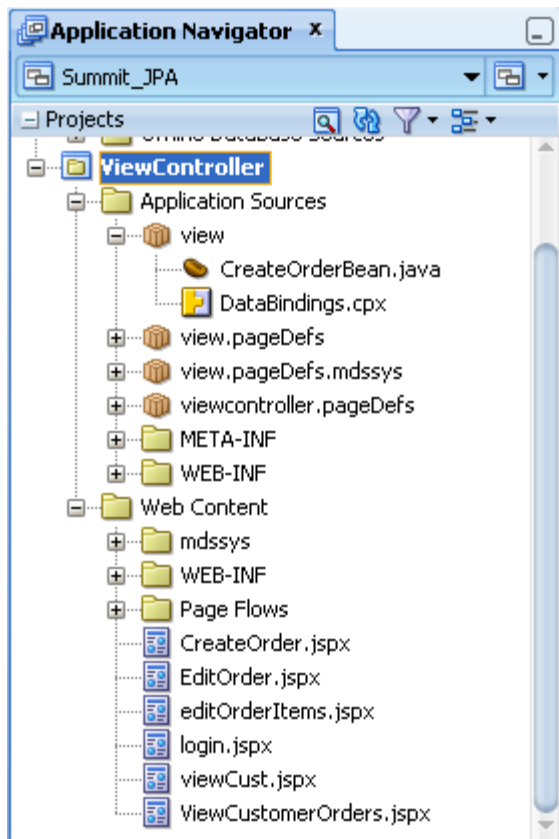


Once you have opened the demo application's projects in Oracle JDeveloper, you can then begin to review the artifacts within each project. The `Model` project contains the Java classes and metadata files that allow the data to be displayed in the web application. The `model` package contains Java classes and the `DataControls.dcx` file. The `persdef.model` package contains data control structure files that contain declarative metadata for some of the beans in the `model` package. Figure 1–3 shows the `Model` project and its associated directories.

Figure 1–3 The Model Project in JDeveloper

The `ViewController` project contains the files for the web interface, including the backing beans, deployment files, and JSPX files. The `Application Sources` node contains the code used by the web client, including a managed bean and the metadata used by Oracle ADF to display bound data. The `Web Content` node contains web files, including the JSP files, images, skin files, deployment descriptors, and libraries. Figure 1–4 shows the `ViewController` project and its associated directories.

Figure 1-4 The ViewController Project in JDeveloper



Using ADF Model Data Binding in a Java EE Web Application

This chapter describes how to create ADF model data controls for EJB session beans and how to use the Data Controls panel to create databound UI components on JSF web pages.

This chapter includes the following sections:

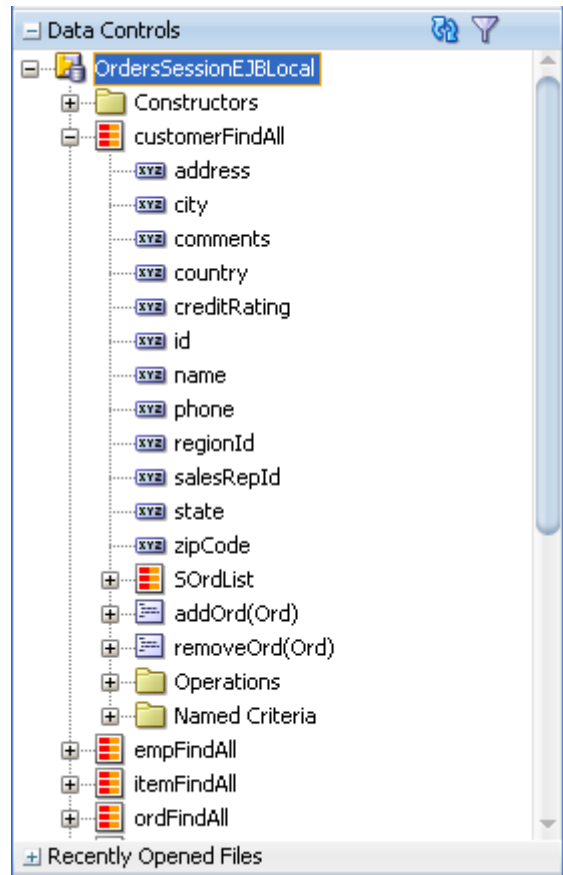
- Section 2.1, "Introduction to ADF Model Data Binding"
- Section 2.2, "Exposing Business Services with ADF Data Controls"
- Section 2.3, "Using the Data Controls Panel"

For information on configuring the business services that you will bind to UI components, see Chapter 3, "Adding Business Logic to Data Controls." For more comprehensive information about using ADF Model data binding, refer to the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

2.1 Introduction to ADF Model Data Binding

ADF Model implements two concepts that enable the decoupling of the user interface technology from the business service implementation: *data controls* and *declarative bindings*.

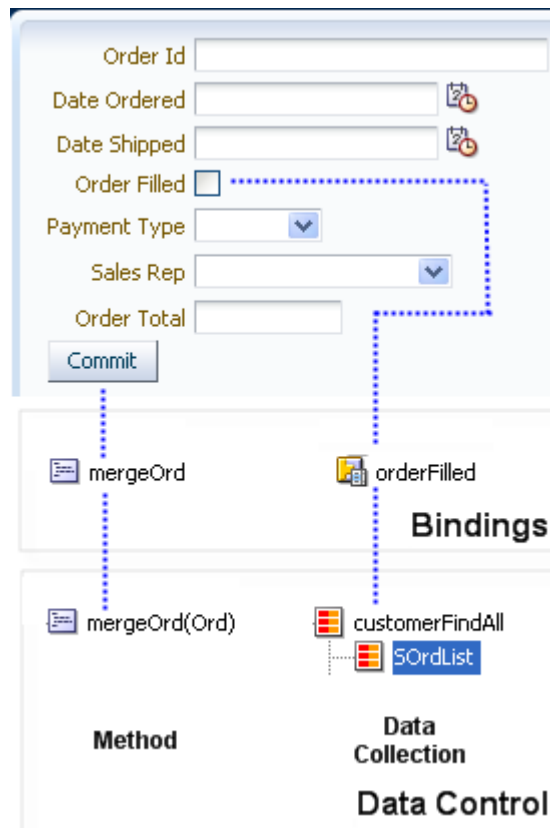
Data controls abstract the implementation technology of a business service by using standard metadata interfaces to describe the service's operations and data collections, including information about the properties, methods, and types involved. In an application that uses an EJB session facade, developers 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 2-1) to create UI components that are automatically bound to the session facade. 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.

Figure 2–1 Data Controls Panel

Declarative bindings abstract the details of accessing data from data collections in a data control and of invoking its operations. The following are the basic categories of 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 command components like hyperlinks or buttons to invoke built-in or custom operations on data collections or a data control without writing code.
- Executable bindings: Include iterator bindings, which are mainly used in the background to manage the query and the current row. 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.

Figure 2–2 shows how bindings connect UI components to data control collections and methods.

Figure 2–2 Bindings Connect UI Components to Data Controls

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}`. Example 2–1 shows the code used for binding a checkbox in a form to the `orderFilled` attribute of the `OrdersSessionEJBLocal` data control.

Example 2–1 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"/>
```

Tip: For more information about ADF EL expressions, see the "Creating ADF Data Binding EL Expressions" section of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

To use the ADF Model layer to bind data, you need to use JDeveloper 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 a collection, operation, method or attribute. You can then create databound components by dragging and dropping those subnodes onto the visual editor for a web page or other user interface component. For example, you can create databound HTML elements for JSP pages, databound UI components for JSF pages, and databound Swing UI components for ADF Swing panels. 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. At runtime, the ADF Model layer reads the metadata information from the appropriate XML files for both the data controls and the bindings, and then implements the two-way connection between your user interface and your business services.

2.2 Exposing Business 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 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 command. 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, the Summit demo application uses the `summit` database schema, which contains a number of relational database tables. The application's Model project has a number of entity beans that represent the tables in the schema used by the Summit demo application. There is a `Customer` bean, a `Product` bean, an `Order` bean, and so on. The module also contains a session bean, `OrdersSessionEJBBean`, which is used to access the beans created from tables. There is a data control for the session bean, which allows developers to declaratively create UI pages using the data and logic contained in the session bean and in the entity beans that the session bean encapsulates.

2.2.1 How to Create ADF Data Controls

You create data controls from within the Application Navigator of JDeveloper.

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 ADF Data Controls."

You will 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 the "How to Work with an EJB Business Services Layer" section of the *Oracle Fusion Middleware User Guide for Oracle JDeveloper*.

Note: You can also base a data control on a Java Service Facade class. With a Java Service Facade, you can expose business methods either from within an EJB container or without using an EJB container. A Java Service Facade template is available in the New Gallery. Also note that any class on which you base a data control must meet the JavaBeans specification. In particular, the class must have a public default constructor.

To create a data control:

1. In the Application Navigator, right-click the session bean for which you want to create a data control and choose **Create Data Control** from the context menu.
2. In the Choose EJB Interface dialog, choose **Local** or **Remote**. For web applications, typically you would choose **Local**.

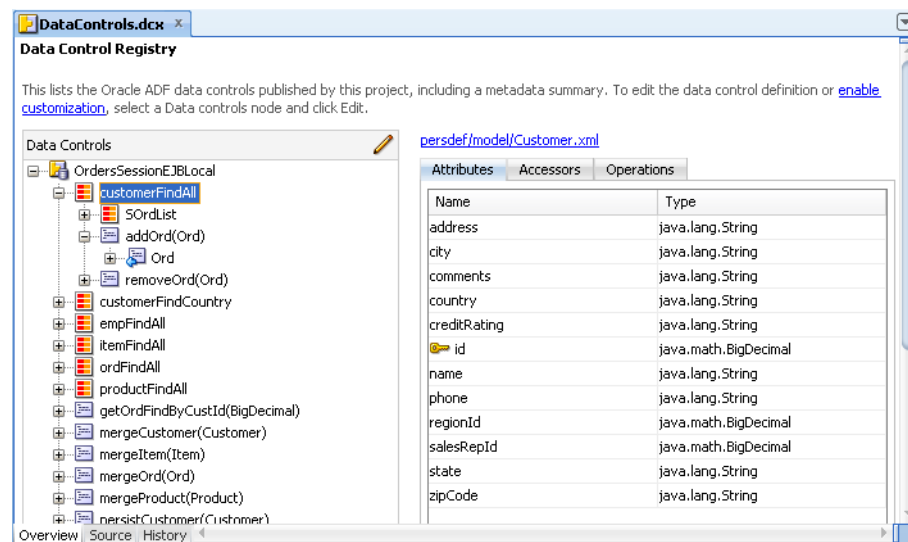
Note: If you later rename the bean on which a data control is based, you must again use the Create Data Control command in order to regenerate the data control's metadata.

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.

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

When you create a data control based on an EJB session bean, 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. Figure 2-3 shows the `DataControls.dcx` file in the overview editor.

Figure 2-3 *DataControls.dcx* File in the Overview Editor



Example 2-2 shows the code from the corresponding XML file, which you can view by clicking on the Source tab in the editor window.

Example 2-2 *DataControls.dcx* File

```
<?xml version="1.0" encoding="UTF-8" ?>
<DataControlConfigs xmlns="http://xmlns.oracle.com/adfm/configuration"
version="11.1.2.58.66" id="DataControls"
Package="model">
  <AdapterDataControl id="OrdersSessionEJBLocal"
FactoryClass="oracle.adf.model.adapter.DataControlFactoryImpl"
```

```

ImplDef="oracle.adfinternal.model.adapter.ejb.EjbDCDefinition"
SupportsTransactions="false"
    SupportsSortCollection="true" SupportsResetState="false"
SupportsRangesize="false"
    SupportsFindMode="false" SupportsUpdates="true"
Definition="model.OrdersSessionEJBLocal"
    BeanClass="model.OrdersSessionEJBLocal"
xmlns="http://xmlns.oracle.com/adfm/datacontrol">
  <CreatableTypes>
    <TypeInfo FullName="model.Product"/>
    <TypeInfo FullName="model.Item"/>
    <TypeInfo FullName="model.Ord"/>
    <TypeInfo FullName="model.Customer"/>
    <TypeInfo FullName="model.Emp"/>
  </CreatableTypes>
  <Source>
    <ejb-definition ejb-version="3.0" ejb-name="OrdersSessionEJB"
    ejb-type="Session"
    ejb-business-interface="model.OrdersSessionEJBLocal"
    ejb-interface-type="local"

    initial-context-factory="weblogic.jndi.WLInitialContextFactory"

    DataControlHandler="oracle.adf.model.adapter.bean.jpa.JPQLDataFilterHandler"
    xmlns="http://xmlns.oracle.com/adfm/adapter/ejb"/>
  </Source>
</AdapterDataControl>
</DataControlConfigs>

```

2.2.2.1 DataControls.dcx Overview Editor

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, you can view methods that operate on the entities, 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 Section 3.2, "Configuring Data Controls."

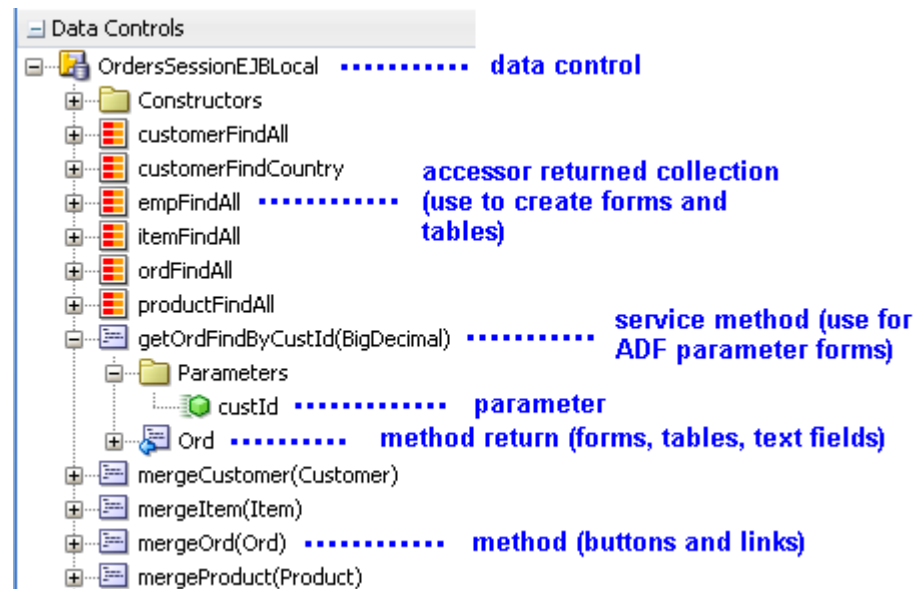
2.2.2.2 Data Controls Panel

The Data Controls panel appears in the Application Navigator once you have created a data control. 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.

Similar to the `DataControls.dcx` overview editor, the Data Controls panel reflects the master-detail hierarchies in your data model by displaying detail data collections nested under their master data collection. For example, Figure 2–4 shows the Data Controls panel with the data control for the Summit demo application. The `Customer`, `Emp`, `Item`, `Ord`, and `Product` beans are all represented by accessor returned

collections in the figure, which in turn correspond to named queries defined in the beans.

Figure 2–4 Data Controls Panel



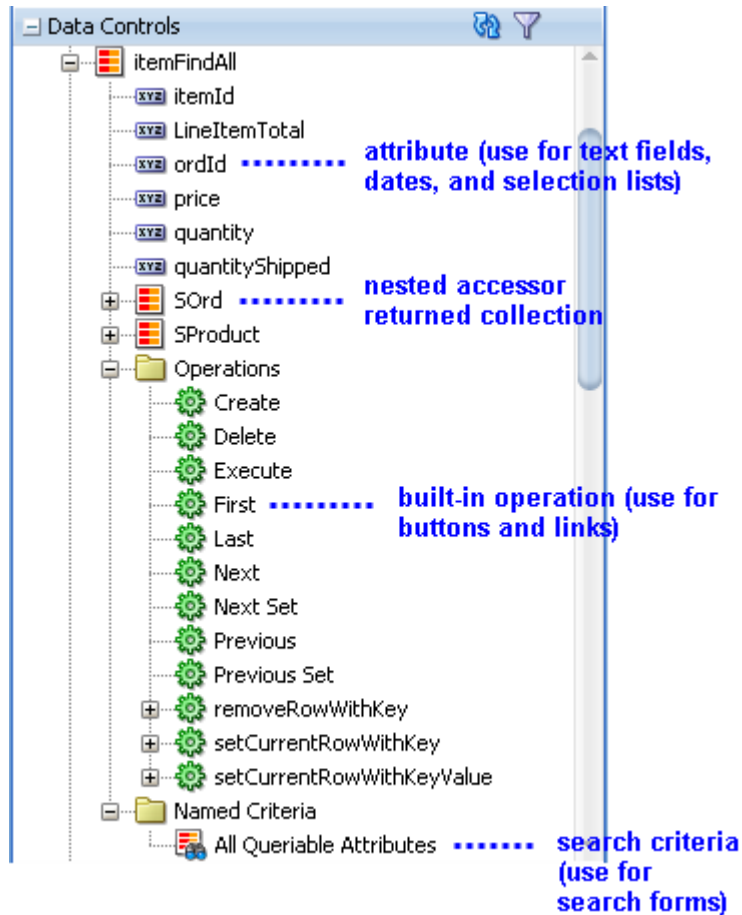
The Data Controls panel also displays each service method on the session bean as a method icon whose name matches the method name. If a method accepts arguments, those arguments appear in a Parameters node as parameters nested inside the method's node. Objects that are returned by the methods appear as well, as shown in Figure 2–4.

In addition to the nodes that are displayed in the overview editor for the `DataControls.dcx` file, the Data Controls panel displays nodes for other objects that are available for binding, such as entity bean attributes, built-in operations, and method parameters.

Tip: If the Data Controls panel is not visible, see the "How to Open the Data Controls Panel" section of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework* for instructions on opening the panel.

Each returned collection or object displays any attributes and custom methods that were defined on the associated bean. Figure 2–5 shows the attributes and methods defined on the `Item` bean that is returned by the `itemFindAll` accessor method.

Figure 2-5 Child Nodes to Returned Collections



By default, implicit named criteria are created for each attribute that is able to be queried on a bean. They appear in the Data Controls panel as the All Queriable Attributes node under the Named Criteria node, as shown in Figure 2-5. This node is used to create quick search forms, as detailed in Chapter 8, "Creating Databound Search Forms."

As shown in Figure 2-5, the Operations node under a returned collection in the Data Controls panel displays all its available built-in operations that are provided by the data control. If an operation accepts one or more parameters, then those parameters appear in a nested Parameters node. At runtime, when one of these data collection operations is invoked by name by the data binding layer, the data control delegates the call to an appropriate method on the bean interface to handle the built-in functionality. 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. Following are the built-in operations:

- `Create`: Creates a new row that becomes the current row, but does not insert it.
- `Delete`: Deletes the current row.
- `Execute`: Refreshes the data collection by executing or reexecuting the accessor method.
- `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. (Only available for stateful bean-managed session facades that contain transactional methods)
- `rollback`: Reverts all changes made within the context of the current transaction. (Only available for stateful bean-managed session facades that contain transactional methods)

Note: By default, JavaBeans components assume the `rowIndex` as the key. If you do not explicitly define a key, the index will be used.

The Data Controls panel is a direct representation of the `DataControls.dcx` file and any data control structure files that are associated with the data control. By editing the files, you can change the elements displayed in the panel. Whenever changes are made to the services on which a data control is based, the data control incorporates those changes.

2.3 Using 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 a data control 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 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 Control Icons and Object Hierarchy









Icon	Name	Description	Used to Create...
	Data Control	Represents a data control. You cannot use the data control itself to create UI components, but you can use any of the child objects listed under it. Depending on how your business services were defined, there may be more than one data control.	Serves as a container for the other objects, and is not used to create anything.
	Accessor Returned Collection	<p>Represents an object returned by a bean-style accessor method on the business service. For example, if when you created a session bean, you chose to also create accessor methods for each of the Java entities under the session bean, then an accessor returned collection is displayed for each of those entities.</p> <p>If an entity contains a relationship to another entity (for example, a foreign key), then a child accessor returned collection is shown for that entity. In Oracle ADF, the relationship between parent and child entities is called a <i>master-detail</i> relationship.</p> <p>The children under a collection may be attributes of the elements that make up the collection, operations on the entire collection, or operations on the row for each element in the collection.</p>	<p>For collections: forms, tables, trees, range navigation components, and master-detail widgets.</p> <p>For single objects: forms, master-detail widgets, and selection lists.</p> <p>For more information about creating forms, and navigation components, see Chapter 4, "Creating a Basic Databound Page."</p> <p>For more information about creating tables, see Chapter 5, "Creating ADF Databound Tables."</p> <p>For information about creating trees and other master-detail UI components, see Chapter 6, "Displaying Master-Detail Data."</p> <p>For information about creating lists, see Chapter 7, "Creating Databound Selection Lists."</p>
	Attribute	<p>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.</p> <p>This icon is used in the Data Controls panel, but not in the overview editor for the <code>DataControls.dcx</code> file.</p>	<p>Label, text field, date and selection list components.</p> <p>For information about creating text fields, see Section 4.2, "Using Attributes to Create Text Fields."</p>
	Method	Represents a method that may accept parameters, perform some business logic, and optionally return a single value, a structure, or a collection of a single value and a structure.	<p>Command components.</p> <p>For methods that accept parameters: command components and parameterized forms.</p> <p>For information about creating command components from methods, see Section 4.6, "Creating a Form to Edit an Existing Record."</p> <p>For information about creating parameterized forms, see Section 4.5, "Creating a Form Using a Method That Takes Parameters."</p>

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

Icon	Name	Description	Used to Create...
	Method Return	<p>Represents an object that is returned by a method. The returned object can be a single value or a collection.</p> <p>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.</p>	<p>For single values: text fields and selection lists.</p> <p>For collections: forms, tables, trees, and range navigation components.</p> <p>When a single-value method return is dropped, code to invoke the method is not automatically generated. You have to also create an invoke action as an executable, or drop the corresponding method as a button to invoke the method.</p>
	Operation	<p>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. The operations that are children of a particular collection or method return operate on those objects only.</p> <p>If an operation requires one or more parameters, they are listed in a Parameters node under the operation.</p> <p>This icon is used in the Data Controls panel, but not in the overview editor for the <code>DataControls.dcx</code> file.</p>	<p>Command components such as buttons or links.</p> <p>For information about creating command components from operations, see Section 4.4, "Incorporating Range Navigation into Forms."</p>
	Parameter	<p>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.</p> <p>This icon is used in the Data Controls panel, but not in the overview editor for the <code>DataControls.dcx</code> file.</p>	Label, text, and selection list components.
	Named criteria	<p>Represents a query from which you can create a user search form.</p> <p>An All Queriable Attributes criteria is generated automatically for each accessor collection. This criteria can be used to create a search form where it is possible for the user to query based on any queriable attribute in the collection.</p> <p>You can create custom view criteria and add them to the Data Controls panel. See Section 3.6, "Filtering Result Sets with Named Criteria".</p> <p>This icon is used in the Data Controls panel, but not in the overview editor for the <code>DataControls.dcx</code> file.</p>	<p>Search forms.</p> <p>For information on creating search forms, see Chapter 8, "Creating Databound Search Forms."</p>

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 drop.

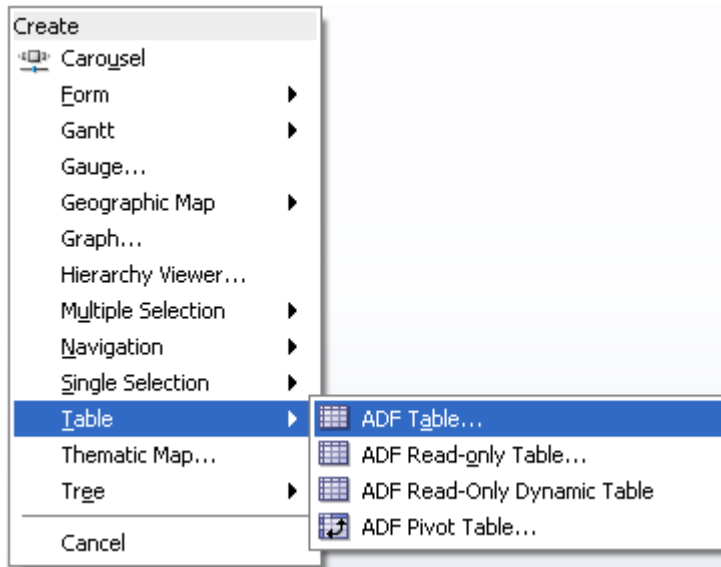
To use the Data Controls panel to create UI components:

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.
2. From the ensuing context menu, select 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.

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

Figure 2–6 Data Controls Panel Context Menu



Depending on the component you select from the context menu, JDeveloper may display a dialog that enables you to define how you want the component to look.

The resulting UI component appears in the JDeveloper visual editor, as shown in Figure 2–7.

Figure 2–7 Databound UI Component: ADF Table

# {...id.label}	# {...dateOrdered.label}	# {...paymentType.label}	# {...salesRepId.label}	# {...total.label}	# {...orderFilled.label}
# {...id}	# {...dateOrdered}	# {...paymentType}	# {...salesRepId}	# {...total}	# {...orderFilled}
# {...id}	# {...dateOrdered}	# {...paymentType}	# {...salesRepId}	# {...total}	# {...orderFilled}
# {...id}	# {...dateOrdered}	# {...paymentType}	# {...salesRepId}	# {...total}	# {...orderFilled}

Tip: Instead of creating automatically bound UI components using the Data Controls panel, you can create your UI first and then bind the components to the ADF Model layer. For more information, see the "Using Simple UI First Development" section of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

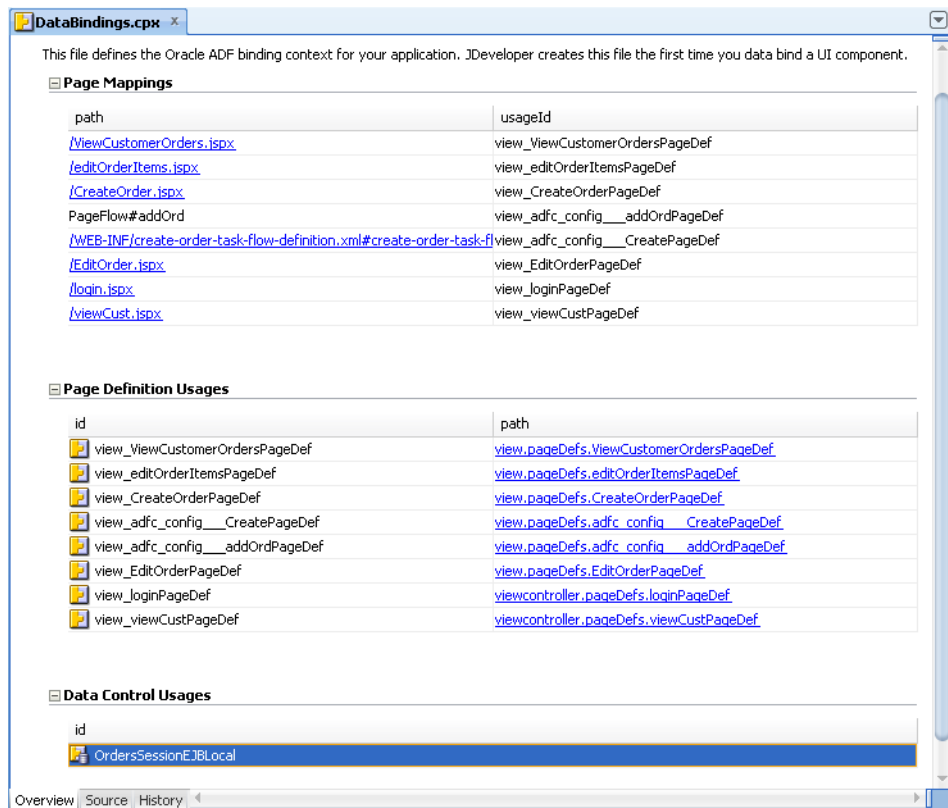
2.3.2 What Happens When You Use the Data Controls Panel to Create UI Components

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.

DataBindings.cpx files define the *binding context* (a container object that holds a list of available data controls and data binding objects) for the application. Each DataBindings.cpx file maps individual pages to the binding definitions in the page definition file and registers the data controls used by those pages. Figure 2–8 shows a DataBindings.cpx file in the overview editor of JDeveloper.

Figure 2–8 DataBindings.cpx File in the Overview Editor



Example 2–3 shows the code from the corresponding XML file.

Example 2–3 DataBindings.cpx File

```
<?xml version="1.0" encoding="UTF-8" ?>
<Application xmlns="http://xmlns.oracle.com/adfm/application"
  version="11.1.2.59.53" id="DataBindings"
  SeparateXMLFiles="false" Package="view" ClientType="Generic">
  <pageMap>
    <page path="/ViewCustomerOrders.jspx"
      usageId="view_ViewCustomerOrdersPageDef" />
    <page path="/editOrderItems.jspx" usageId="view_editOrderItemsPageDef" />
    <page path="/CreateOrder.jspx" usageId="view_CreateOrderPageDef" />
    <page path="PageFlow#addOrd" usageId="view_adfc_config__addOrdPageDef" />
    <page path="/WEB-INF/create-order-task-flow-definition.xml
      #create-order-task-flow-definition@Create"
      usageId="view_adfc_config__CreatePageDef" />
    <page path="/EditOrder.jspx" usageId="view_EditOrderPageDef" />
    <page path="/login.jspx" usageId="view_loginPageDef" />
    <page path="/viewCust.jspx" usageId="view_viewCustPageDef" />
  </pageMap>
  <pageDefinitionUsages>
    <page id="view_ViewCustomerOrdersPageDef"
```

```

        path="view.pageDefs.ViewCustomerOrdersPageDef" />
    <page id="view_editOrderItemsPageDef"
        path="view.pageDefs.editOrderItemsPageDef" />
    <page id="view_CreateOrderPageDef" path="view.pageDefs.CreateOrderPageDef" />
    <page id="view_adfc_config__CreatePageDef"
        path="view.pageDefs.adfc_config__CreatePageDef" />
    <page id="view_adfc_config__addOrdPageDef"
        path="view.pageDefs.adfc_config__addOrdPageDef" />
    <page id="view_EditOrderPageDef" path="view.pageDefs.EditOrderPageDef" />
    <page id="view_loginPageDef" path="viewcontroller.pageDefs.loginPageDef" />
    <page id="view_viewCustPageDef"
        path="viewcontroller.pageDefs.viewCustPageDef" />
</pageDefinitionUsages>
<dataControlUsages>
    <dc id="OrdersSessionEJBLocal" path="model.OrdersSessionEJBLocal" />
</dataControlUsages>
</Application>

```

For more information about the `.cpx` file, see the "Working with the DataBindings.cpx File" section of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

- Creates the `adfm.xml` file in the `META-INF` directory. This file creates a registry for the `DataBindings.cpx` file, and is used by the applications metadata layer to allow customization and personalization of the application. Example 2-4 shows an example of an `adfm.xml` file.

Example 2-4 `adfm.xml` File

```

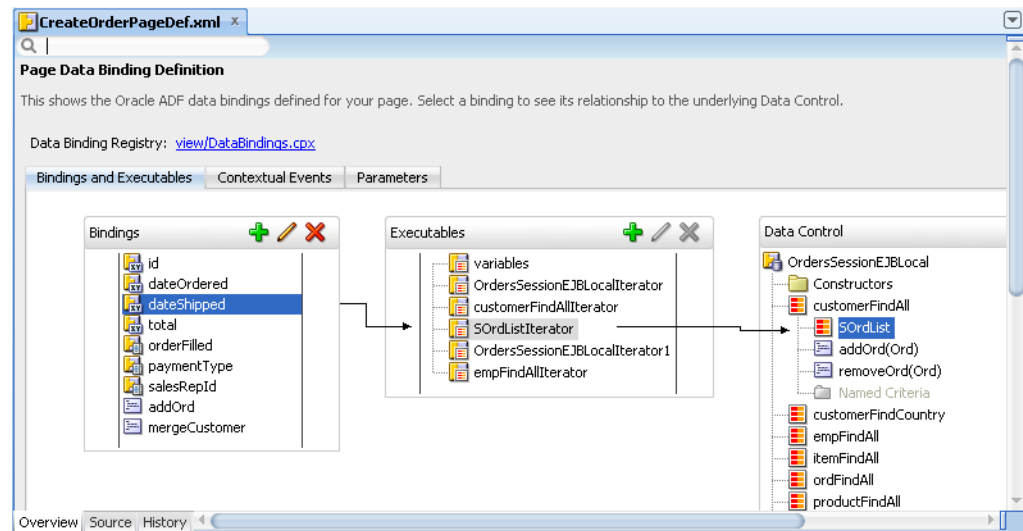
<?xml version="1.0" encoding="UTF-8" ?>
<MetadataDirectory xmlns="http://xmlns.oracle.com/adfm/metainf"
    version="11.1.1.0.0">
    <DataBindingRegistry path="view/DataBindings.cpx" />
</MetadataDirectory>

```

- For web applications, 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 filter, see the "Configuring the ADF Binding Filter" section of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.
- Adds the following libraries to the project:
 - ADF Model Runtime
 - ADF Model Generic 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`.

The page definition file (`pageNamePageDef.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. Figure 2-9 shows a page definition file in the overview editor of JDeveloper.

Figure 2–9 Page Definition File

- Configures the page definition file, which includes adding definitions of the binding objects referenced by the page. Example 2–5 shows the corresponding XML for a page definition.

Example 2–5 Page Definition File

```
<?xml version="1.0" encoding="UTF-8" ?>
<pageDefinition xmlns="http://xmlns.oracle.com/adfm/uimodel"
version="11.1.2.59.53" id="CreateOrderPageDef"
Package="view.pageDefs">
  <parameters/>
  <executables>
    <variableIterator id="variables"/>
    <iterator Binds="root" RangeSize="25" DataControl="OrdersSessionEJBLocal"
id="OrdersSessionEJBLocalIterator"/>
    <accessorIterator MasterBinding="OrdersSessionEJBLocalIterator"
Binds="customerFindAll" RangeSize="25"
DataControl="OrdersSessionEJBLocal"
BeanClass="model.Customer" id="customerFindAllIterator"/>
    <accessorIterator MasterBinding="customerFindAllIterator"
Binds="SOrdList" RangeSize="25"
DataControl="OrdersSessionEJBLocal"
BeanClass="model.Ord" id="SOrdListIterator"/>
    <iterator Binds="root" RangeSize="25" DataControl="OrdersSessionEJBLocal"
id="OrdersSessionEJBLocalIterator1"/>
    <accessorIterator MasterBinding="OrdersSessionEJBLocalIterator1"
Binds="empFindAll" RangeSize="-1"
DataControl="OrdersSessionEJBLocal"
BeanClass="model.Emp" id="empFindAllIterator"/>
  </executables>
  <bindings>
    <attributeValues IterBinding="SOrdListIterator" id="id">
      <AttrNames>
        <Item Value="id"/>
      </AttrNames>
    </attributeValues>
    <attributeValues IterBinding="SOrdListIterator" id="dateOrdered">
      <AttrNames>
        <Item Value="dateOrdered"/>
      </AttrNames>
    </attributeValues>
  </bindings>
</pageDefinition>
```

```

        </AttrNames>
    </attributeValues>
    <attributeValues IterBinding="SOrdListIterator" id="dateShipped">
        <AttrNames>
            <Item Value="dateShipped" />
        </AttrNames>
    </attributeValues>
    <attributeValues IterBinding="SOrdListIterator" id="total">
        <AttrNames>
            <Item Value="total" />
        </AttrNames>
    </attributeValues>
    <button IterBinding="SOrdListIterator" id="orderFilled"
        DTSupportsMRU="false" StaticList="true">
        <AttrNames>
            <Item Value="orderFilled" />
        </AttrNames>
        <ValueList>
            <Item Value="Y" />
            <Item Value="N" />
        </ValueList>
    </button>
    <list IterBinding="SOrdListIterator" id="paymentType"
        DTSupportsMRU="true" StaticList="true">
        <AttrNames>
            <Item Value="paymentType" />
        </AttrNames>
        <ValueList>
            <Item Value="CASH" />
            <Item Value="CREDIT" />
        </ValueList>
    </list>
    <list IterBinding="SOrdListIterator" id="salesRepId"
        DTSupportsMRU="true" StaticList="false"
        ListIter="empFindAllIterator">
        <AttrNames>
            <Item Value="salesRepId" />
        </AttrNames>
        <ListAttrNames>
            <Item Value="id" />
        </ListAttrNames>
        <ListDisplayAttrNames>
            <Item Value="firstName" />
            <Item Value="lastName" />
        </ListDisplayAttrNames>
    </list>

    <methodAction IterBinding="customerFindAllIterator" id="addOrd"
        RequiresUpdateModel="true"
        Action="invokeMethod" MethodName="addOrd"
        IsViewObjectMethod="false"
        DataControl="OrdersSessionEJBLocal"
        InstanceName="bindings.customerFindAllIterator.
            currentRow.dataProvider"
        IsLocalObjectReference="true"
        ReturnName="data.OrdersSessionEJBLocal.methodResults.
            addOrd_addOrd_addOrd_result">
        <NamedData NDName="order"
            NDValue="#{bindings.SOrdListIterator.currentRow.dataProvider}"
            NDType="model.Ord" />
    </methodAction>

```

```

</methodAction>
<methodAction id="mergeCustomer" RequiresUpdateModel="true"
    Action="invokeMethod" MethodName="mergeCustomer"
    IsViewObjectMethod="false"
    DataControl="OrdersSessionEJBLocal"
    InstanceName="data.OrdersSessionEJBLocal.dataProvider"
    ReturnName="data.OrdersSessionEJBLocal.methodResults.
        mergeCustomer_OrdersSessionEJBLocal_
        dataProvider_mergeCustomer_result">
    <NamedData NDName="customer"
        NDValue="#{bindings.customerFindAllIterator.currentRow.
            dataProvider}"
        NDType="model.Customer"/>
</methodAction>
</bindings>
</pageDefinition>

```

For more information about the page definition file, see the "Working with Page Definition Files" section of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

- Adds prebuilt components to the view page.

These prebuilt components include ADF data bindings that reference the binding objects in the page definition file. Example 2–6 shows JSF page code for components that have been bound using ADF Model data binding.

Example 2–6 JSF Page Code Fragment with ADF Model Data Binding

```

.
.
.
<af:table value="#{bindings.Ord.collectionModel}" var="row"
    rows="#{bindings.Ord.rangeSize}"
    emptyText="#{bindings.Ord.viewable ? 'No data to display.' :
        'Access Denied.'}"
    fetchSize="#{bindings.Ord.rangeSize}" rowBandingInterval="0"
    selectedRowKeys="#{bindings.Ord.collectionModel.selectedRow}"
    selectionListener="#{bindings.Ord.collectionModel.makeCurrent}"
    rowSelection="single" id="t1">
    <af:column sortProperty="#{bindings.Ord.hints.dateOrdered.name}"
        sortable="false"
        headerText="#{bindings.Ord.hints.dateOrdered.label}" id="c1">
    <af:outputText value="#{row.dateOrdered}" id="ot1">
    <af:convertDateTime
        pattern="#{bindings.Ord.hints.dateOrdered.format}"/>
    </af:outputText>
    </af:column>
    <af:column sortProperty="#{bindings.Ord.hints.dateShipped.name}"
        sortable="false"
        headerText="#{bindings.Ord.hints.dateShipped.label}" id="c2">
    <af:outputText value="#{row.dateShipped}" id="ot2">
    <af:convertDateTime
        pattern="#{bindings.Ord.hints.dateShipped.format}"/>
    </af:outputText>
    </af:column>
    <af:column sortProperty="#{bindings.Ord.hints.id.name}" sortable="false"
        headerText="#{bindings.Ord.hints.id.label}" id="c3">
    <af:outputText value="#{row.id}" id="ot3"/>
    </af:column>

```

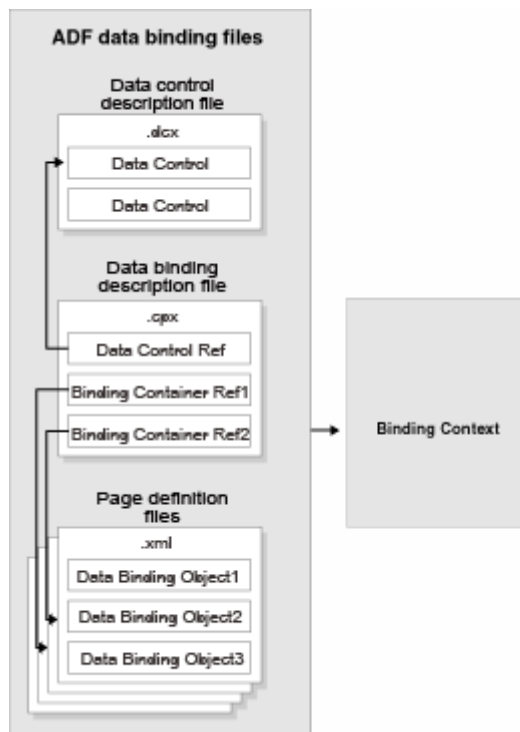
- For applications that use ADF Faces, adds all the files, and configuration elements required by ADF Faces components. For more information, see the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

2.3.3 What Happens at Runtime

When a page contains ADF bindings, at runtime the interaction with the business services initiated from the client or controller is managed by the application through the binding context. The binding context is a runtime map (named `data` and accessible through the EL expression `# {data}`) of all data controls and page definitions within the application.

The ADF lifecycle creates the ADF binding context from the `DataControls.dcx`, `DataBindings.cpx`, and page definition files, as shown in Figure 2–10. The `DataControls.dcx` file defines the data controls available to the application at design time, while the `DataBindings.cpx` files define what data controls are available to the application at runtime. A `DataBindings.cpx` file lists all the data controls that are being used by pages in the application and maps the binding containers, which contain the binding objects defined in the page definition files, to web page URLs, or in the case of a Java Swing application, the Java class. The page definition files define the binding objects used by the application pages. There is one page definition file for each page.

Figure 2–10 ADF Binding File Runtime Usage



The binding context does not contain real live instances of these objects. Instead, the map first contains references that become data control or binding container objects on demand. When the object (such as the page definition) is released from the application (such as when a task flow ends or when the binding container or data control is

released at the end of the request), data controls and binding containers turn back into reference objects. For more information, see the "Understanding the Fusion Page Lifecycle" chapter of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

2.3.4 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 the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*). 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. If your page does not have a button whose action attribute can be bound to a method, then you can use an `invokeAction` executable that will be invoked whenever 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 navigation will no longer work. You must use a button to reexecute the iterator. For more information about the navigation operations, see Section 4.4, "Incorporating Range Navigation into Forms."

Performance Tip: If you set an iterator to not cache its result set, and that result set is a collection that may return a large number of rows, performance will be negatively affected. For large result sets, you should use an `invokeAction`.

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 Property Inspector, expand the **Advanced** section and set **CacheResults** to **false**.

To use a button to reexecute the iterator:

1. From the ADF Faces page of the Component Palette, drag and drop a **Button** onto the page.

2. In the Structure window, right click the button and in the context menu, choose **Bind to ADF Control**.
3. In the Bind to ADF Control dialog, expand the accessor associated with the iterator to reexecute, expand that accessor's Operations node, and select **Execute**.

To use an invokeAction to reexecute the iterator:

1. Open the page definition file, and in the Structure window, right-click **executables** and choose **Insert inside executables > invokeAction**.
2. In the Insert invokeAction dialog, set **id** to a unique name. Use the **Binds** dropdown list to select the iterator to be reexecuted.
3. With the newly created **invokeAction** still selected, in the Property Inspector, set **Refresh** to **prepareModel**.

This setting will cause the accessor method to be invoked during the Prepare Model phase. This refreshes the binding container.

4. Set **RefreshCondition** to an EL expression that will cause the refresh to happen only if any value has actually changed. If this condition is not set, the `invokeAction` will be called twice.

For more information about the page lifecycle phases and using the `refresh` and `refreshCondition` attributes, see "About the JSF and ADF Page Lifecycles" section of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

2.3.5 What You May Need to Know About Configuring Validation

In addition to being able to set validation on attributes at the data control level, you can also set validation on the attribute bindings in a page definition file. When a user edits or enters data in a field for an attribute for which validation has been defined, and submits the form, the bound data is validated against the configured rules and conditions. If validation fails, the application displays an error message.

In most cases, setting validation at the data control level is preferable to setting validation at the page level. Any validation rules that you set on a data control are applied in all of the UI components that are created from the data control.

For information and procedures on setting validation on data controls, see Section 3.5, "Defining Validation Rules on Attributes Declaratively".

For information and procedures on setting validation on individual UI components, see the "Using Validation in the ADF Model Layer" chapter of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

When you set validation, you can define the error message that will be displayed. By default, these messages are displayed in a client dialog. You can configure these messages to display inline instead. For more information, see the "Displaying Hints and Error Messages for Validation and Conversion" section of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

You can also change the way messages are handled by creating your own error handling class. For more information, see the "Customizing Error Handling" section of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

2.3.6 What You May Need to Know About Custom Session Bean Accessor Methods

By default, when you drag an accessor returned collection from a session bean data control onto a page, the result is a paginated collection. In this case, the built-in session bean's `queryByRange()` method will be referenced from the bindings (instead of the corresponding `getBeanFindAll()` method in the session bean). If you have custom code for a `getBeanFindAll()`, `getBeanFindAllSize()`, or `getBeanFindAll(x,y)` method on a bean that you want to make sure is used at runtime, you need to remove the data control's `DataControlHandler` attribute. You can remove the `DataControlHandler` attribute in the source editor for the `DataControls.dcx` file. However, when you remove that attribute, the built-in support for named queries and pagination is also disabled.

Adding Business Logic to Data Controls

This chapter describes how to configure your data controls with custom business logic, such as validation rules, UI control hints, and default values for attributes. By configuring your data controls, you can define business logic that is then used whenever UI components are created from those data controls.

This chapter includes the following sections:

- Section 3.1, "Introduction to Adding Business Logic to Data Controls"
- Section 3.2, "Configuring Data Controls"
- Section 3.3, "Working with Attributes"
- Section 3.4, "Adding Transient Attributes to a Bean"
- Section 3.5, "Defining Validation Rules on Attributes Declaratively"
- Section 3.6, "Filtering Result Sets with Named Criteria"
- Section 3.7, "Testing Bean Metadata Using the Oracle ADF Model Tester"
- Section 3.8, "Groovy Language Support"

3.1 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, you can configure data controls to 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.

3.2 Configuring Data Controls

When you create a data control for an EJB session bean, 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 corresponds to your entity beans. You first generate a data control structure file using the overview editor for the `.dcx` file.

3.2.1 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 the entity beans. 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 3.2, "Configuring Data Controls."

You will need to complete this task:

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

To edit a data control:

1. In the Application Navigator, double-click `DataControls.dcx`.
2. In the overview editor, in the Data Controls tree, select an accessor returned collection for the bean that you would like to configure and click the **Edit** icon to generate a data control structure file.
3. In the overview editor of the data control structure file, make the desired modifications.

3.2.2 What Happens When You Edit a Data Control

When you edit a data control based on an EJB session bean, JDeveloper creates a data control structure file that contains metadata for the affected bean and opens that file in the overview editor. This file stores configuration data for the data control that is specific to that bean, such as any UI hints or validators that you have specified for the bean.

A data control structure file has the same base name as the entity bean with which it corresponds. For example, if you click the **Edit** icon when you have an accessor returned 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, in the Summit application, the `Customer.java` bean is in the `model` package, and the `Customer.xml` data control definition file is 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. By default, 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 information about setting UI hints, see the "Defining Attribute UI Hints for View Objects" section of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

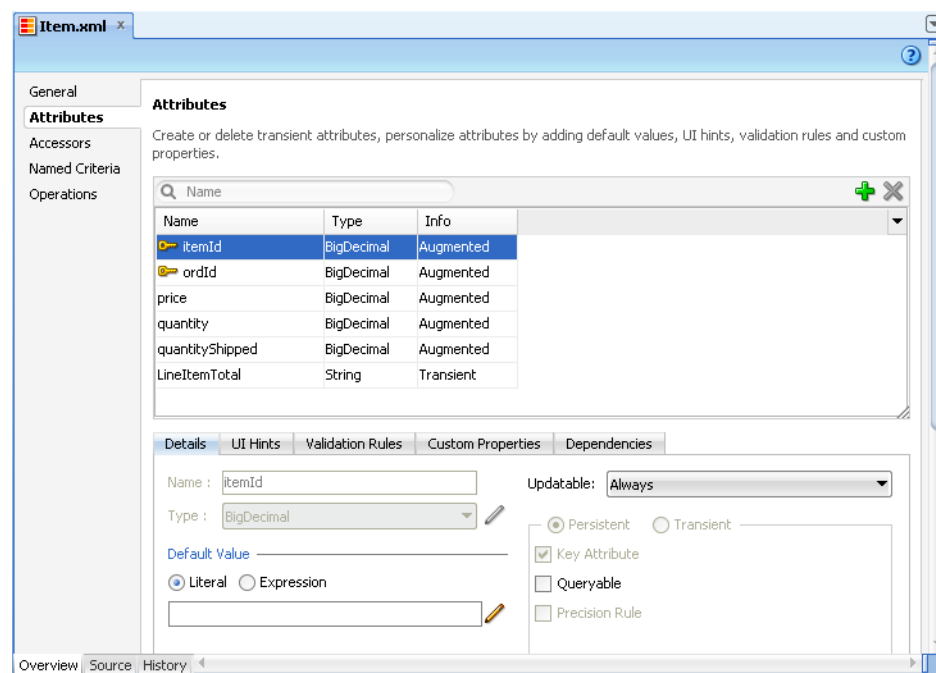
Note: View objects are ADF Business Components used to encapsulate SQL queries and to simplify working with the results. When reading this section, simply substitute "bean" for "view object."

You can also set validation for an attribute and create custom properties. For more information on validation, see Section 3.5, "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 the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

- **Accessors:** Describes data control elements that return result sets on beans. The Base Accessors section lists accessors that are based on bean properties for which an association to another entity class is defined on the property's field.
- **Named Criteria:** Enables you to create rules to filter the rows that are presented. For more information, see Section 3.6, "Filtering Result Sets with Named Criteria."
- **Operations:** Describes methods on the bean 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 3–1 shows the data control structure file for the `Item` bean in the Summit demo application.

Figure 3–1 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 bean. 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 bean. Also, when you make changes to the underlying bean, the data control inherits those changes.

3.3 Working with Attributes

When you create a data control for your EJBs, you can create a data control structure file for the bean in which you can declaratively augment the functionality of the bean'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 3.2.1, "How to Edit a Data Control."

3.3.1 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

Before you begin:

It may be helpful to have an understanding of how you set attribute properties. For more information, see Section 3.3, "Working with Attributes."

You will need to complete this task:

Create the desired data control structure files as described in Section 3.2.1, "How to Edit a Data Control."

To set the updatability of an attribute:

1. In the Application Navigator, 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, set the **Updatable** property.

3.3.2 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 3.3, "Working with Attributes."

To define a static default value for an attribute:

1. In the Application Navigator, 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.

3.3.3 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 3.8, "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 3.3, "Working with Attributes."

You will need to complete this task:

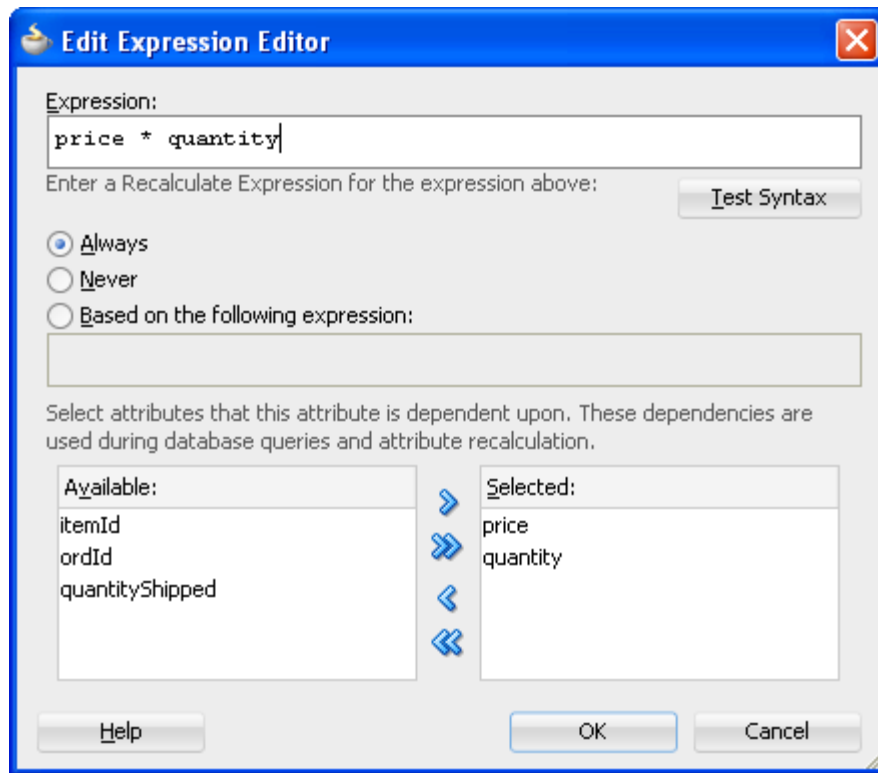
Create the desired data control structure files as described in Section 3.2.1, "How to Edit a Data Control."

To define a default value using a Groovy expression:

1. In the Application Navigator, 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, as shown in Figure 3–2.

Attributes that you reference can include any attribute that is defined for the bean.

Figure 3–2 Edit Expression Editor



6. In the same dialog, select the appropriate recalculate setting.
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.
7. Optionally, provide a condition for when to recalculate the expression.
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"));
```
8. 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.
9. Click **OK** to save the expression.

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

When you define a default value using a Groovy expression, a `<TransientExpression>` tag is added within the tag for the corresponding attribute in the data control structure file. Example 3–1 shows sample XML code for a Groovy expression that returns the current date for a default value.

Example 3–1 Default Date Value

```
<TransientExpression>
  <![CDATA[
```

```

        adf.currentDate
    ]]>
</TransientExpression>

```

3.3.5 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 bean's data control structure file, which is accessible from the Application Navigator.

Before you begin:

It may be helpful to have an understanding of how you set attribute properties. For more information, see Section 3.3, "Working with Attributes."

You will need to complete this task:

Create the desired data control structure files as described in Section 3.2.1, "How to Edit a Data Control."

To set a UI hint:

1. In the Application Navigator, 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. On the UI Hints page, set the desired UI hints.

For information about the various UI hints, see the "Defining Attribute UI Hints for View Objects" section of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

Note: View objects are ADF Business Components used to encapsulate SQL queries and to simplify working with the results. When reading this section, simply substitute "bean" for "view object."

3.3.6 What Happens When You Set UI Hints on Attributes

When you set UI hints on an attribute, JDeveloper treats those hints as properties. Tags for the properties are added to the bean'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. For example, in the Summit demo application, the resource bundle is called `ModelBundle.properties`.

Example 3-2 shows the code for the `price` attribute in the Summit demo application's `Item.xml` data control structure file, including tags for the Label and Format Type hints which have been set for the attribute.

Example 3-2 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 3–3 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 3–3 Resource Bundle Code for UI Hints

```

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

```

3.4 Adding Transient Attributes to a Bean

In addition to having attributes that map to columns in an underlying table, your bean 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.

3.4.1 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 3.4, "Adding Transient Attributes to a Bean."

You will need to complete this task:

Create the desired data control structure files as described in Section 3.2.1, "How to Edit a Data Control."

To add a transient attribute to a bean’s data control structure file:

1. In the Application Navigator, double-click the bean’s data control structure file.
2. In the overview editor, click the **Attributes** navigation tab, and then click the **New** icon.
3. In the New View Attribute dialog, enter a name for the attribute and click OK.
4. In 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 3.3.3, "How to Define a Default Value Using a Groovy Expression."

6. If the value will be calculated with an expression, set **Updatable** to **Never**.

3.4.2 What Happens When You Add a Transient Attribute

When you add a transient attribute, JDeveloper adds a `<ViewAttribute>` tag to the bean's data control structure file to reflect the new attribute. Example 3–4 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 3–4 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>
```

3.5 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 bean'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 that satisfy many of your business needs. You can also base validation on a Groovy expression, as described in Section 3.5.4, "How to Use Groovy Expressions For Validation and Business 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 beans are also available at the page 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 the *Oracle Fusion Middleware Fusion Developer's Guide for 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 the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

3.5.1 How to Add Validation Rules to Attributes

The process for adding a validation rule to a bean 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 bean'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 bean'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 3.5, "Defining Validation Rules on Attributes Declaratively."

You will need to complete this task:

Create the desired data control structure files as described in Section 3.2.1, "How to Edit a Data Control."

To add a validation rule:

1. In the Application Navigator, 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 Rule section of the 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.
6. Use the dialog settings to configure the new rule.

The controls will change depending on the kind of validation rule you select. For more information about the different validation rules, see Section 3.5.3, "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 3.5.6, "How to Set the Severity Level for Validation Exceptions."

8. Click OK.

3.5.2 What Happens When You Add a Validation Rule

When you add a validation rule to a bean, JDeveloper updates the bean'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 3-5.

Example 3-5 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>
```

3.5.3 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
- 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

3.5.3.1 Validating Based on a Comparison

The Compare validator performs a logical comparison between an entity attribute and a value. When you add a Compare validator, you specify an operator and something to compare with. You can compare the following:

- Literal value

When you use a Compare validator with a literal value, the value in the attribute is compared against the specified literal value. When using this kind of comparison, it is important to consider data types and formats. The literal value must conform to the format specified by the data type of the entity attribute to which you are applying the rule. In all cases, the type corresponds to the type mapping for the entity attribute.

For example, an attribute of column type DATE maps to the `oracle.jbo.domain.Date` class, which accepts dates and times in the same format accepted by `java.sql.TimeStamp` and `java.sql.Date`. You can use format masks to ensure that the format of the value in the attribute matches that of the specified literal.

- Expression

For information on the expression option, see Section 3.5.4, "How to Use Groovy Expressions For Validation and Business 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 3.5, "Defining Validation Rules on Attributes Declaratively."

You will need to complete this task:

Create the desired data control structure files as described in Section 3.2.1, "How to Edit a Data Control."

To validate based on a comparison:

1. In the Application Navigator, 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 Rule section of the page, click the **Add Validation Rule** icon.
5. In the Add Validation Rule dialog, in the **Rule Type** dropdown list, select **Compare**. Note that the subordinate fields change depending on your choices.
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 3.5.5, "How to Create Validation Error Messages."
9. Click **OK**.

3.5.3.2 What Happens When You Validate Based on a Comparison

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

Example 3-5 shows the XML code for the validator on the `dateShipped` attribute in the `Ord` data control structure file.

3.5.3.3 Validating Using a List of Values

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

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 3.5, "Defining Validation Rules on Attributes Declaratively."

You will need to complete this task:

Create the desired data control structure files as described in Section 3.2.1, "How to Edit a Data Control."

To validate using a list of values:

1. In the Application Navigator, 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 Rule section of the 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 3.5.5, "How to Create Validation Error Messages."
9. Click **OK**.

3.5.3.4 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 bean's data control structure file.

3.5.3.5 Ensuring That a Value Falls Within a Certain Range

The Range validator performs a logical comparison between an entity attribute and a range of values. When you add a Range validator, you specify minimum and maximum literal values. The Range validator verifies that the value of the entity attribute falls within the range (or outside the range, if specified).

If you need to dynamically calculate the minimum and maximum values, or need to reference other attributes on the entity, use the Script Expression validator and provide a Groovy expression. For more information, see Section 3.8.1, "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 3.5, "Defining Validation Rules on Attributes Declaratively."

You will need to complete this task:

Create the desired data control structure files as described in Section 3.2.1, "How to Edit a Data Control."

To validate within a certain range:

1. In the Application Navigator, 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 Rule section of the 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 3.5.5, "How to Create Validation Error Messages."
9. Click **OK**.

3.5.3.6 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 3–6 shows the `quantity` attribute with a minimum of zero and a maximum of 10.

Example 3–6 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>
```

3.5.3.7 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.

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 3.5, "Defining Validation Rules on Attributes Declaratively."

You will need to complete this task:

Create the desired data control structure files as described in Section 3.2.1, "How to Edit a Data Control."

To validate against a number of bytes or characters:

1. In the Application Navigator, 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 Rule section of the 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.
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 3.5.5, "How to Create Validation Error Messages."
9. Click **OK**.

3.5.3.8 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, as shown in Example 3–7. For example, you might have a field where the user enters a password or PIN and the application wants to validate that it is at least 6 characters long, but not longer than 10. You would use the Length validator with the Between operator and set the minimum and maximum values accordingly.

Example 3–7 Validating the Length Between Two Values

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

3.5.3.9 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 3.5.4, "How to Use Groovy Expressions For Validation and Business 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 3.5, "Defining Validation Rules on Attributes Declaratively."

You will need to complete this task:

Create the desired data control structure files as described in Section 3.2.1, "How to Edit a Data Control."

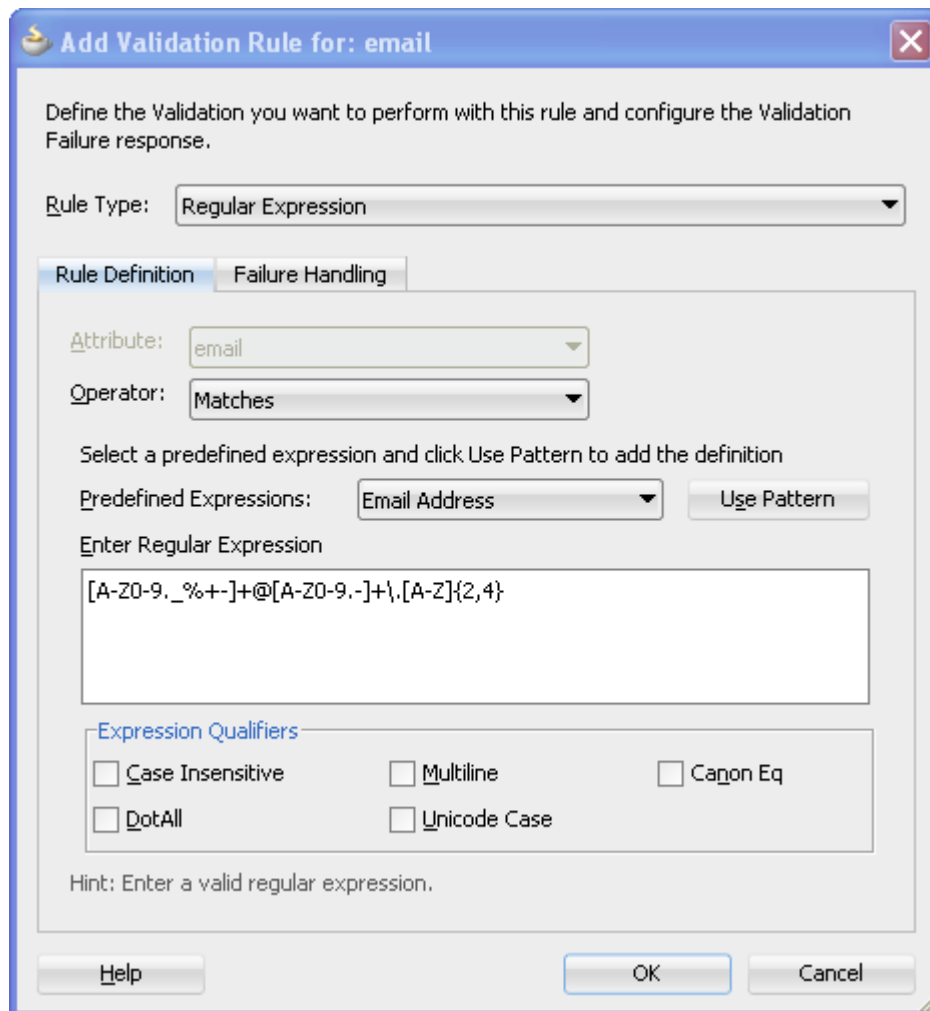
To validate using a regular expression:

1. In the Application Navigator, 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 Rule section of the 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 3.5.5, "How to Create Validation Error Messages."
9. Click **OK**.

Figure 3–3 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 3–3 Regular Expression Validator Matching Email Address



3.5.3.10 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 3–8 shows an `Email` attribute that must match a regular expression.

Example 3–8 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"/>
```

3.5.4 How to Use Groovy Expressions For Validation and Business 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 3.5, "Defining Validation Rules on Attributes Declaratively."

For more information about using Groovy expressions in your business logic, see Section 3.8, "Groovy Language Support."

3.5.4.1 Referencing Bean Methods in Groovy Validation Expressions

You can call methods on the bean using the `adf.source.dataProvider` property of the current object. The `adf.source.dataProvider` property allows you to access the bean 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 3-9 would call the `getXyzAbc()` method.

Example 3-9 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 bean 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.

3.5.4.2 Validating Using a True/False Expression

You can use a Groovy expression to return a true/false statement. The Script Expression validator *requires* that the expression either return `true` or `false`, or that it calls the `adf.error.raise/warn()` method. A common use of this feature would be to validate an attribute value, for example, to make sure that an account number is valid.

Note: Using the `adf.error.raise()` 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 3.5.5, "How to Create Validation Error Messages."

Before you begin:

It may be helpful to have an understanding of validation in data control structure files. For more information, see Section 3.5, "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 3.5.4, "How to Use Groovy Expressions For Validation and Business Rules."

You will need to complete this task:

Create the desired data control structure files as described in Section 3.2.1, "How to Edit a Data Control."

To validate using a true/false expression:

1. In the Application Navigator, 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 Rule section of the 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 3.5.5, "How to Create Validation Error Messages."
8. Click **OK**.

3.5.4.3 What Happens When You Add a True/False Expression

When you create a Groovy expression, it is saved in the bean's data control structure file. The Groovy expression is wrapped by a `<TransientExpression>` tag. For some Groovy expressions, the `<TransientExpression>` tag is wrapped by an `<validation:ExpressionValidationBean>` tag as well.

3.5.5 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.

3.5.5.1 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 3.5, "Defining Validation Rules on Attributes Declaratively."

You will need to complete this task:

Create the desired data control structure files as described in Section 3.2.1, "How to Edit a Data Control."

To create validation error messages:

1. In the Application Navigator, 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 section of the Attributes 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.

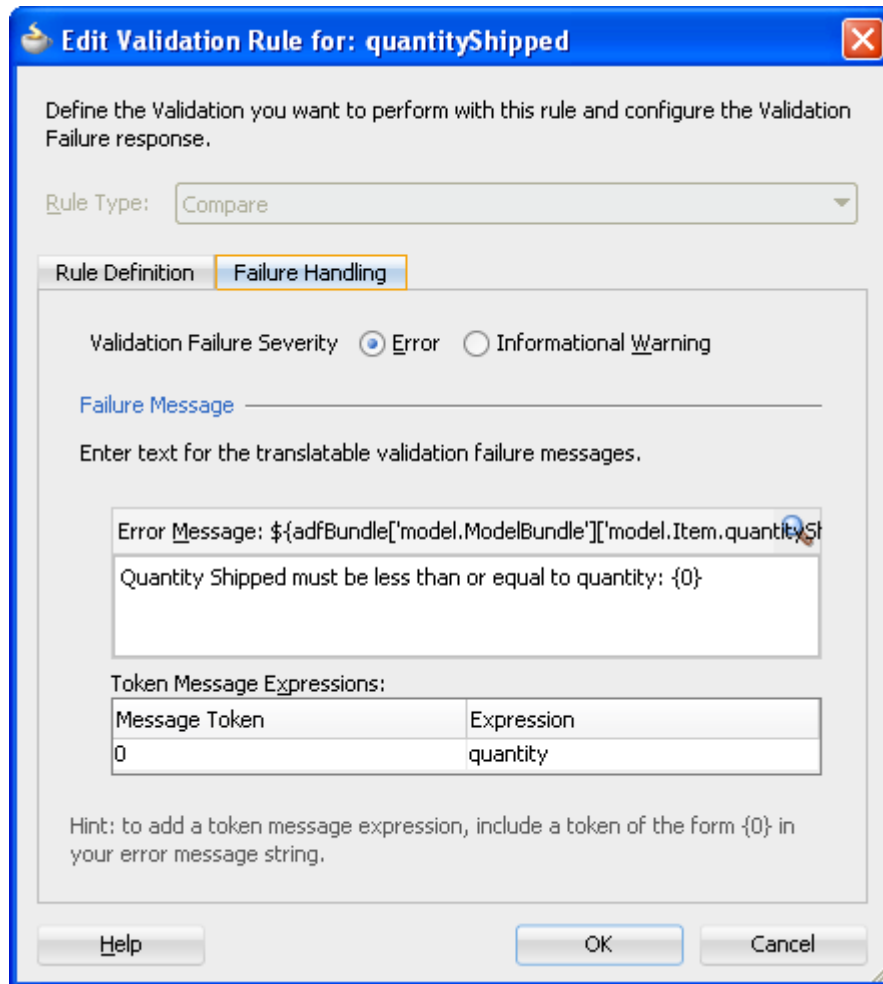
You can also define error messages in a message bundle file. To select a previously defined error message or to define a new one in a message bundle file, click the **Select Message** icon to open the Select Text Resource dialog.

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 3.5.5.3, "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 section. Then define the value of the message token in the **Token Message Expressions** list.

Figure 3–4 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 3.5.5.4, "Embedding a Groovy Expression in an Error Message."

8. Click **OK**.

Figure 3–4 Failure Handling Message for a Validation Rule

3.5.5.2 Localizing Validation Messages

The error message is a translatable string and is managed in the same way as translatable UI control hints in a message bundle file. To view the error message for the defined rule in the message bundle class, locate the `String` key in the message bundle that corresponds to the `ResId` property in the data control structure file entry for the validator.

3.5.5.3 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. For example, if an attribute value is x , then validate as follows, and if the validation fails, raise error messageA; whereas if the attribute value is y , then instead validate a different way and if validation fails, raise error messageB.

If the expression returns `false` (versus raising a specific error message using the `raise()` method), the validator calls the first error message associated with the validator.

The syntax of the `raise()` method takes one required parameter (the `msgId` to use from the message bundle), and optionally can take the `attrName` parameter.

You can use either `adf.error.raise()` or `adf.error.warn()` methods, depending on whether you want to throw an exception, or whether you want processing to continue, as described in Section 3.5.6, "How to Set the Severity Level for Validation Exceptions."

3.5.5.4 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 3-4 shows the failure message that contains a message token for a validation rule in the `Item.xml` data control structure file.

The expression shown in Figure 3-4 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 3.8, "Groovy Language Support."

3.5.6 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 a Informational Warning message if the user has a certain security clearance. For example, a store manager may want to be able to make changes that would surface as an error if a clerk tried to do the same thing.

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 3.5, "Defining Validation Rules on Attributes Declaratively."

You will need to complete this task:

Create the desired data control structure files as described in Section 3.2.1, "How to Edit a Data Control."

To set the severity level of a validation exception:

1. In the Application Navigator, 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 section of the Attributes 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.

3.6 Filtering Result Sets with Named Criteria

JDeveloper enables you to create named criteria for data control structure files. 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 on an entity bean 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 entity bean. 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.

Note: In addition to named criteria, there are also implicit criteria that are generated automatically for entity beans when you create a data control. Implicit criteria are based on the "FindAll" service methods that return all data for given tables.

Both named and implicit criteria are available under the Named Criteria node in the Data Controls panel. Implicit criteria appear as All Queriable Attributes.

3.6.1 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 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. The web page designer will see the named criteria in the JDeveloper Data Controls panel and, from them, easily create a search form. For more information about the utilizing the named criteria in the Data Controls panel, see Section 8.1.1, "Query Search Forms."

3.6.2 How to Create Named Criteria Declaratively

To define named criteria for the data control structure file that you wish to filter, you open the data control structure file in the overview editor and use the **Named Criteria** section of the View Criteria page. A dedicated editor that you open from the **Named Criteria** section 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 bean.

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 3.8, "Groovy Language Support."

When you define a named criteria, you control the source of the filtered results. You can limit the results to:

- Just the database table specified by the entity bean.
- Just the in-memory results of the accessor query.
- Both the database and the in-memory results of the accessor query.

Filtering on both database tables and the accessor's in-memory results allows you to filter rows that were created in the transaction but not yet committed to the database.

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 3.6, "Filtering Result Sets with Named Criteria."

You will need to complete these tasks:

- Create the desired data control structure files as described in Section 3.2.1, "How to Edit a Data Control."
- If the named criteria will use a bind variable in the operand, create the bind variable as described in Section 3.6.4, "How to Use Bind Variables in Named Criteria."

To define a named criteria:

1. In the Application Navigator, 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 View Criteria dialog, enter the name of the named criteria to identify its usage in your application.
5. In the **Criteria Execution Mode** dropdown list, decide how you want the named criteria to filter the query results.

You can limit the named criteria to filter the database table specified by the accessor query, the in-memory row set produced by the query, or both the database table and the in-memory results.

Choosing **Both** may be appropriate for situations where you want to include rows created as a result of enforced association consistency. In this case, in-memory filtering is performed after the initial fetch.

6. 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.

7. Select a named criteria item node in the named criteria hierarchy and define the added node in the **Criteria Item** section.
8. Select the desired **Attribute** for the criteria item. By default the editor adds the first one in the list.
9. 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.

10. 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 3.6.4, "How to Use Bind Variables in Named Criteria."
11. 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** conjunction 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** conjunction specifies that the query results meet either or both joined conditions. This is the default for named criteria groups.
12. 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.
13. 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.
14. 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.

15. Click **OK**.

3.6.3 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 View Criteria dialog.

3.6.4 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 can 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 includes dot notation access to attribute property values. Expressions are based on the Groovy scripting language, as described in Section 3.8, "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 3.6, "Filtering Result Sets with Named Criteria."

You will need to complete this task:

Create the desired data control structure files as described in Section 3.2.1, "How to Edit a Data Control."

To create a named bind variable

1. In the Application Navigator, 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 Bind 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 **Value** field. Optionally, click **Edit** to open the Expression dialog. The Expression dialog gives you a larger text area to write the expression.
 - When you want to define a default value, select the **Literal** value type and enter the literal value in the **Value** field.
6. In the Bind Variable dialog, click the **Control 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**.

3.6.5 What Happens When You Use Bind Variables in Named Criteria

Once you've added one or more named bind variables to a structure definition 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 bean's data control structure file. If you have defined UI hints for the bind variables, this information is saved in the model project's message bundle file along with other UI hints for the data control structure file.

3.6.6 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 Section 8.1.1, "Query Search Forms."

3.6.7 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 JDeveloper 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 Section 8.2, "Creating Query Search Forms."

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 3.6, "Filtering Result Sets with Named Criteria."

You will need to complete these tasks:

- Create the desired data control structure files as described in Section 3.2.1, "How to Edit a Data Control."
- Create the named criteria, as described in Section 3.6.2, "How to Create Named Criteria Declaratively."

To configure a named criteria for the user interface:

1. In the Application Navigator, 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 view 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, apply **Criteria Item UI Hints** 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.

If a rendered criteria item is of type `Date`, you must also define UI hints for the corresponding bean attribute. Set the bean attribute's Format Type hint to **Simple Date** and set the Format Mask to an appropriate value, as described in the "Defining Attribute Control Hints for View Objects" section of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework* guide. This will allow the search form to accept date values.

9. Click **OK**.

3.7 Testing Bean 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 example, if you have added validation rules or UI control 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 reproduce the issues in the Oracle ADF Model Tester to discover whether the problem lies in the view or controller layers of the application, or whether there is instead a problem in the business service layer application module itself.

3.7.1 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 Application Navigator.

Before you begin:

It may be helpful to have an understanding of the Oracle ADF Model Tester. For more information, see Section 3.7, "Testing Bean Metadata Using the Oracle ADF Model Tester."

Note: For EJB data controls, the tester only works on data controls for the remote session bean interface. If you created the data control based on the local bean interface, you need to add metadata for the remote bean interface to the `DataControls.dcx` file in order to use the tester. You can add metadata for the remote bean interface by right-clicking the session bean, choosing **Create Data Control**, and then selecting the **Remote** radio button in the Choose EJB Interface dialog.

When you later use the Data Controls panel to add bindings to pages, you will see high-level nodes for both the local and remote interfaces. Each node will contain identical hierarchies of data control collections that reference the same data control structure files. The only difference between the two hierarchies is in which session bean interface is referenced.

To test the bean metadata in your application:

1. In the Application Navigator, expand the project containing the desired data control.
2. Right-click the appropriate session bean and choose **Run**.

Wait for the application server to start and for the bean to be loaded on the server.

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

Alternatively, choose Debug when you want to run the application in the Oracle ADF Model Tester with debugging enabled. JDeveloper opens the debugger process panel in the Log window and the various debugger windows. For example, when debugging using the Oracle ADF Model Tester, you can view status message and exceptions, step in and out of source code, and manage breakpoints.

3. Right-click the `DataControls.dcx` file and choose **Run**.

The Oracle ADF Model Tester opens, as shown in Figure 3–5

Figure 3–5 Oracle ADF Model Tester

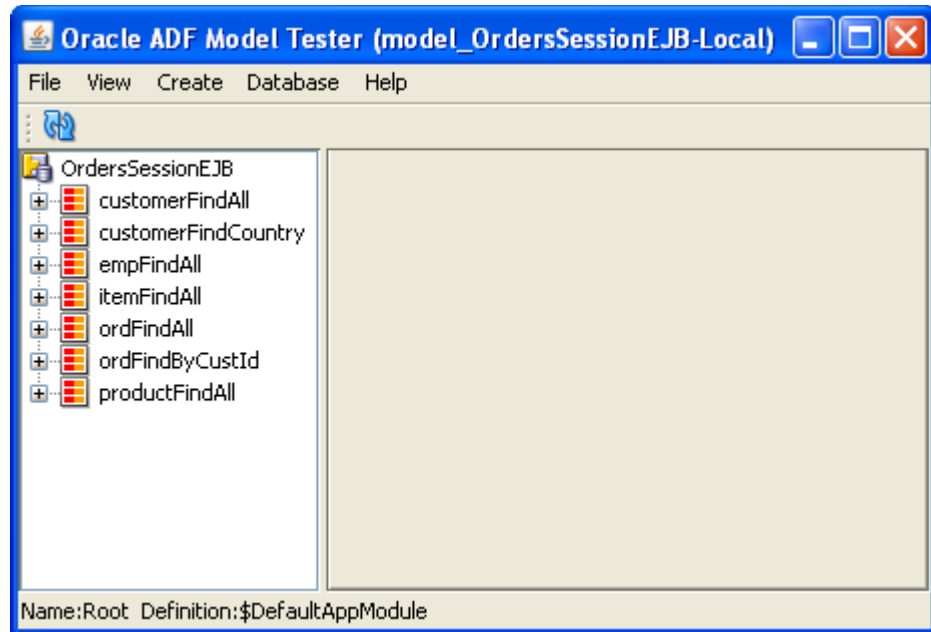


Table 3–1 gives an overview of the operations that the Oracle ADF Model Tester toolbar buttons perform when you display an entity bean.

Table 3–1 Oracle ADF Model Tester Toolbar Buttons








Button	Operation	Usage
	Move to ... row	Changes the current row displayed by the Oracle ADF Model Tester. Moves to the first, previous, next, or last row.
	Insert a new row	Creates and inserts a new row.
	Delete the current row	Deletes the current row.
	Save changes to the database	Runs the commit operation on any pending transactions. However, for EJB data controls, the changes are not actually committed to the database. This button is only available when transactional methods are implemented on the session facade.
	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 session facade.
	Specify view criteria	Displays the View Criteria dialog that you can use to create and apply view criteria to the result set.

Table 3–1 (Cont.) Oracle ADF Model Tester Toolbar Buttons

Button	Operation	Usage
	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.

3.7.2 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 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 model project JAR file, you also need to rebuild and redeploy that JAR 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 3.7, "Testing Bean Metadata Using the Oracle ADF Model Tester."

To reload the data model metadata in the running Oracle ADF Model Tester:

1. In the Application Navigator, right-click the **DataControls.dcx** file and choose **Run**.
2. In the Oracle ADF Model Tester, test the data model and determine any changes you want to make.
3. In JDeveloper, make the desired changes to your project.
4. 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 Application Navigator and choose **Rebuild**. The rebuilt JAR file is then deployed to the server.)

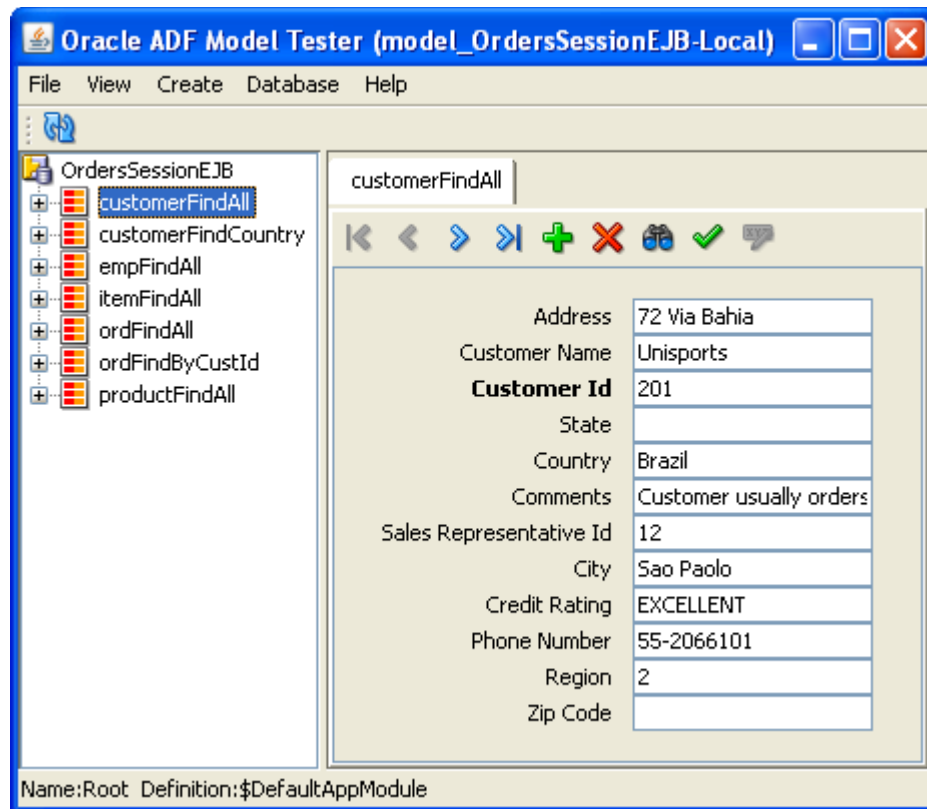
If you have only made changes to data control structure files, you do not need to rebuild the project.

5. Close the Oracle ADF Model Tester.
6. In the Application Navigator, right-click the **DataControls.dcx** file and choose **Run** to reopen the tester with the updated metadata.

3.7.3 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 accessor returned 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 3–6 shows the **customerFindAll** accessor returned collection that has been double-clicked to display the first record for this collection in the data view page on the right.

Figure 3–6 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 3.3.5, "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 view 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.

3.7.4 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 3.7, "Testing Bean Metadata Using the Oracle ADF Model Tester."

You will need to complete this task:

Start the tester as described in Section 3.7.1, "How to Run the Oracle ADF Model Tester."

To test business layer validation:

1. 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
```

2. In the toolbar, click the **Rollback** button to revert data to the previous state.

3.7.5 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 3.7, "Testing Bean 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.

3.7.6 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 3.7, "Testing Bean 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 3.3, "Working with Attributes."

You will need to complete this task:

Start the tester as described in Section 3.7.1, "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.

3.7.7 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:

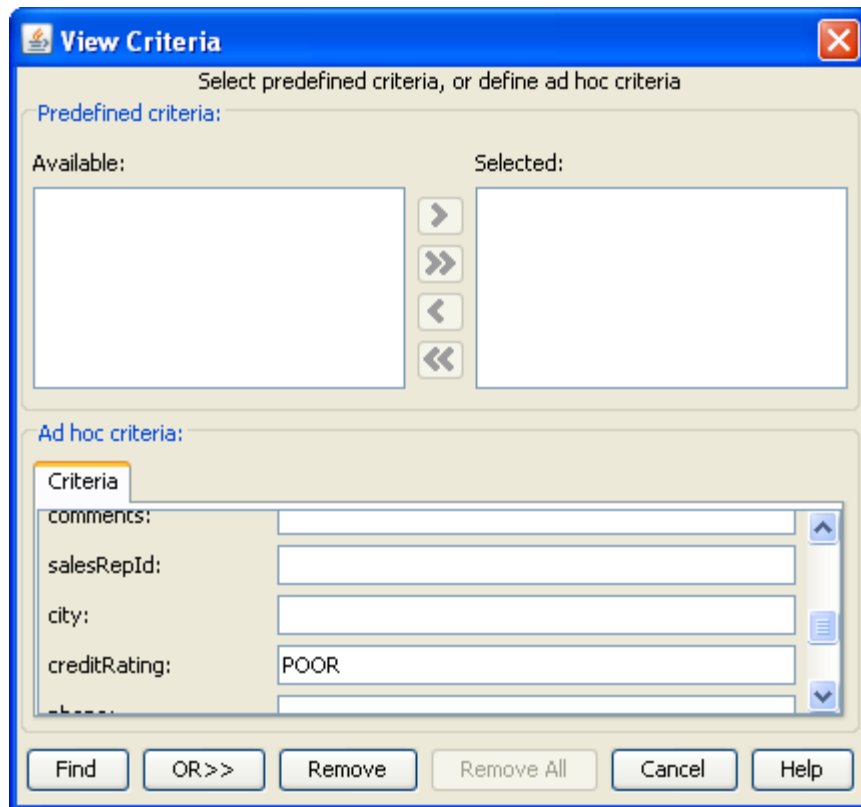
It may be helpful to have an understanding of the Oracle ADF Model Tester. For more information, see Section 3.7, "Testing Bean Metadata Using the Oracle ADF Model Tester."

You may also find it helpful to understand named criteria. For more information, see Section 3.6, "Filtering Result Sets with Named Criteria."

To test named criteria and ad hoc query criteria using the Oracle ADF Model Tester:

1. In the Application Navigator, expand the project containing the desired data control and session bean.
2. Right-click the session bean, choose **Run**, and then wait for the bean to be deployed to the built-in application server.
3. Right-click the **DataControls.dcx** file and choose **Run** in order to start the Oracle ADF Model Tester.
4. In the Oracle ADF Model Tester, double-click the accessor returned collection that you want to filter.
5. Click the **Specify View Criteria** toolbar button to test the named criteria.
6. 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 3–7 shows the filter to return all customers who possess a credit rating of `POOR`.

Figure 3–7 Oracle ADF Model Tester View Criteria Dialog

3.8 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 beans 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 3.5.4.2, "Validating Using a True/False Expression") or a Compare validator (see Section 3.5.3.1, "Validating Based on a Comparison").
- Define error message tokens for handling validation failure (see Section 3.5.5.4, "Embedding a Groovy Expression in an Error Message").
- Handle conditional execution of validators (see Section 3.5.5.3, "Raising Error Message Conditionally Using Groovy").

- Define the default value and optional recalculate condition for an entity bean attribute (see Section 3.3.3, "How to Define a Default Value Using a Groovy Expression").
- Determine the value of a transient attribute of a bean's data control structure file (see Section 3.4, "Adding Transient Attributes to a Bean").

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 web site:

- <http://groovy.codehaus.org/>

3.8.1 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 bean object on which the expression is being applied. Other accessible member names come from the context in which the Groovy script is applied.
 - Entity bean attributes and methods: The context is the bean'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 bean class.
 - Script validation rules: The context is the validator object (`JboValidatorContext`) merged with the entity bean on which the validator is applied. For details about keywords that you can use in this context, see Section 3.8.2, "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).

3.8.2 How to Reference ADF Methods and Attributes in Groovy Expressions

The simplest example of referencing data control members, including methods and attributes that the bean and the bean's data control structure file define, is to reference attributes that exist in the same bean 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 bean 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 entity that defines the attribute on which the expression is applied.

Note that when you want to reference the method of an entity bean 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 entity bean 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
```

Creating a Basic Databound Page

This chapter describes how to use the Data Controls panel to create databound forms using ADF Faces components. It includes information on creating text fields from individual attributes, generating entire forms from accessor returned collections, and creating forms for editing existing records and creating new records.

This chapter includes the following sections:

- Section 4.1, "About Creating a Basic Databound Page"
- Section 4.2, "Using Attributes to Create Text Fields"
- Section 4.3, "Creating a Basic Form"
- Section 4.4, "Incorporating Range Navigation into Forms"
- Section 4.5, "Creating a Form Using a Method That Takes Parameters"
- Section 4.6, "Creating a Form to Edit an Existing Record"
- Section 4.7, "Creating an Input Form"
- Section 4.8, "Modifying the UI Components and Bindings on a Form"

4.1 About Creating a Basic Databound Page

You can create UI pages that allow you to display and collect information using data controls created for your business services. For example, using the Data Controls panel, you can drag an attribute for an item, and then choose to display the value either as read-only text or as an input text field with a label. JDeveloper creates all the necessary JSF tag and binding code needed to display and update the associated data. For more information about the Data Controls panel and the declarative binding experience, see Chapter 2, "Using ADF Model Data Binding in a Java EE Web Application."

Instead of having to drop individual attributes, JDeveloper allows you to drop all attributes for an object at once as a form. The actual UI components that make up the form depend on the type of form dropped. You can create forms that display values, forms that allow users to edit values, and forms that collect values (input forms).

For example, the Summit demo application contains a page that allows users to view and edit information about an order, as shown in Figure 4-1. This form was created by dragging and dropping the `SOrdList` accessor collection from the Data Controls panel. (The `SOrdList` node is nested within the `customerFindAll` node.)

Figure 4–1 Edit Order Form in Summit Demo Application

The screenshot shows a form with the following fields and values:

- Order Id: 97
- Date Ordered: 28-Aug-1992
- Date Shipped: 17-Sep-1992
- Order Filled:
- Payment Type: CREDIT
- Sales Rep: Henry Giljum
- Order Total: \$84,000.00

Buttons: Save, Cancel

Once you create the UI components, you can then drop built-in operations as command UI components that allow you to navigate through the records in a collection or that allow users to operate on the data. For example, you can create a button that allows users to delete data objects displayed in the form. You can also modify the default components to suit your needs.

4.2 Using Attributes to Create Text Fields

JDeveloper allows you to create text fields declaratively in a WYSIWYG development environment for your JSF pages, meaning you can design most aspects of your pages without needing to look at the code. When you drag and drop items from the Data Controls panel, JDeveloper declaratively binds ADF Faces text UI components to attributes on a data control using an attribute binding.

4.2.1 How to Create a Text Field

To create a text field that can display or update an attribute, you drag and drop an attribute of a collection from the Data Controls panel.

Before you begin:

It may be helpful to have a general understanding of using data control attributes to create text fields. For more information, see Section 4.2, "Using Attributes to Create Text Fields."

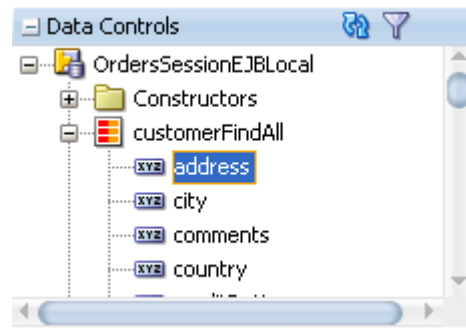
You will need to complete this task:

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

To create a bound text field:

1. From the Data Controls panel, select an attribute for a collection. For a description of the icons that represent attributes and other objects in the Data Controls panel, see Table 2–1.

For example, Figure 4–2 shows the address attribute under the customerFindAll accessor collection of the OrdersSessionEJBLocal data control in the Summit demo application. This is the attribute to drop to display or enter the first part of an address.

Figure 4–2 Attributes Associated with a Collection in the Data Controls Panel

2. Drag the attribute onto the page, and from the context menu choose the type of widget to display or collect the attribute value. For an attribute, you are given the following choices:

- **Text:**

- **ADF Input Text w/ Label:** Creates an ADF Faces `inputText` component with a nested `validator` component. The `label` attribute is populated.

Tip: For more information about validators and other attributes of the `inputText` component, see the "Using Input Components and Defining Forms" chapter of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

- **ADF Input Text:** Creates an ADF Faces `inputText` component with a nested `validator` component. The `label` attribute is not populated.
- **ADF Output Text w/ Label:** Creates a `panelLabelAndMessage` component that holds an ADF Faces `outputText` component. The `label` attribute on the `panelLabelAndMessage` component is populated.
- **ADF Output Text:** Creates an ADF Faces `outputText` component. No label is created.
- **ADF Output Formatted w/Label:** Same as ADF Output Text w/Label, but uses an `outputFormatted` component instead of an `outputText` component. The `outputFormatted` component allows you to add a limited amount of HTML formatting. For more information, see the "Displaying Output Text and Formatted Output Text" section of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*
- **ADF Output Formatted:** Same as ADF Output Formatted w/Label, but without the label.
- **ADF Label:** An ADF Faces `outputLabel` component.
- **Single Selections:** Creates single selection lists. For more information about creating lists on a JSF page, see Chapter 7, "Creating Databound Selection Lists."

For the purposes of this chapter, only the text components (and not the lists) will be discussed.

4.2.2 What Happens When You Create a Text Field

When you drag an attribute onto a JSF page and drop it as a UI component, among other things, a page definition file is created for the page (if one does not already exist). For a complete account of what happens when you drag an attribute onto a page, see Section 2.3.2, "What Happens When You Use the Data Controls Panel to Create UI Components." Bindings for the iterator and attributes are created and added to the page definition file. Additionally, the necessary JSPX page code for the UI component is added to the JSF page.

4.2.2.1 Creating and Using Iterator Bindings

Whenever you create UI components on a page by dropping an item that is part of a collection from the Data Controls panel (or you drop the whole collection as a form or table), JDeveloper creates an iterator binding if it does not already exist. An iterator binding references an iterator for the data collection, which facilitates iterating over its data objects. It also manages currency and state for the data objects in the collection. An iterator binding does not actually access the data. Instead, it simply exposes the object that can access the data and it specifies the current data object in the collection. Other bindings then refer to the iterator binding in order to return data for the current object or to perform an action on the object's data. Note that the iterator binding is not an iterator. It is a binding to an iterator.

For example, if you drop the address attribute from the `customerFindAll` collection, JDeveloper creates an iterator binding for the `OrdersSessionEJBLocal` data control and `accessorIterator` binding for the `customerFindAll` collection, which in turn has the `OrdersSessionEJBLocal` iterator as its master binding.

Tip: There is one accessor iterator binding created for each collection returned from an accessor. This means that when you individually drop two attributes from the same accessor (or drop the attribute twice), they use the same binding. This is fine, unless you need the binding to behave differently for the different components. In that case, you will need to manually create separate iterator bindings.

The iterator binding's `rangeSize` attribute determines how many rows of data are fetched from a data control each time the iterator binding is accessed. This attribute gives you a relative set of 1-*n* rows positioned at some absolute starting location in the overall row set. By default, the attribute is set to 25.

Example 4-1 shows the iterator bindings created when you drop an attribute from the `customerFindAll` accessor collection.

Example 4-1 Page Definition Code for an Iterator Accessor Binding

```
<executables>
  <variableIterator id="variables"/>
  <iterator Binds="root" RangeSize="25"
    DataControl="OrdersSessionEJBLocal"
    id="OrdersSessionEJBLocalIterator"/>
  <accessorIterator MasterBinding="OrdersSessionEJBLocalIterator"
    Binds="customerFindAll" RangeSize="25"
    DataControl="OrdersSessionEJBLocal"
    BeanClass="model.Customer" id="customerFindAllIterator"/>
</executables>
```

This metadata allows the ADF binding container to access the attribute values. Because the iterator binding is an executable, by default it is invoked when the page is

loaded, thereby allowing the iterator to access and iterate over the collection returned by the `customerFindAll` accessor. This means that the iterator will manage all the objects in the collection, including determining the current row in the collection or determining a range of address objects.

4.2.2.2 Creating and Using Value Bindings

When you drop an attribute from the Data Controls panel, JDeveloper creates an attribute binding that is used to bind the UI component to the attribute's value. This type of binding presents the value of an attribute for a single object in the current row in the collection. Value bindings can be used both to display and to collect attribute values.

For example, if you drop the `address` attribute under the `customerFindAll` accessor as an ADF Output Text w/Label widget onto a page, JDeveloper creates an attribute binding for the `address` attribute. This allows the binding to access the attribute value of the current record. Example 4-2 shows the attribute binding for `address` created when you drop the attribute from the `customerFindAll` accessor. Note that the attribute value references the iterator named `customerFindAllIterator`.

Example 4-2 Page Definition Code for an Attribute Binding

```
<bindings>
  <attributeValues IterBinding="customerFindAllIterator" id="address">
    <AttrNames>
      <Item Value="address" />
    </AttrNames>
  </attributeValues>
</bindings>
```

For information regarding the attribute binding element properties, see the "Oracle ADF Binding Properties" appendix of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

4.2.2.3 Using EL Expressions to Bind UI Components

When you create a text field by dropping an attribute from the Data Controls panel, JDeveloper creates the UI component associated with the widget dropped by writing the corresponding tag to the JSF page.

For example, when you drop the `address` attribute as an Output Text w/Label widget, JDeveloper inserts the tags for a `panelLabelAndMessage` component and an `outputText` component. It creates an EL expression that binds the `label` attribute of the `panelLabelAndMessage` component to the `label` property of hints created for the `address`'s binding. This expression evaluates to the label UI hint set in the data control for the bean. It creates another expression that binds the `outputText` component's `value` attribute to the `inputValue` property of the `address` binding, which evaluates to the value of the `address` attribute for the current row. An ID is also automatically generated for both components.

Tip: JDeveloper automatically generates IDs for all ADF Faces components. You can override these values as needed.

Example 4-3 shows the code generated on the JSF page when you drop the `address` attribute as an Output Text w/Label widget.

Example 4–3 JSF Page Code for an Attribute Dropped as an Output Text w/Label

```
<af:panelLabelAndMessage label="#{bindings.address.hints.label}"
    id="plam1">
    <af:outputText value="#{bindings.address.inputValue}" id="ot1"/>
</af:panelLabelAndMessage>
```

If instead you drop the `address` attribute as an Input Text w/Label widget, JDeveloper creates an `inputText` component. As Example 4–4 shows, similar to the output text component, the value is bound to the `inputValue` property of the address binding. Additionally, the following properties are also set:

- `label`: Bound to the attribute's `label` UI hint.
- `required`: Bound to the mandatory property, which in turn references the attribute's `isNotNull` property (which is represented by the **Mandatory** option in the Details section for the attribute in the overview editor for the bean's data control structure file).
- `columns`: Bound to the attribute's `displayWidth` UI hint, which determines how wide the text box will be.
- `maxLength`: Bound to the attribute's **Precision** option. This property determines the maximum number of characters per line that can be entered into the field.
- `shortDesc`: Bound to the tooltip UI hint.

In addition, JDeveloper adds a validator component.

Example 4–4 JSF Page Code for an Attribute Dropped as an Input Text w/Label

```
<af:inputText value="#{bindings.address.inputValue}"
    label="#{bindings.address.hints.label}"
    required="#{bindings.address.hints.mandatory}"
    columns="#{bindings.address.hints.displayWidth}"
    maxLength="#{bindings.address.hints.precision}"
    shortDesc="#{bindings.address.hints.tooltip}" id="it1">
    <f:validator binding="#{bindings.address.validator}"/>
</af:inputText>
```

You can change any of these values to suit your needs. For example, the `isNotNull` property for the attribute in the data control is set to `false` by default, which means that the `required` attribute on the component will evaluate to `false` as well. You can override this value by setting the `required` attribute on the component to `true`. If you decide that all instances of the attribute should be mandatory, then you can change the **Mandatory** option for the attribute in the data control structure file, and all instances will be required. For more information about editing attributes in a data control, see Section 3.3, "Working with Attributes."

4.3 Creating a Basic Form

Instead of dropping each of the individual attributes of a collection to create a form, you can create a complete form that displays or collects data for all the attributes on an object. For example, in the Summit demo application, the form on the `EditOrder.jspx` page was created by dropping the `ordFindAll` accessor collection from the Data Controls panel.

You can also create forms that provide more functionality than simply displaying data from a collection. For information about creating a form that allows a user to update data, see Section 4.6, "Creating a Form to Edit an Existing Record." For information

about creating forms that allow users to create a new object for the collection, see Section 4.7, "Creating an Input Form." You can also create search forms. For more information, see Chapter 8, "Creating Databound Search Forms."

4.3.1 How to Create a Form

To create a form using a data control, you bind the UI components to the attributes on the corresponding object in the data control. JDeveloper allows you to do this declaratively by dragging and dropping a returned collection from the Data Controls panel.

Before you begin:

It may be helpful to have a general understanding of creating forms from data controls. For more information, see Section 4.3, "Creating a Basic Form."

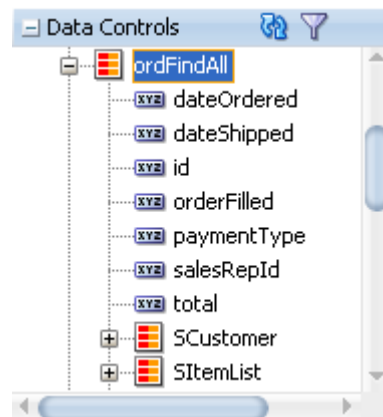
You will need to complete this task:

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

To create a basic form:

1. From the Data Controls panel, select the collection that returns the data you wish to display. Figure 4-3 shows the `ordFindAll` accessor returned collection.

Figure 4-3 *productFindAll* Accessor in the Data Controls Panel



2. Drag the collection onto the page, and from the context menu choose the type of form that will be used to display or collect data for the object. For a form, you are given the following choices:
 - **ADF Form:** Launches the Edit Form Fields dialog that allows you to select individual attributes instead of having JDeveloper create a field for every attribute by default. It also allows you to select the label and UI component used for each attribute. By default, ADF `inputText` components are used for most attributes. Each `inputText` component has the `label` attribute populated.

Attributes that are dates use the `InputDate` component. Additionally, if a UI hint has been created for an attribute, or if the attribute has been configured to be a list, then the component set by the hint is used instead. `InputText` components contain a validator tag that allows you to set up validation for the attribute, and if the attribute is a number or a date, a converter is also included.

Tip: For more information about validators, converters, and other attributes of the `inputText` component, see the "Using Input Components and Defining Forms" chapter of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

- **ADF Read-Only Form:** Same as the ADF Form, but read-only `outputText` components are used. Since the form is meant to display data, no `validator` tags are added (converters are included). Attributes of type `Date` use the `outputText` component when in a read-only form. All components are placed inside `panelLabelAndMessage` components, which have the `label` attribute populated. The `panelLabelAndMessage` components are placed inside a `panelFormLayout` component.
3. In the Edit Form Fields dialog, configure your form.
You can add, remove, and reorder form fields.

You can also include navigational controls that allow users to navigate through all the data objects in the collection. For more information, see Section 4.4, "Incorporating Range Navigation into Forms." You can also include a **Submit** button used to submit the form. This button submits the HTML form and applies the data in the form to the bindings as part of the JSF/ADF page lifecycle. For additional help in using the dialog, click **Help**. All UI components are placed inside a `panelFormLayout` component.
 4. If you are building a form that allows users to update data, you now need to drag and drop a method that will perform the update. For more information, see Section 4.6, "Creating a Form to Edit an Existing Record."

4.3.2 What Happens When You Create a Form

Dropping an object as a form from the Data Controls panel has the same effect as dropping a single attribute, except that multiple attribute bindings and associated UI components are created. The attributes on the UI components (such as `value`) are bound to properties on that attribute's binding object (such as `inputValue`) or to the values of control hints set on the corresponding service. Example 4–5 shows some of the code generated on the JSF page when you drop the `SOrdList` accessor collection as a default ADF form to create the Edit Order form.

Note: If an attribute is marked as hidden on the associated data control structure file, then no corresponding UI is created for it.

Example 4–5 Code on a JSF Page for an Input Form

```
<af:panelFormLayout id="pf11">
  <af:inputText value="#{bindings.id.inputValue}"
    label="#{bindings.id.hints.label}"
    required="#{bindings.id.hints.mandatory}"
    columns="#{bindings.id.hints.displayWidth}"
    maximumLength="#{bindings.id.hints.precision}"
    shortDesc="#{bindings.id.hints.tooltip}" id="it1">
    <f:validator binding="#{bindings.id.validator}"/>
  </af:inputText>
  <af:inputDate value="#{bindings.dateOrdered.inputValue}"
    label="#{bindings.dateOrdered.hints.label}"
    required="#{bindings.dateOrdered.hints.mandatory}"
```

```

        columns="#{bindings.dateOrdered.hints.displayWidth}"
        shortDesc="#{bindings.dateOrdered.hints.tooltip}" id="id1">
    <f:validator binding="#{bindings.dateOrdered.validator}" />
    <af:convertDateTime pattern="#{bindings.dateOrdered.format}" />
</af:inputDate>
<af:inputDate value="#{bindings.dateShipped.inputValue}"
    label="#{bindings.dateShipped.hints.label}"
    required="#{bindings.dateShipped.hints.mandatory}"
    columns="#{bindings.dateShipped.hints.displayWidth}"
    shortDesc="#{bindings.dateShipped.hints.tooltip}" id="id2">
    <f:validator binding="#{bindings.dateShipped.validator}" />
    <af:convertDateTime pattern="#{bindings.dateShipped.format}" />
</af:inputDate>
<af:selectBooleanCheckbox value="#{bindings.orderFilled.inputValue}"
    label="#{bindings.orderFilled.label}"
    shortDesc="#{bindings.orderFilled.hints.tooltip}"
    id="sbc1" />
<af:selectOneChoice value="#{bindings.paymentType.inputValue}"
    label="#{bindings.paymentType.label}"
    required="#{bindings.paymentType.hints.mandatory}"
    shortDesc="#{bindings.paymentType.hints.tooltip}"
    id="soc2">
    <f:selectItems value="#{bindings.paymentType.items}" id="si2" />
</af:selectOneChoice>
. . .
</af:panelFormLayout>

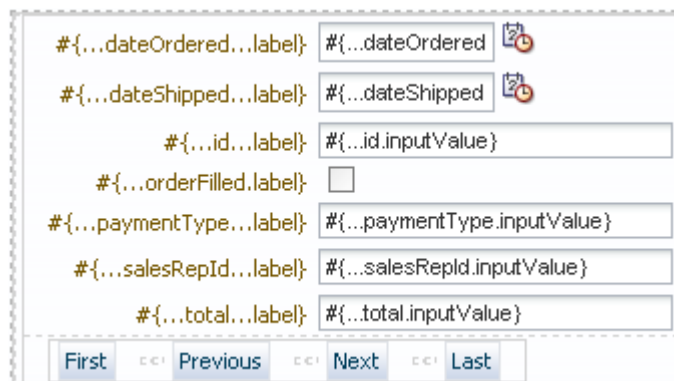
```

Note: For information regarding the validator and converter tags, see the "Validating and Converting Input" chapter of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

4.4 Incorporating Range Navigation into Forms

When you create an ADF Form, if you elect to include navigational controls, JDeveloper includes ADF Faces command components bound to existing navigational logic on the data control. This built-in logic allows the user to navigate through all the data objects in the collection. For example, Figure 4-4 shows a form that would be created if you dragged the `ordFindAll` accessor and dropped it as an ADF Form that uses navigation.

Figure 4-4 Navigation in a Form



4.4.1 How to Insert Navigation Controls into a Form

By default, when you choose to include navigation when creating a form using the Data Controls panel, JDeveloper creates **First**, **Last**, **Previous**, and **Next** buttons that allow the user to navigate within the collection.

You can also add navigation buttons to an existing form manually.

Before you begin:

It may be helpful to have a general understanding of navigation controls. For more information, see Section 4.4, "Incorporating Range Navigation into Forms."

You will need to complete this task:

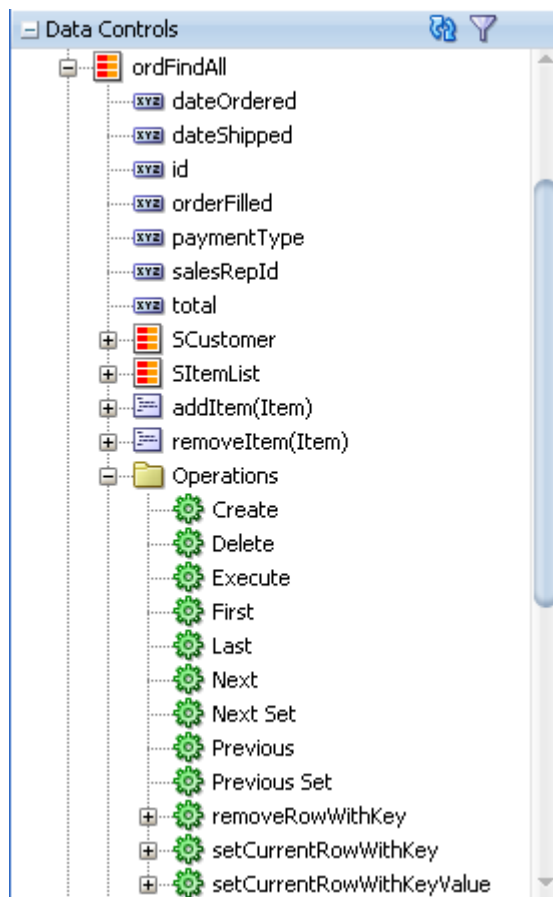
Create a data control for your session bean, as described in Section 2.2.1, "How to Create ADF Data Controls."

To manually add navigation buttons:

1. From the Data Controls panel, select the operation associated with the collection of objects on which you wish the operation to execute, and drag it onto the JSF page.

For example, if you want to navigate through a collection of orders, you would drag the **Next** operation associated with the `OrdFindAll` accessor. Figure 4–5 shows the operations associated with the `OrdFindAll` accessor.

Figure 4–5 Operations Associated with a Collection



2. From the ensuing context menu, choose either **ADF Button** or **ADF Link**.

Tip: You can also drop the **First**, **Previous**, **Next**, and **Last** buttons at once. To do so, drag the corresponding collection, and from the context menu, choose **Navigation > ADF Navigation Buttons**.

4.4.2 What Happens When You Create Command Buttons

When you drop any operation as a command component, JDeveloper:

- Defines an action binding in the page definition file for the associated operations
- Configures the iterator binding to use partial page rendering for the collection
- Inserts code in the JSF page for the command components

4.4.2.1 Action Bindings for Built-in Navigation Operations

Action bindings execute business logic. For example, they can invoke built-in methods on the action binding object. These built-in methods operate on the iterator or on the data control itself, and are represented as operations in the Data Controls panel. JDeveloper provides navigation operations that allow users to navigate forward, backwards, to the first object in the collection, and to the last object.

Like value bindings, action bindings for operations contain a reference to the iterator binding when the action binding is bound to one of the iterator-level actions, such as `Next` or `Previous`. These types of actions are performed by the iterator, which determines the current object and can therefore determine the correct object to display when a navigation button is clicked.

Action bindings use the `RequiresUpdateModel` property, which determines whether or not the model needs to be updated before the action is executed. In the case of navigation operations, by default this property is set to `true`, which means that any changes made at the view layer must be moved to the model before navigation can occur. Example 4–6 shows the action bindings for the navigation operations.

Example 4–6 Page Definition Code for an Operation Action Binding

```
<action IterBinding="ordFindAllIterator" id="First"
    RequiresUpdateModel="true" Action="first"/>
<action IterBinding="ordFindAllIterator" id="Previous"
    RequiresUpdateModel="true" Action="previous"/>
<action IterBinding="ordFindAllIterator" id="Next"
    RequiresUpdateModel="true" Action="next"/>
<action IterBinding="ordFindAllIterator" id="Last"
    RequiresUpdateModel="true" Action="last"/>
```

4.4.2.2 Iterator RangeSize Attribute

Iterator bindings have a `rangeSize` attribute that the binding uses to determine the number of data objects to make available for the page for each iteration. This attribute helps in situations when the number of objects in the data source is quite large. Instead of returning all objects, the iterator binding returns only a set number, which then become accessible to the other bindings. Once the iterator reaches the end of the range, it accesses the next set. Example 4–7 shows the default range size for the `ordFindAll` iterator.

Example 4–7 RangeSize Attribute for an Iterator

```
<accessorIterator MasterBinding="OrdersSessionEJBLocalIterator"
    Binds="ordFindAll" RangeSize="25"
    DataControl="OrdersSessionEJBLocal" BeanClass="model.Ord"
```

```
id="ordFindAllIterator" ChangeEventPolicy="ppr" />
```

Note: This `rangeSize` attribute is not the same as the `rows` attribute on a table component.

By default, the `rangeSize` attribute is set to 25. This means that a user can view 25 objects, navigating back and forth between them, without needing to access the data source. The iterator keeps track of the current object. Once a user clicks a button that requires a new range (for example, clicking the **Next** button on object number 25), the binding object executes its associated method against the iterator, and the iterator retrieves another set of 25 records. The bindings then work with that set. You can change this setting as needed. You can set it to -1 to have the full record set returned.

Note: When you create a navigable form using the Data Controls panel, the `CacheResults` property on the associated iterator is set to `true`. This ensures that the iterator’s state, including currency information, is cached between requests, allowing it to determine the current object. If this property is set to `false`, navigation will not work.

Table 4–1 shows the built-in navigation operations provided on data controls and the result of invoking the operation or executing an event bound to the operation.

Table 4–1 Built-in Navigation Operations

Operation	When invoked, the associated iterator binding will...
First	Move its current pointer to the beginning of the result set.
Last	Move its current pointer to the end of the result set.
Previous	Move its current pointer to the preceding object in the result set. If this object is outside the current range, the range is scrolled backward a number of objects equal to the range size.
Next	Move its current pointer to the next object in the result set. If this object is outside the current range, the range is scrolled forward a number of objects equal to the range size.
Previous Set	Move the range backward a number of objects equal to the range size attribute.
Next Set	Move the range forward a number of objects equal to the range size attribute.

4.4.2.3 EL Expressions Used to Bind to Navigation Operations

When you create command components using navigation operations, the command components are placed in a `panelGroupLayout` component. JDeveloper creates an EL expression that binds a navigational command button’s `actionListener` attribute to the `execute` property of the action binding for the given operation.

At runtime an action binding will be an instance of the `FacesCtrlActionBinding` class, which extends the core `JUCtrlActionBinding` implementation class. The `FacesCtrlActionBinding` class adds the following methods:

- public void execute(ActionEvent event):** This is the method that is referenced in the `actionListener` property, for example `{bindings.First.execute}`.

This expression causes the binding's operation to be invoked on the iterator when a user clicks the button. For example, the **First** command button's `actionListener` attribute is bound to the `execute` method on the `First` action binding.

- `public String outcome():` This can be referenced in an Action property, for example `#{bindings.Next.outcome}`.

This can be used for the result of a method action binding (once converted to a `String`) as a JSF navigation outcome to determine the next page to navigate to.

Note: Using the `outcome` method on the action binding implies tying the view-controller layer too tightly to the model, so it should rarely be used.

Every action binding for an operation has an `enabled` boolean property that Oracle ADF sets to `false` when the operation should not be invoked. By default, JDeveloper binds the UI component's `disabled` attribute to this value to determine whether or not the component should be enabled. For example, the UI component for the **First** button has the following as the value for its `disabled` attribute:

```
#{!bindings.First.enabled}
```

This expression evaluates to `true` whenever the binding is not enabled, that is, when the operation should not be invoked, thereby disabling the button. In this example, because the framework will set the `enabled` property on the binding to `false` whenever the first record is being shown, the **First** button will automatically be disabled because its `disabled` attribute is set to be `true` whenever `enabled` is `False`. For more information about the `enabled` property, see the "Oracle ADF Binding Properties" appendix of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

Example 4–8 shows the code generated on the JSF page for navigation operation buttons. For more information about the `partialSubmit` attribute on the button, see the "Enabling Partial Page Rendering Declaratively" section of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

Example 4–8 JSF Code for Navigation Buttons Bound to ADF Operations

```
<f:facet name="footer">
  <af:panelGroupLayout>
    <af:commandButton actionListener="#{bindings.First.execute}"
      text="First"
      disabled="#{!bindings.First.enabled}"
      partialSubmit="true" id="cb1"/>
    <af:commandButton actionListener="#{bindings.Previous.execute}"
      text="Previous"
      disabled="#{!bindings.Previous.enabled}"
      partialSubmit="true" id="cb2"/>
    <af:commandButton actionListener="#{bindings.Next.execute}"
      text="Next"
      disabled="#{!bindings.Next.enabled}"
      partialSubmit="true" id="cb3"/>
    <af:commandButton actionListener="#{bindings.Last.execute}"
      text="Last"
      disabled="#{!bindings.Last.enabled}"
      partialSubmit="true" id="cb4"/>
  </af:panelGroupLayout>
</f:facet>
```

```
</af:panelGroupLayout>
</f:facet>
```

4.5 Creating a Form Using a Method That Takes Parameters

There may be cases where a page needs information before it can display content. For these types of pages, you create the form using a returned collection from a method that takes parameters. The requesting page needs to supply the value of the parameters in order for the method to execute.

4.5.1 How to Create a Form or Table Using a Method That Takes Parameters

To create forms that require parameters, you must be able to access the values for the parameters in order to determine the record(s) to return. You access those values by adding logic to a command button on another page that will set the parameter value on some object that the method can then access.

Before you begin:

It may be helpful to have a general understanding of parameter forms. For more information, see Section 4.5, "Creating a Form Using a Method That Takes Parameters."

You will need to complete these tasks:

- Create a data control for your session bean, as described in Section 2.2.1, "How to Create ADF Data Controls."
- Create or identify a method on your session bean that will return the items needed to be displayed in your form. For example, the `OrderSessionEJBBean.java` class contains the `mergeOrd(Ord)` method.

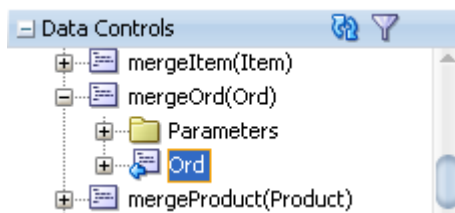
Tip: If you add a service method to your session bean after you have generated the data control, you need to refresh the Data Controls panel in order to display the new method in the panel. To refresh the panel, click the **Refresh** icon.

To create a form or table that uses parameters:

1. From the Data Controls panel, drag a collection that is a return of a method that takes a parameter or parameters and drop it as any type of form.

For example, to create a form that displays when you click a button to edit an order, you would drag and drop the `Ord` return, as shown in Figure 4–6.

Figure 4–6 Return of a Custom Method That Takes Parameters



2. In the Edit Form Fields dialog, configure the form as needed and click **OK**.
For help in using the dialogs, click **Help**.

Because the method takes parameters, the Edit Action Binding dialog opens, asking you to set the value of the parameters.

3. In the Action Binding Editor, enter the value for each parameter by clicking the browse (...) icon in the **Value** field to open the EL Expression Builder. Select the node that represents the value for the parameter.

This editor uses the value to create the `NamedData` element that will represent the parameter when the method is executed. Since you are dropping a collection that is a return of the method (unlike a method bound to a command button), this method will be run when the associated iterator is executed as the page is loaded. You want the parameter value to be set before the page is rendered. This means the `NamedData` element needs to get this value from wherever the sending page has set it.

Tip: If you are creating a form that displays based on input from the same page, you can use an ADF Parameter Form to create the action bindings. First drag the method from the Data Controls panel and drop it as an ADF Parameter Form. Then drag the method return and drop it as any kind of form.

4.5.2 What Happens When You Create a Form Using a Method That Takes Parameters

When you use a return of a method that takes parameters to create a form, JDeveloper:

- Creates an action binding for the method, a method iterator binding for the result of the method, and attribute bindings for each of the attributes of the object, or in the case of a table, a table binding. It also creates `NamedData` elements for each parameter needed by the method.
- Inserts code in the JSF page for the form using ADF Faces components.

Example 4–9 shows the action method binding created when you drop the `Ord` return of the `mergeOrd(Ord)` method, where the value for the parameter was set to the current row data provider of the `SOrdListIterator`.

Example 4–9 Method Action Binding for a Method Return

```
<bindings>
  <methodAction id="mergeOrd" RequiresUpdateModel="true"
    Action="invokeMethod" MethodName="mergeOrd"
    IsViewObjectMethod="false" DataControl="OrdersSessionEJBLocal"
    InstanceName="Sdata.OrdersSessionEJBLocal.dataProvider"
    ReturnName="data.OrdersSessionEJBLocal.methodResults.mergeOrd_
      OrdersSessionEJBLocal_dataProvider_mergeOrd_result">
    <NamedData NDName="productId"
      NDValue="#{bindings.SOrdListIterator.currentRow.dataProvider}"
      NDType="model.Ord"/>
  </methodAction>
  ...
</bindings>
```

4.5.3 What Happens at Runtime: Setting Parameters for a Method

Unlike a method executed when a user clicks a command button, a method used to create a form is executed as the page is loaded. When the method is executed in order to return the data for the page, the method evaluates the EL expression for the `NamedData` element and uses that value as its parameter. It is then able to return the

correct data. If the method takes more than one parameter, each is evaluated in turn to set the parameters for the method.

For example, when the `EditOrder` page loads, it takes the value of the `currentRow.dataProvider` parameter on the `SOrdListIterator` and sets it as the value of the parameter needed by the `mergeOrd(Ord)` method. Once that method executes, it returns only the record that matches the value of the parameter. Because you dropped the return of the method to create the form, that return is the product that is displayed.

4.5.4 What You May Need to Know About Setting Parameters with Methods

There may be cases where an action on one page needs to set parameters that will be used to determine application functionality. For example, you can create a command button on one page that will navigate to another page, but a component on the resulting page will display only if the parameter value is `false`.

You can use a managed bean to pass this parameter between the pages, and to contain the method that is used to check the value of this parameter. A `setPropertyListener` component with the `type` property set to `action`, which is nested in the command button, is then used to set parameter. For more information about setting parameters using methods, see the "Setting Parameter Values Using a Command Component" section of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

Note: If you are using task flows, you can use the task flow parameter passing mechanism. For more information, see the "Using Parameters in Task Flows" chapter of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

4.5.5 What You May Need to Know About Using Contextual Events Instead of Parameters

Often a page or a region within a page needs information from somewhere else on the page or from a different region (for more information about regions, see the "Using Task Flows as Regions" sections of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*). While you can pass parameters to obtain that information, doing so makes sense only when the parameters are well known and the inputs are EL-accessible to the page. Parameters are also useful when a task flow may need to be restarted if the parameter value changes.

However, suppose you have a task flow with multiple page fragments that contain various interesting values that could be used as input on one of the pages in the flow. If you were to use parameters to pass the value, the task flow would need to surface output parameters for the union of each of the interesting values on each and every fragment. Instead, for each fragment that contains the needed information, you can define a contextual event that will be raised when the page is submitted. The page or fragment that requires the information can then subscribe to the various events and receive the information through the event.

You can create and configure contextual events using a page definition file. For more information, see the "Using Contextual Events" section of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

4.6 Creating a Form to Edit an Existing Record

You can create a form that allows a user to edit the current data, and then commit those changes to the data source. To do this, you use methods that can modify data records associated with the collection to create command buttons. For example, you can use the default `mergeOrd(Ord)` method to create a button that allows a user to update an order.

If the page is not part of a bounded task flow, you need to use the `merge` or `persist` method associated with the collection to merge the changes back into the collection. For more information about the difference between the two, see Section 4.6.3, "What You May Need to Know About the Difference Between the Merge and Persist Methods." If the page is part of a transaction within a bounded task flow, you use the `commit` and `rollback` operations to resolve the transaction in a task flow return activity. For more information, see the "Using Task Flow Return Activities" section of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

4.6.1 How to Create Edit Forms

To use methods on a form, you follow the same procedures as with the operations.

Before you begin:

It may be helpful to have a general understanding of editable forms. For more information, see Section 4.6, "Creating a Form to Edit an Existing Record."

You will need to complete these tasks:

- Create a data control for your session bean, as described in Section 2.2.1, "How to Create ADF Data Controls."
- Create or identify a method on your session bean that will return the items needed to be displayed in your form. For example, the `OrderSessionEJBBean.java` class contains the `mergeOrd(Ord)` method.

To create an edit form:

1. From the Data Controls panel, drag the collection for which you wish to create the form, and choose **ADF Form** from the context menu.

This creates a form using `inputText` components, which will allow the user to edit the data in the fields.

2. From the Data Controls panel, select the merge method associated with the collection of objects on which you wish the operation to execute, and drag it onto the JSF page.

For example, if you want to be able to update an order record and will not be working with that instance again, you would drag the `mergeOrd(Ord)` method.

3. From the ensuing context menu, choose either **ADF Button** or **ADF Link**.
4. In the Edit Action Binding dialog, you need to populate the value for the method's parameter. For the merge methods (and the other default methods), this is the object being updated.
 - a. In the **Parameters** section, use the **Value** dropdown list to select **Show EL Expression Builder**.
 - b. In the Expression Builder, expand the node for the accessor's iterator, then expand the **currentRow** node, and select **dataProvider**.

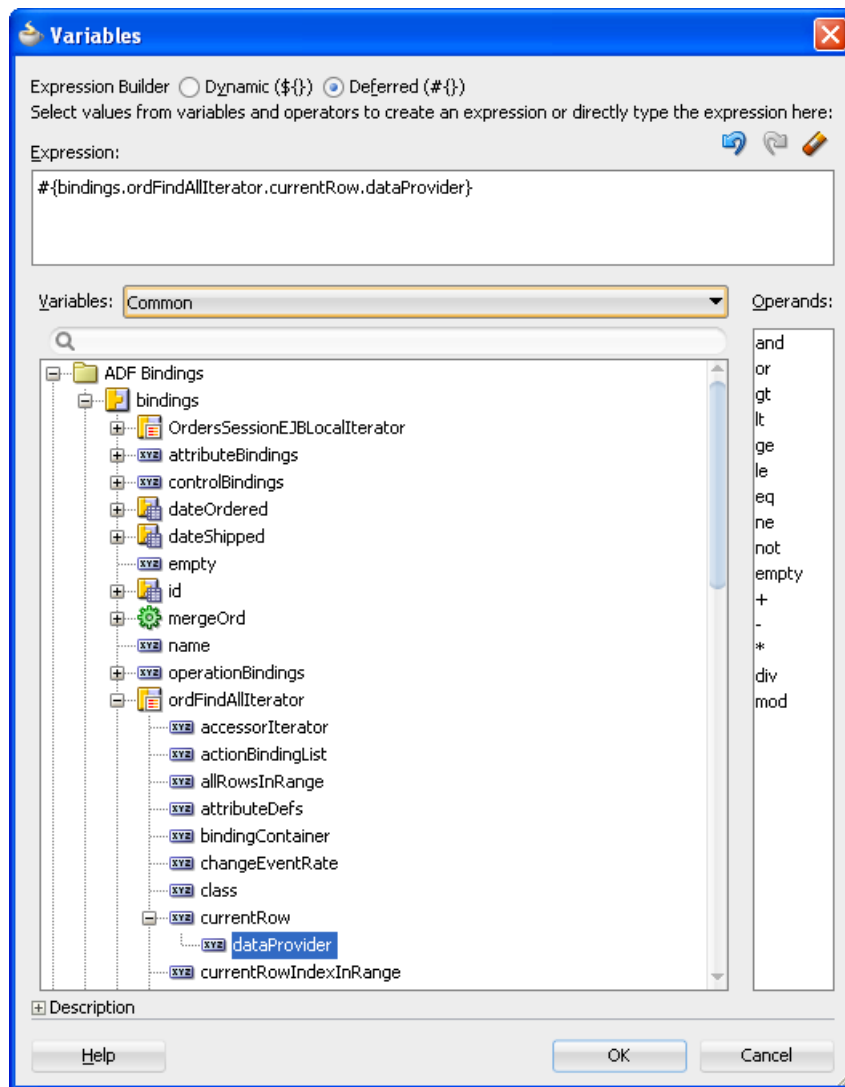
This will create an EL expression that evaluates to the data for the current row in the accessor's iterator.

c. Click **OK**.

For example, if you created a form using the `ordFindAll` accessor collection, then JDeveloper would have created an `accessorIterator` binding named `ordFindAllIterator`. You would need to select the `dataProvider` for the current row under that iterator, as shown in Figure 4-7. This reference means that the parameter value will resolve to the value of the row currently being shown in the form.

5. In the Edit Action Binding dialog, click **OK**.

Figure 4-7 *dataProvider for the Current Row on the ordFindAllIterator Binding*



Note: If the page is part of a transaction within a bounded task flow, then instead of creating a button from the merge method (or other default method), you would set that method as the value for the transaction resolution when creating the task flow return activity. For more information, see the "Using Task Flow Return Activities" section of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

4.6.2 What Happens When You Use Methods to Change Data

When you drop a method as a command button, JDeveloper:

- Defines a method binding for the method. If the method takes any parameters, JDeveloper creates `NamedData` elements that hold the parameter values. For more information about `NamedData` elements, see Section 4.5.3, "What Happens at Runtime: Setting Parameters for a Method."
- Inserts code in the JSF page for the ADF Faces command component. This code is similar to the code for any other command button. However, instead of being bound to the `execute` method of an action binding for an operation, the buttons are bound to the `execute` method of the action binding for the method that was dropped.

4.6.2.1 Method Bindings

Similar to when you create a button from a built-in operation, when you create a button from a method, JDeveloper creates an action binding for the method. Example 4–10 shows the action binding created when you drop the `mergeOrd (Ord)` method.

Example 4–10 Page Definition Code for an Action Binding Used by the Iterator

```
<bindings>
  <methodAction id="mergeOrd" RequiresUpdateModel="true"
    Action="invokeMethod" MethodName="mergeOrd"
    IsViewObjectMethod="false" DataControl="OrdersSessionEJBLocal "
    InstanceName="OrdersSessionEJBLocal.dataProvider"
    ReturnName="OrdersSessionEJBLocal.methodResults.mergeOrd_
      OrdersSessionEJBLocal_dataProvider_persistOrd_result">
    <NamedData NDName="ord"
      NDValue="#{bindings.ordFindAllIterator.currentRow.dataProvider}"
      NDType="model.Ord/>
  </methodAction>
</bindings>
```

In this example, when the binding is accessed, the method is invoked because the action property value is `invokeMethod`.

When you drop a method that takes parameters onto a JSF page, JDeveloper also creates `NamedData` elements for each parameter. These elements represent the parameters of the method. For example, the `mergeOrd (Ord)` method action binding contains a `NamedData` element for the `Ord` parameter.

4.6.2.2 Using EL Expressions to Bind to Methods

Like creating command buttons using navigation operations, when you create a command button using a method, JDeveloper binds the button to the method using the `actionListener` attribute. The button is bound to the `execute` property of the

action binding for the given method. This binding causes the binding's method to be invoked on the business service. For more information about the `actionListener` attribute, see the "What Happens at Runtime: How Action Events and Action Listeners Work" section of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

Tip: Instead of binding a button to the `execute` method on the action binding, you can bind the button to a method in a backing bean that overrides the `execute` method. Doing so allows you to add logic before or after the original method runs. For more information, see Section 4.6.4, "What You May Need to Know About Overriding Declarative Methods."

Like navigation operations, the `disabled` property on the button uses an EL expression to determine whether or not to display the button. Example 4–11 shows the EL expression used to bind the command button to the `mergeOrd(Ord)` method.

Example 4–11 JSF Code to Bind a Command Button to a Method

```
<af:commandButton actionListener="#{bindings.mergeOrd.execute}"
    text="mergeOrd"
    disabled="#{!bindings.mergeOrd.enabled}"
    id="cb1"/>
```

Tip: When you drop a UI component onto a page, JDeveloper automatically gives it an ID based on the number of the same type of component previously dropped, for example, `cb1`, `cb2`. You may want to change the ID to something more descriptive, especially if you will need to refer to it in a backing bean that contains methods for multiple UI components on the page.

4.6.3 What You May Need to Know About the Difference Between the Merge and Persist Methods

If when you created your session bean, you chose to expose the merge and persist methods for a structured object, 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.

4.6.4 What You May Need to Know About Overriding Declarative Methods

When you drop an operation or method as a command button, JDeveloper binds the button to the `execute` method for the operation or method. However, there may be occasions when you need to add logic before or after the existing logic. JDeveloper

allows you to add logic to a declarative operation by creating a new method and property on a managed bean that provides access to the binding container. By default, this generated code executes the operation or method. You can then add logic before or after this code. JDeveloper automatically binds the command component to this new method, instead of to the execute property on the original operation or method. Now when the user clicks the button, the new method is executed. For more information, see the "Overriding Declarative Methods" section in the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

4.7 Creating an Input Form

You can create a form that allows a user to enter information for a new record and then commit that record to the data source. You need to use a task flow that contains a method activity that will call the `Create` operation before the page with the input form is displayed. This method activity causes a blank row to be inserted into the row set which the user can then populate using a form.

Tip: For more information about task flows, see the "Creating ADF Task Flows" part of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

For example, if you have an `Address` bean, you could create a new form that allows the user to create an address. You might create a `create-address-task-flow` task flow that contains a `createAddress` method activity, which calls the `Create` operation on the `AddressFindAll` accessor. Control is then passed to the `createAddress` view activity, which displays a form where the user can enter a new address, as shown in Figure 4–8.

Figure 4–8 Create an Address

Note: If your application does not use task flows, then the calling page should invoke the `create` operation similar to the way in which a task flow's method activity would. For example, you could provide application logic within an event handler associated with a command button on the calling page.

4.7.1 How to Create an Input Form Using a Task Flow

You create an input form within a bounded task flow to ensure proper transaction handling.

Before you begin:

It may be helpful to have a general understanding of input forms. For more information, see Section 4.7, "Creating an Input Form."

You will need to complete these tasks:

- Create a data control for your session bean, as described in Section 2.2.1, "How to Create ADF Data Controls."
- Create a bounded task flow that will contain both the form and the method activity that will execute the `Create` operation. The task flow should start a new transaction. For procedures, see the "Creating a Task Flow" section of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

To create an input form:

1. To the bounded task flow, add a method activity. Have this activity execute the `Create` operation associated with the accessor for which you are creating the form. For these procedures on using method activities, see the "Using Method Call Activities" section of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

For example, to create the form that would allow users to create addresses, you would have the method activity execute the `Create` operation associated with the `AddressFindAll` accessor.

2. In the Property Inspector, enter a string for the **fixed-outcome** property. For example, you might enter `create` as the `fixed-outcome` value.
3. Add a view activity that represents the page for the input form. For information about adding view activities, see the "Using View Activities" section of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.
4. Add a control flow case from the method activity to the view activity. In the Property Inspector, enter the value of the **fixed-outcome** property of the method activity set in Step 2 as the value of the `from-outcome` of the control flow case.
5. Open the page for the view activity in the design editor, and from the Data Controls panel, drag the collection for which the form will be used to create a new record, and choose **ADF Form** from the context menu.

For example, for the form to create addresses, you would drag the `AddressFindAll` accessor collection from the Data Controls panel.

Tip: If you want the user to be able to create multiple entries before committing to the database, do the following:

1. In the task flow, add another control flow case from the view activity back to the method activity, and enter a value for the `from-outcome` method. For example, you might enter `createAnother`.
 2. Drag and drop a command component from the Component Palette onto the page, and set the `action` attribute to the `from-outcome` just created. This will cause the task flow to return to the method activity and reinvoke the `Create` operation.
6. In the task flow, add a return activity. This return activity must execute the `commit` operation on the data control. For these procedures, see the "Using Task Flow Return Activities" section of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

Tip: If when you set the return activity to execute the `commit` operation the activity shows an error, it is probably because the task flow itself is not set up to start a transaction. You need to set it to do so. For more information, see the "Managing Transactions in Task Flows" section of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

7. Add a control flow case from the view activity to the return activity. Set the `fixed-outcome` attribute to a text string. For example, you might set it to `return`.
8. From the Component Palette, drag and drop a **button** or other command component that will be used to invoke the return activity. Set the `action` attribute to the text string set as the `fixed-outcome` created in Step 7.

4.7.2 What Happens When You Create an Input Form Using a Task Flow

When you use an ADF Form to create an input form, JDeveloper:

- Creates an iterator binding for the accessor and an action binding for the `Create` operation in the page definition for the method activity. The `Create` operation is responsible for creating a row in the row set and populating the data source with the entered data. In the page definition for the page, JDeveloper creates an iterator binding for the returned collection and attribute bindings for each of the attributes of the object in the collection, as for any other form.
- Inserts code in the JSF page for the form using ADF Faces `inputText` components, and in the case of the operations, `commandButton` components.

For example, the form shown in Figure 4–8 might be displayed by clicking a "Create Address" link on the main page. This link then navigates to the form where you can input data for a new address. Once the address is created, and you click the **Save** button, you return to the main page. Figure 4–9 shows a `create-address-task-flow` task flow with the `newAddress` method activity.

Figure 4–9 Task Flow for an Input Form



Example 4–12 shows the code in the page definition file for the method activity.

Example 4–12 Page Definition Code for a Creation Method Activity

```

<executables>
  <iterator Binds="root" RangeSize="25" DataControl="SupplierFacadeLocal"
    id="SupplierFacadeLocalIterator" />
  <accessorIterator MasterBinding="SupplierFacadeLocalIterator"
    Binds="addresses" RangeSize="25"
    DataControl="SupplierFacadeLocal"
    BeanClass="oracle.fodemo.supplier.model.Addresses"
    id="addressesIterator" />

```

```
</executables>
<bindings>
  <action IterBinding="addressesIterator" id="Create"
    RequiresUpdateModel="true" Action="createRow" />
</bindings>
```

4.7.3 What Happens at Runtime: Invoking the Create Action Binding from the Method Activity

In the example above, when the `newAddress` method activity is accessed, the `Create` action binding is invoked, which executes the `createRow` operation, and a new blank instance for the collection is created. Note that during routing from the method activity to the view activity, the method activity's binding container skips validation for required attributes, allowing the blank instance to be displayed in the form on the page.

4.8 Modifying the UI Components and Bindings on a Form

Once you use the Data Controls panel to create any type of form, you can then delete attributes, change the order in which they are displayed, change the component used to display data, and change the attribute to which the components are bound.

For more information about modifying existing UI components and bindings, see the "Modifying the UI Components and Bindings on a Form" section of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

Creating ADF Databound Tables

This chapter describes how to use the Data Controls panel to create basic databound tables that are based on ADF Faces components, including editable tables and input tables.

This chapter includes the following sections:

- Section 5.1, "About Adding Tables"
- Section 5.2, "Creating a Basic Table"
- Section 5.3, "Creating an Editable Table"
- Section 5.4, "Creating an Input Table"
- Section 5.5, "Modifying the Attributes Displayed in the Table"

5.1 About Adding Tables

Unlike forms, tables allow you to display more than one record from a collection returned by an accessor at a time. Figure 5–1 shows a table that was created by dragging the Summit demo application's `CustomerFindAll` collection and then selecting four columns for display.

Figure 5–1 Results Table Displays Products That Match the Search Criteria

id	name	country	creditRating
201	Unisports	Brazil	EXCELLENT
202	Simms Athletics	Japan	POOR
203	Delhi Sports	India	GOOD
204	Womansport	USA	EXCELLENT

You can create tables that simply display data, or you can create tables that allow you to edit or create data. Once you drop an accessor as a table, you can add command buttons bound to actions that execute some logic on a selected row. You can also modify the default components to suit your needs.

5.2 Creating a Basic Table

Unlike with forms where you bind the individual UI components that make up a form to the individual attributes on the collection, with a table you bind the ADF Faces `table` component to the complete collection or to a range of n data objects at a time from the collection. The individual components used to display the data in the columns are then bound to the attributes. The iterator binding handles displaying the

correct data for each object, while the `table` component handles displaying each object in a row. JDeveloper allows you to do this declaratively, so that you don't need to write any code.

5.2.1 How to Create a Basic Table

To create a table using a data control, you bind the `table` component to a returned collection. JDeveloper allows you to do this declaratively by dragging and dropping a collection from the Data Controls panel.

Tip: You can also create a table by dragging a table component from the Component Palette and completing the Create ADF Faces Table wizard. For more information, see the "How to Display a Table on a Page" section of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

Before you begin:

It may be helpful to have a general understanding of using data control attributes to create tables. For more information, see Section 5.2, "Creating a Basic Table."

You will need to complete this task:

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

To create a databound table:

1. From the Data Controls panel, select a collection.

For example, to create a simple table in the Summit demo application that displays orders in the system, you would select the `ordFindAll` accessor collection.

2. Drag the collection onto a JSF page, and from the context menu, choose the appropriate table.

When you drag the collection, you can choose from the following types of tables:

- **ADF Table:** Allows you to select the specific attributes you need your editable table columns to display, and what UI components to use to display the data. By default, ADF `inputText` components are used for most attributes, thus enabling the table to be editable. Attributes that are dates use the `inputDate` component. Additionally, if a UI hint has been created for an attribute, or if the attribute has been configured to be a list, then the component set by the hint is used instead. For more information about setting UI hints, see the Section 3.3.5, "How to Set UI Hints on Attributes."
 - **ADF Read-Only Table:** Same as the ADF Table; however, each attribute is displayed in an `outputText` component.
 - **ADF Read-Only Dynamic Table:** Allows you to create a table when the attributes returned and displayed are determined dynamically at runtime. This component is helpful when the attributes for the corresponding object are not known until runtime, or you do not wish to hardcode the column names in the JSF page.
3. The ensuing Edit Table Columns dialog shows each attribute in the collection, and allows you to determine how these attributes will behave and appear as columns in your table.

Note: If the collection contains a structured attribute (an attribute that is neither a Java primitive type nor a collection), the attributes of the structured attributes will also appear in the dialog.

Using this dialog, you can do the following:

- In the **Row Selection** panel, determine how the ADF Model layer handles selection. Selecting the **Single** radio button means that the user can select one row and the iterator binding will access the iterator to determine the selected row. Selecting the **Multiple** radio button means that the user can select multiple rows and the iterator binding will access the iterator to determine those selected rows. Select **None** if you do not want the table to allow selection.
- Allow the ADF Model layer to handle column sorting by selecting the **Sorting** checkbox. Selecting this option means that the iterator binding will access the iterator, which will perform an order-by query to determine the order. Select this option unless you do not want to allow column sorting.
- Allow the columns in the table to be filtered using entered criteria by selecting the **Filtering** checkbox. Selecting this option allows the user to enter criteria in text fields above each column. That criteria is then used to build a Query-by-Example (QBE) search on the collection, so that the table will display only the results returned by the query. For more information, see Section 8.5, "Creating Standalone Filtered Search Tables."
- Group columns for selected attributes together under a parent column, by selecting the desired attributes (shown as rows in the dialog) and clicking the **Group** button. Figure 5–2 shows how two grouped columns appear in the visual editor after the table is created.

Figure 5–2 Grouped Columns in an ADF Faces Table

Group		Cost Price	List Price
Product Id	Product Name		
#{...productId}	#{...productName}	#{...costPrice}	#{...listPrice}
#{...productId}	#{...productName}	#{...costPrice}	#{...listPrice}
#{...productId}	#{...productName}	#{...costPrice}	#{...listPrice}

- Change the display label for a column by entering text or an EL expression to bind the label value to something else, such as a key in a resource file. By default, the label is bound to the `label` property for any UI hint defined for the attribute on the table binding. This binding allows you to change the value of a label text one time in the data control structure file, and have the change propagate to all pages that display the label.
- Change the value binding for a column by selecting a different attribute to bind to. If you simply want to rearrange the columns, you should use the order buttons. If you do change the attribute binding for a column, the label for the column also changes.
- Change the UI component used to display an attribute using the dropdown menu. The UI components are set based on the table you selected when you dropped the collection onto the page, on the type of the corresponding attribute (for example, `inputDate` components are used for attributes that are dates), and on whether or not default components were set as UI hints in the bean's data control structure file.

Tip: If one of the attributes for your table is also a primary key, you may want to choose a UI component that will not allow a user to change the value.

Tip: If you want to use a component that is not listed in the dropdown menu, use this dialog to select the `outputText` component, and then manually add the other tag to the page.

- Change the order of the columns using the order buttons.
 - Add a column using the **Add** icon. There's no limit to the number of columns you can add. When you first click the icon, JDeveloper adds a new column line at the bottom of the dialog and populates it with the values from the first attribute in the bound collection; subsequent new columns are populated with values from the next attribute in the sequence, and so on.
 - Delete a column using the **Delete** icon.
4. Once the table is dropped on the page, you can use the Property Inspector to set other display properties of the table. For example, you may want to set the width of the table to a certain percentage or size. For more information about display properties, see the "Using Tables and Trees" chapter of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

Tip: When you set the table width to 100%, the table will not include borders, so the actual width of the table will be larger. To have the table set to 100% of the container width, expand the **Style** section of the Property Inspector, select the **Box** tab, and set the `Border Width` attribute to 0 pixels.

5. If you want the user to be able to edit information in the table and save any changes, you need to provide a way to submit and persist those changes. For more information, see Section 5.3, "Creating an Editable Table." For procedures on creating tables that allow users to input data, see Section 5.4, "Creating an Input Table."

5.2.2 What Happens When You Create a Table

Dropping a table from the Data Controls panel has the same effect as dropping a text field or form. Briefly, JDeveloper does the following:

- Creates the bindings for the table and adds the bindings to the page definition file
- Adds the necessary code for the UI components to the JSF page

For more information, see Section 4.2.2, "What Happens When You Create a Text Field."

5.2.2.1 Iterator and Value Bindings for Tables

When you drop a table from the Data Controls panel, a tree value binding is created. A tree consists of a hierarchy of nodes, where each subnode is a branch off a higher level node. In the case of a table, it is a flattened hierarchy, where each attribute (column) is a subnode off the table. Like an attribute binding used in forms, the tree value binding references the accessor iterator binding, while the accessor iterator binding references the iterator for the data control, which facilitates iterating over the data objects in the collection. Instead of creating a separate binding for each attribute, only the tree binding to the table node is created. In the tree binding, the `AttrNames` element

within the `nodeDefinition` element contains a child element for each attribute that you want to be available for display or reference in each row of the table.

The tree value binding is an instance of the `FacesCtrlHierBinding` class that extends the core `JUCtrlHierBinding` class to add two JSF specific properties:

- `collectionModel`: Returns the data wrapped by an object that extends the `javax.faces.model.DataModel` object that JSF and ADF Faces use for collection-valued components like tables.
- `treeModel`: Extends `collectionModel` to return data that is hierarchical in nature. For more information, see Chapter 6, "Displaying Master-Detail Data."

Example 5-1 shows the value binding for the table created when you drop the `SOrdList` accessor collection.

Example 5-1 Value Binding Entries for a Table in the Page Definition File

```
<bindings>
  <tree IterBinding="SOrdListIterator" id="SOrdList">
    <nodeDefinition DefName="model.Ord" Name="SOrdList0">
      <AttrNames>
        <Item Value="id"/>
        <Item Value="dateOrdered"/>
        <Item Value="paymentTypeId"/>
        <Item Value="salesRepId"/>
        <Item Value="total"/>
        <Item Value="orderFilled"/>
        <Item Value="dateShipped"/>
      </AttrNames>
    </nodeDefinition>
  </tree>
  ...
</bindings>
```

Only the table component needs to be bound to the model (as opposed to the columns or the text components within the individual cells), because only the table needs access to the data. The tree binding for the table drills down to the individual structure attributes in the table, and the table columns can then derive their information from the table component.

5.2.2.2 Code on the JSF Page for an ADF Faces Table

When you use the Data Controls panel to drop a table onto a JSF page, JDeveloper inserts an ADF Faces `table` component, which contains an ADF Faces `column` component for each attribute named in the table binding. Each column then contains another component (such as an `inputText` or `outputText` component) bound to the attribute's value. Each column's heading is bound to the `labels` property for the control hint of the attribute.

Tip: If an attribute is marked as hidden in the associated structure file, no corresponding UI is created for it.

Example 5-2 shows a code excerpt from a table created by dropping the `ordFindAll` accessor collection as a read-only table.

Example 5-2 JSF Code for an ADF Faces Read-Only Table

```
<af:table value="#{bindings.ordFindAll.collectionModel}" var="row"
  rows="#{bindings.ordFindAll.rangeSize}"
```

```

        emptyText="#{bindings.ordFindAll.viewable ? 'No data to display.' :
            'Access Denied.'}"
        fetchSize="#{bindings.ordFindAll.rangeSize}" rowBandingInterval="0"
        id="t1">
<af:column sortProperty="#{bindings.ordFindAll.hints.dateOrdered.name}"
    sortable="false"
    headerText="#{bindings.ordFindAll.hints.dateOrdered.label}"
    id="c1">
<af:outputText value="#{row.dateOrdered}" id="ot1">
    <af:convertDateTime
        pattern="#{bindings.ordFindAll.hints.dateOrdered.format}"/>
</af:outputText>
</af:column>
<af:column sortProperty="#{bindings.ordFindAll.hints.dateShipped.name}"
    sortable="false"
    headerText="#{bindings.ordFindAll.hints.dateShipped.label}"
    id="c2">
<af:outputText value="#{row.dateShipped}" id="ot2">
    <af:convertDateTime
        pattern="#{bindings.ordFindAll.hints.dateShipped.format}"/>
</af:outputText>
</af:column>
. . .
</af:table>

```

The tree binding iterates over the data exposed by the iterator binding. Note that the table's value is bound to the `collectionModel` property, which accesses the `collectionModel` object. The table wraps the result set from the iterator binding in a `collectionModel` object. The `collectionModel` allows each item in the collection to be available within the table component using the `var` attribute.

In the example, the table iterates over the rows in the current range of the `ordFindAll` accessor binding. This binding binds to a row set iterator that keeps track of the current row. When you set the `var` attribute on the table to `row`, each column then accesses the current data object for the current row presented to the table tag using the `row` variable, as shown for the value of the `af:outputText` tag:

```
<af:outputText value="#{row.dateOrdered}"/>
```

When you drop an ADF Table (as opposed to an ADF Read-Only Table), instead of being bound to the row variable, the value of the input component is implicitly bound to a specific row in the binding container through the `bindings` property, as shown in Example 5-3. Additionally, JDeveloper adds validator and converter components for each input component. By using the `bindings` property, any raised exception can be linked to the corresponding binding object or objects. The controller iterates through all exceptions in the binding container and retrieves the binding object to get the client ID when creating `FacesMessage` objects. This retrieval allows the table to display errors for specific cells. This strategy is used for all input components, including selection components such as lists.

Example 5-3 Using Input Components Adds Validators and Converters

```

<af:table value="#{bindings.ordFindAll.collectionModel}" var="row"
    rows="#{bindings.ordFindAll.rangeSize}"
    emptyText="#{bindings.ordFindAll.viewable ? 'No data to display.' :
        'Access Denied.'}"
    fetchSize="#{bindings.ordFindAll.rangeSize}" rowBandingInterval="0"
    id="t1">
<af:column sortProperty="#{bindings.ordFindAll.hints.dateOrdered.name}"
    sortable="false"

```

```

        headerText="#{bindings.ordFindAll.hints.dateOrdered.label}"
        id="c1">
<af:inputDate value="#{row.bindings.dateOrdered.inputValue}"
        label="#{bindings.ordFindAll.hints.dateOrdered.label}"
        required="#{bindings.ordFindAll.hints.dateOrdered.mandatory}"
        columns="#{bindings.ordFindAll.hints.dateOrdered.displayWidth}"
        shortDesc="#{bindings.ordFindAll.hints.dateOrdered.tooltip}"
        id="id1">
        <f:validator binding="#{row.bindings.dateOrdered.validator}"/>
        <af:convertDateTime
            pattern="#{bindings.ordFindAll.hints.dateOrdered.format}"/>
    </af:inputDate>
</af:column>
<af:column sortProperty="#{bindings.ordFindAll.hints.dateShipped.name}"
        sortable="false"
        headerText="#{bindings.ordFindAll.hints.dateShipped.label}"
        id="c2">
    <af:inputDate value="#{row.bindings.dateShipped.inputValue}"
        label="#{bindings.ordFindAll.hints.dateShipped.label}"
        required="#{bindings.ordFindAll.hints.dateShipped.mandatory}"
        columns="#{bindings.ordFindAll.hints.dateShipped.displayWidth}"
        shortDesc="#{bindings.ordFindAll.hints.dateShipped.tooltip}"
        id="id2">
        <f:validator binding="#{row.bindings.dateShipped.validator}"/>
        <af:convertDateTime
            pattern="#{bindings.ordFindAll.hints.dateShipped.format}"/>
    </af:inputDate>
</af:column>
<af:column sortProperty="#{bindings.ordFindAll.hints.id.name}"
        sortable="false"
        headerText="#{bindings.ordFindAll.hints.id.label}" id="c3">
    <af:inputText value="#{row.bindings.id.inputValue}"
        label="#{bindings.ordFindAll.hints.id.label}"
        required="#{bindings.ordFindAll.hints.id.mandatory}"
        columns="#{bindings.ordFindAll.hints.id.displayWidth}"
        maxLength="#{bindings.ordFindAll.hints.id.precision}"
        shortDesc="#{bindings.ordFindAll.hints.id.tooltip}"
        id="it1">
        <f:validator binding="#{row.bindings.id.validator}"/>
    </af:inputText>
</af:column>
    . . .
</af:table>

```

For more information about using ADF Faces validators and converters, see the "Validating and Converting Input" chapter of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

Table 5–1 shows the other attributes defined by default for ADF Faces tables created using the Data Controls panel.

Table 5–1 ADF Faces Table Attributes and Populated Values

Attribute	Description	Default Value
rows	Determines how many rows to display at one time.	An EL expression that, by default, evaluates to the <code>rangeSize</code> property of the associated iterator binding, which determines how many rows of data are fetched from a data control at one time. Note that the value of the <code>rows</code> attribute must be equal to or less than the corresponding iterator's <code>rangeSize</code> value, as the table cannot display more rows than are returned. For more information about the <code>rangeSize</code> property, see Section 4.4.2.2, "Iterator RangeSize Attribute."
emptyText	Text to display when there are no rows to return.	An EL expression that evaluates to the <code>viewable</code> property on the iterator. If the table is viewable, the attribute displays No data to display when no objects are returned. If the table is not viewable (for example, if there are authorization restrictions set against the table), it displays Access Denied .
fetchSize	Number of rows of data fetched from the data source.	An EL expression that, by default, evaluates to the <code>rangeSize</code> property of the associated iterator binding. For more information about the <code>rangeSize</code> property, see Section 4.4.2.2, "Iterator RangeSize Attribute." This attribute can be set to a larger number than the <code>rows</code> attribute.
selectedRowKeys	The selection state for the table.	An EL expression that, by default, evaluates to the selected row on the collection model.
selectionListener	Reference to a method that listens for a selection event.	An EL expression that, by default, evaluates to the <code>makeCurrent</code> method on the collection model.
rowSelection	Determines whether rows are selectable.	Set to <code>single</code> to allow one row to be selected at a time.
Column Attributes		
sortProperty	Determines the property by which to sort the column.	Set to the column's corresponding attribute binding value.
sortable	Determines whether a column can be sorted.	Set to <code>false</code> . When set to <code>true</code> , the iterator binding will access the iterator to determine the order.
headerText	Determines the text displayed at the top of the column.	An EL expression that, by default, evaluates to the <code>label</code> UI hint set on the corresponding attribute.

5.2.3 What You May Need to Know About Setting the Current Row in a Table

When you use tables in an application and you allow the ADF Model layer to manage row selection, the current row is determined by the iterator. When a user selects a row in an ADF Faces table, the row in the table is shaded, and the component notifies the iterator of the selected row. To do this, the `selectedRowKeys` attribute of the table is bound to the collection model's selected row, as shown in Example 5–4.

Example 5–4 Selection Attributes on a Table

```
<af:table value="#{bindings.ordFindAll.collectionModel}" var="row"
```

```

.
.
.
selectedRowKeys="#{bindings.ordFindAll.collectionModel.selectedRow}"
selectionListener="#{bindings.ordFindAll.collectionModel.makeCurrent}"
rowSelection="single" id="t1">

```

This binding binds the selected keys in the table to the selected row of the collection model. The `selectionListener` attribute is then bound to the collection model's `makeCurrent` property. This binding makes the selected row of the collection the current row of the iterator.

Note: If you create a custom selection listener, you must create a method binding to the `makeCurrent` property on the collection model (for example, `#{binding.Ord.collectionModel.makeCurrent}`) and invoke this method binding in the custom selection listener before any custom logic.

Although a table can handle selection automatically, there may be cases where you need to programmatically set the current row for an object on an iterator.

You can call the `getKey()` method on any view row to get a `Key` object that encapsulates the one or more key attributes that identify the row. You can also use a `Key` object to find a view row in a row set using the `findByKey()`. At runtime, when either the `setCurrentRowWithKey` or the `setCurrentRowWithKeyValue` built-in operation is invoked by name by the data binding layer, the `findByKey()` method is used to find the row based on the value passed in as a parameter before the found row is set as the current row.

The `setCurrentRowWithKey` and `setCurrentRowWithKeyValue` operations both expect a parameter named `rowKey`, but they differ precisely by what each expects that `rowKey` parameter value to be at runtime:

The `setCurrentRowWithKey` Operation

`setCurrentRowWithKey` expects the `rowKey` parameter value to be the *serialized string representation* of a view row key. This is a hexadecimal-encoded string that looks similar to the following string:

```
000200000002C20200000002C102000000010000010A5AB7DAD9
```

The serialized string representation of a key encodes all of the key attributes that might comprise a view row's key in a way that can be conveniently passed as a single value in a browser URL string or form parameter. At runtime, if you inadvertently pass a parameter value that is not a legal serialized string key, you may receive exceptions like `oracle.jbo.InvalidParamException` or `java.io.EOFException` as a result. In your web page, you can access the value of the serialized string key of a row by referencing the `rowKeyStr` property of an ADF control binding (for example, `#{bindings.SomeAttrName.rowKeyStr}`) or the row variable of an ADF Faces table (for example, `#{row.rowKeyStr}`).

`setCurrentRowWithKeyValue`

The `setCurrentRowWithKeyValue` operation expects the `rowKey` parameter value to be the literal value representing the key of the view row. For example, its value would be simply "201" to find order number 201.

5.3 Creating an Editable Table

You can create a table that allows the user to edit information within the table, and then commit those changes to the data source. To do this, you use operations that can modify data records associated with the returned collection (or the data control itself) to create command buttons, and place those buttons in a toolbar in the table. For example, the table in the `browse.jspx` page has a button that allows the user to remove a product. While this button currently causes a dialog to display that allows the user to confirm the removal, the button could be bound to a method that directly removes the product.

Tip: To create a table that allows you to insert a new record into the data store, see Section 5.4, "Creating an Input Table."

As with editable forms, it is important to note that the ADF Model layer is not aware that any row has been changed until a new instance of the collection is presented. Therefore, you need to invoke the `execute` operation on the accessor iterator in order for any changes to be committed. For more information, see Section 2.3.4, "What You May Need to Know About Iterator Result Caching."

When you decide to use editable components to display your data, you have the option of having the table displaying all rows as editable at once, or having it display all rows as read-only until the user double-clicks within the row. Figure 5–3 shows a table whose rows all have editable fields. The page is rendered using the components that were added to the page (for example, `inputText`, `inputDate`, and `inputNumberSpinbox` components).

Figure 5–3 Table with Editable Fields

No.	Name	Size	Date Modified	A Spinbox
0	.	0 B	7/12/2004	1979
1	..	0 B	7/12/2004	1979
2	admin.jar	1 KB	5/11/2004	1979
3	applib	0 B	7/12/2004	1979
4	applications	0 B	7/12/2004	1979
5	config	0 B	7/12/2004	1979
6	connectors	0 B	7/12/2004	1979

Figure 5–4 shows the same table, but configured so that the user must double-click (or single-click if the row is already selected) a row in order to edit or enter data. Note that `outputText` components are used to display the data in the nonselected rows, even though the same input components as in Figure 5–3 were used to build the page. The only row that actually renders those components is the row selected for editing.

Figure 5–4 Table with Clickable Row Selected

No.	Name	Size	Date Modified	A Spinbox
0	.	0 B	7/12/2004	1979
1	..	0 B	7/12/2004	1979
2	admin.jar	1 KB	5/11/2004	1979
3	applib	0 B	7/12/2004	1979
4	applications	0 B	7/12/2004	1979
5	config	0 B	7/12/2004	1979
6	connectors	0 B	7/12/2004	1979

For more information about how ADF Faces table components handle editing, see the "Editing Data in Tables, Trees, and Tree Tables" section of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

5.3.1 How to Create an Editable Table

To create an editable table, you follow procedures similar to those for creating a basic table, and then you add command buttons bound to operations. However, in order for the table to contain a toolbar, you need to add an ADF Faces component that associates the toolbar with the items in the collection used to build the table.

Before you begin:

It may be helpful to have a general understanding of editable tables. For more information, see Section 5.3, "Creating an Editable Table."

You will need to complete this task:

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

To create an editable table:

1. From the Data Controls panel, select the collection to display in the table.

For example, to create a simple table in the Summit demo application that will allow you to edit customers in the system, you would select the `customerFindAll` accessor collection.

2. Drag the accessor onto a JSF page, and from the context menu, choose **ADF Table**.
3. Use the ensuing Edit Table Columns dialog to determine how the attributes should behave and appear as columns in your table. Be sure to select the **Row Selection** checkbox, which will allow the user to select the row to edit.

For more information about using this dialog to configure the table, see Section 5.2, "Creating a Basic Table."

4. With the table selected in the Structure window, expand the **Behavior** section of the Property Inspector and set the **EditingMode** attribute. If you want all the rows to be editable select **editAll**. If you want the user to click into a row to make it editable, select **clickToEdit**.
5. From the Structure window, right-click the table component and select **Surround With** from the context menu.
6. In the Surround With dialog, ensure that **ADF Faces** is selected in the dropdown list, select the **Panel Collection** component, and click **OK**.

The `panelCollection` component's toolbar facet will hold the toolbar which, in turn, will hold the command components used to update the data.

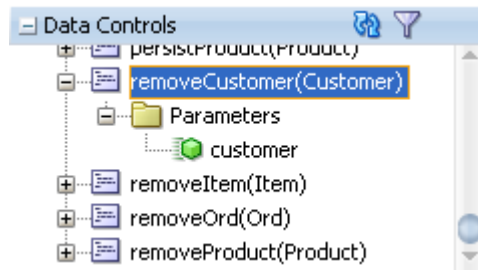
7. In the Structure window, right-click the `panelCollection`'s **toolbar** facet folder, and from the context menu, choose **Insert inside toolbar > Toolbar**.

This creates a toolbar that already contains a default menu which allows users to change how the table is displayed, and a **Detach** link that detaches the entire table and displays it such that it occupies the majority of the space in the browser window. For more information about the `panelCollection` component, see the "Displaying Table Menus, Toolbars, and Status Bars" section of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

8. From the Data Controls panel, select the method or operation associated with the collection of objects on which you wish to execute the logic, and drag it onto the `toolbar` component in the Structure window. This will place the databound command component inside the toolbar.

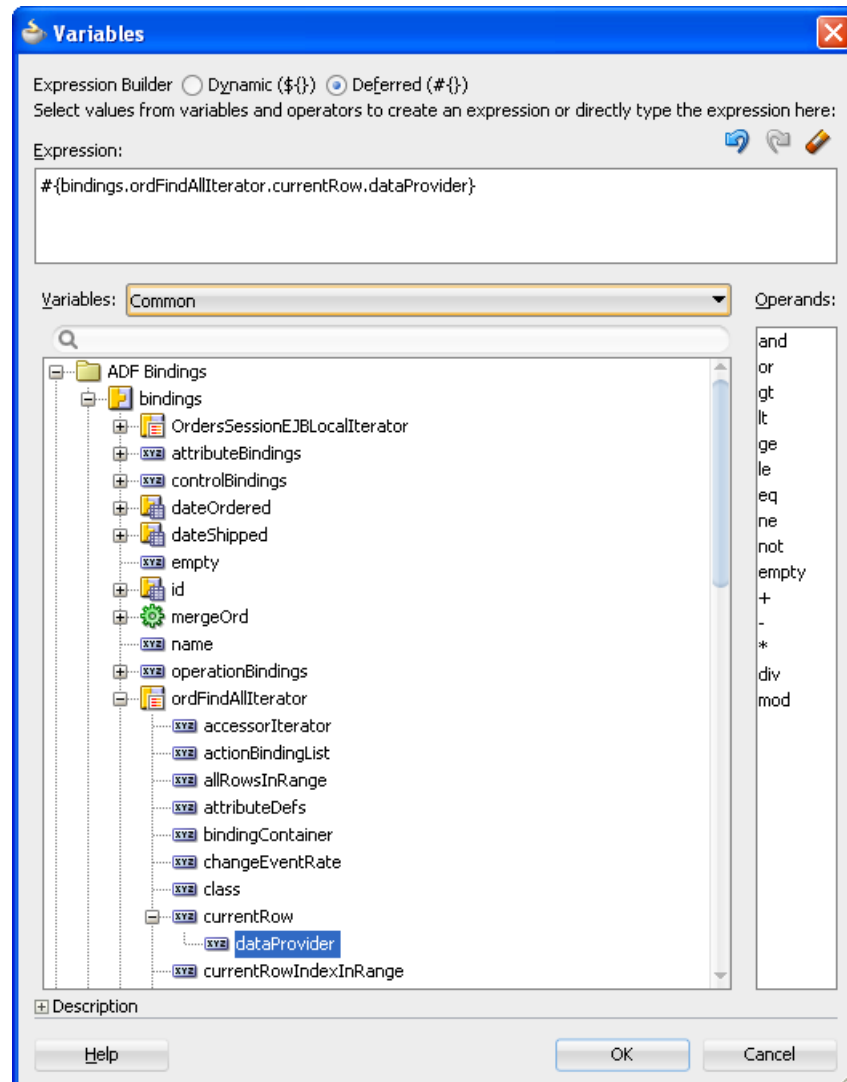
For example, if you want to be able to remove a customer record, you would drag the `removeCustomer(Customer)` method. Figure 5–5 shows the remove methods in the Summit demo application.

Figure 5–5 Remove Methods in a Data Control



9. For the context menu, choose **Operations > ADF Toolbar Button**.
Because the method takes parameters, the Action Binding Editor opens, asking you to set the value of the parameters.
10. In the Edit Action Binding dialog, you need to populate the value for the method's parameter. For the remove methods (and the other default methods), this is the selected object.
 - a. In the **Parameters** section, use the **Value** dropdown list to select **Show EL Expression Builder**.
 - b. In the Expression Builder, expand the node for the accessor's iterator, then expand the **currentRow** node, and select **dataProvider**.
This will create an EL expression that evaluates to the data for the current row in the accessor's iterator.
 - c. Click **OK**.

For example, if you created a table using the `ordFindAll` accessor, then JDeveloper would have created an `accessorIterator` binding named `ordFindAllIterator`. You would need to select the `dataProvider` object for the current row under that iterator, as shown in Figure 5–6. This reference means that the parameter value will resolve to the value of the currently selected row.

Figure 5–6 *dataProvider for the Current Row on the ordFindAllIterator Binding*

11. To notify the ADF Model layer that the collection has been modified, you need to also bind the toolbar button to a method that will refresh the iterator.

- a. Open the page definition for the JSPX file by right-clicking the file and choosing **Go to Page Definition**.
- b. In the Structure window for the page definition, right-click **bindings** and choose **Insert inside bindings > Generic Bindings > action**.
- c. In the Create Action Binding dialog, use the **Select an Iterator** dropdown list to select the iterator associated with the collection, and for **Operation**, select **Execute**.

JDeveloper creates an action binding for the `execute` operation of the iterator. You now need to have your command button call this operation.

12. In the JSF page, select the command component created when you dropped the method in Step 8. In the Property Inspector, set **Action** to the following:

```
{bindings.Execute.execute}
```

When the command component is clicked, the binding to the `action` attribute is evaluated after the binding for the `actionListener` attribute. This order ensures that the iterator refreshes and/or executes after the deletion of entity.

5.3.2 What Happens When You Create an Editable Table

Creating an editable table is similar to creating a form used to edit records. Action bindings are created for the operations dropped from the Data Controls panel. For more information, see Section 4.6.2, "What Happens When You Use Methods to Change Data."

5.4 Creating an Input Table

You can create a table that allows users to insert a new blank row into a table and then add values for each column (any default values set on the corresponding data control structure file will be automatically populated).

5.4.1 How to Create an Input Table

When you create an input table, you want the user to see the new blank row in the context of the other rows within the current row set. To allow this insertion, you need to use the `create` operation associated with the accessor for the collection that you used to create the table.

Because the `create` operation only creates a row in the cache, you also need to add a button that actually merges the newly created row into the collection. Figure 5-7 shows how this table might look with a new row created.

Figure 5–7 Input Table With Buttons for Creating Rows and Saving Data

Supplier ID	Supplier Name	Phone Number	Email	Status
100	Stuffz	320.555	contact@stuffz.exe	ACTIVE
101	Nexus	212.555	contact@nexus.exe	ACTIVE
102	Gifts-N-More	212.555	contact@giftsnmore	ACTIVE
103	Emporium	212.555	contact@emporium.	ACTIVE
104	Jeffery And Michae	419.555	contact@jeffery-mi	ACTIVE
105	Games Galore	630.555	contact@games_ga	ACTIVE
106	Transistor City	303.555	contact@transistor	ACTIVE
107	Mercury Imports	862.555	contact@mercury-ir	ACTIVE
108	BigSwamp	248.555	contact@bigswamp	ACTIVE
109	Z-Mart	959.555	contact@zmart.exe	ACTIVE
110	Wally's Mart	470.555	contact@wallymart.	ACTIVE
111	Ellis	641.555	contact@ellis.examj	ACTIVE
112	Electronics and Mo	413.555	contact@electronic	ACTIVE
113	Zibbers	717.555	contact@zibbers.ex	ACTIVE
114	Great Gadgets	320.555	contact@greatgadg	ACTIVE
115	Perfect Purchase	971.555	contact@perfect_pi	ACTIVE
116	Silicon Super Store	829.555	contact@siliconsupe	ACTIVE
117	Value Valley	325.555	contact@valuevalle	ACTIVE
118	Super Stuff	661.555	contact@superstuff	ACTIVE
119	Gifts-to-Go	308.555	contact@giftstogo.	ACTIVE
120	MoreStuffz	662.555	contact@morestuff.	ACTIVE
121	MoreAndMoreStuff	614.555	contact@moreandr	ACTIVE
122	MaxStuffz	570.555	contact@maxstuffz	ACTIVE
123	Geeky Gadgets	256.555	contact@geekygadi	ACTIVE
124	Gobs of Goods	647.555	contact@gobsofgoc	ACTIVE
125	Stock Plus	563.555	contact@stockplus.	ACTIVE
126	Zip City	985.555	contact@zipcity.exe	ACTIVE
127	Shop-a-lot	269.555	contact@shopalot.e	ACTIVE
128	Shopster	706.555	contact@shopster.e	ACTIVE
129	Simply Shop	848.555	contact@simplyshor	ACTIVE
200	BuyMyJunk	980.555	contact@buymyjunk	ACTIVE

ADF Faces components can be set so that one component refreshes based on an interaction with another component, without the whole page needing to be refreshed. This is known as *partial page rendering*. When the user clicks the button to create the new row, you want the table to refresh to display that new row. To have that happen, you need to configure the table to respond to that user action.

Before you begin:

It may be helpful to have a general understanding of input tables. For more information, see Section 5.4, "Creating an Input Table."

You will need to complete these tasks:

- Create a data control for your session bean, as described in Section 2.2.1, "How to Create ADF Data Controls."
- Create an editable table, as described in Section 5.3, "Creating an Editable Table."

To create an input table:

1. From the Data Controls panel, drag the **Create** operation associated with the dropped collection and drop it as a toolbar button into the toolbar.

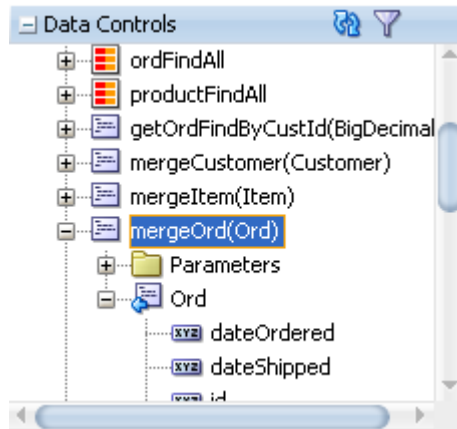
Tip: You may want to change the ID to something more recognizable, such as `Create`. This will make it easier to identify when you need to select it as the partial trigger.

2. In the Structure window, select the table component.
3. In the Property Inspector, expand the **Behavior** section, click the dropdown menu for the **PartialTriggers** attribute, and select **Edit**.
4. In the Edit Property dialog, expand the toolbar facet for the `panelCollection` component and then expand the toolbar that contains the **Create** command component. Select that component and shuttle it to the **Selected** panel. Click **OK**. This sets that component to be the trigger that will cause the table to refresh.
5. Create a button that allows the user to merge the new object(s) into the collection. From the Data Controls panel, drag the merge method associated with the collection used to create the table, and drop it as a toolbar button or link into the toolbar.

Tip: If you will want the user to be able to continue updating the row after it is persisted, then you should create the button using the `persist` method instead. For more information, see Section 4.6.3, "What You May Need to Know About the Difference Between the Merge and Persist Methods."

Figure 5–8 shows the merge method for the `Ord` collection.

Figure 5–8 Merge Methods in the Data Controls Panel



5.4.2 What Happens When You Create an Input Table

When you use the `create` operation to create an input table, JDeveloper:

- Creates an iterator binding for the collection, an action binding for the `create` operation, and attribute bindings for the table. The `create` operation is responsible for creating the new row in the row set. If you created command buttons or links using the merge method, JDeveloper also creates an action binding for that method.
- Inserts code in the JSF page for the table for the ADF Faces components.

Example 5–5 shows the page definition file for an input table.

Example 5–5 Page Definition Code for an Input Table

```
<executables>
  <variableIterator id="variables"/>
  <iterator Binds="root" RangeSize="25" DataControl="SessionEJBLocal"
```

```

        id="SessionEJBLocalIterator"/>
<accessorIterator MasterBinding="SessionEJBLocalIterator"
    Binds="suppliersFindAll" RangeSize="25"
    DataControl="SessionEJBLocal" BeanClass="model.Suppliers"
    id="suppliersFindAllIterator"/>
</executables>
<bindings>
    <action IterBinding="suppliersFindAllIterator" id="Create"
        RequiresUpdateModel="true" Action="createRow"/>
    <methodAction id="mergeSuppliers" RequiresUpdateModel="true"
        Action="invokeMethod" MethodName="mergeSuppliers"
        IsViewObjectMethod="false" DataControl="SessionEJBLocal"
        InstanceName="SessionEJBLocal.dataProvider"
        ReturnName="SessionEJBLocal.methodResults.mergeSuppliers_
            SessionEJBLocal_dataProvider_mergeSuppliers_result">
    <NamedData NDName="suppliers"
        NDValue="#{bindings.Create.currentRow.dataProvider}"
        NDType="model.Suppliers"/>
    </methodAction>
    <tree IterBinding="suppliersFindAllIterator" id="suppliersFindAll">
        <nodeDefinition DefName="model.Suppliers">
            <AttrNames>
                <Item Value="email"/>
                <Item Value="phoneNumber"/>
                <Item Value="supplierId"/>
                <Item Value="supplierName"/>
                <Item Value="supplierStatus"/>
            </AttrNames>
        </nodeDefinition>
    </tree>
</bindings>

```

Example 5–6 shows the code added to the JSF page that provides partial page rendering, using command toolbar buttons as the triggers to refresh the table.

Example 5–6 Partial Page Trigger Set on a Command Button for a Table

```

<af:form id="f1">
    <af:panelCollection id="pc1">
        <f:facet name="menus"/>
        <f:facet name="toolbar">
            <af:toolbar id="t2">
                <af:commandToolbarButton actionListener="#{bindings.Create.execute}"
                    text="Create New Supplier"
                    disabled="#{!bindings.Create.enabled}"
                    id="ctb1"/>
                <af:commandToolbarButton
                    actionListener="#{bindings.mergeSuppliers.execute}"
                    text="Commit New Suppliers"
                    disabled="#{!bindings.mergeSuppliers.enabled}"
                    id="ctb2"/>
            </af:toolbar>
        </f:facet>
        <f:facet name="statusbar"/>
    </af:panelCollection>
    <af:table value="#{bindings.suppliersFindAll.collectionModel}"
        var="row" rows="#{bindings.suppliersFindAll.rangeSize}"
        emptyText="#{bindings.suppliersFindAll.viewable ? 'No data to
            display.' : 'Access Denied.'}"
        fetchSize="#{bindings.suppliersFindAll.rangeSize}"
        rowBandingInterval="0"

```

```

        selectedRowKeys=
            "#{bindings.suppliersFindAll.collectionModel.selectedRow}"
        selectionListener=
            "#{bindings.suppliersFindAll.collectionModel.makeCurrent}"
        rowSelection="single" id="t1"
        partialTriggers="::ctb1 ::ctb2">
<af:column sortProperty="supplierId" sortable="false"
    headerText=
        "#{bindings.suppliersFindAll.hints.supplierId.label}"
    id="c6">
    <af:inputText value="#{row.bindings.supplierId.inputValue}"
        label="#{bindings.suppliersFindAll.hints.supplierId.label}"
        required="#{bindings.suppliersFindAll.hints.supplierId.mandatory}"
        columns="#{bindings.suppliersFindAll.hints.supplierId.displayWidth}"
        maximumLength="#{bindings.suppliersFindAll.hints.supplierId.precision}"
        shortDesc="#{bindings.suppliersFindAll.hints.supplierId.tooltip}"
        id="it4">
    <f:validator binding="#{row.bindings.supplierId.validator}"/>
    <af:convertNumber groupingUsed="false"
        pattern="#{bindings.suppliersFindAll.hints.supplierId.format}"/>
    </af:inputText>
</af:column>
.
.
.
    </af:table>
</af:panelCollection>
</af:form>

```

5.4.3 What Happens at Runtime: How Create and Partial Page Refresh Work

When the button bound to the `create` operation is invoked, the action executes, and a new instance for the collection is created as the page is rerendered. Because the button was configured to be a trigger that causes the table to refresh, the table redraws with the new empty row shown at the top. When the user clicks the button bound to the merge method, the newly created rows in the row set are inserted into the database. For more information about partial page refresh, see the "Rerendering Partial Page Content" chapter in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

5.4.4 What You May Need to Know About Creating a Row and Sorting Columns

If your table columns allow sorting, and the user has sorted on a column before inserting a new row, then that new row will not be sorted. To have the column sort with the new row, the user must first sort the column opposite to the desired sort, and then re-sort. This is because the table assumes the column is already sorted, so clicking on the desired sort order first will have no effect on the column.

For example, say a user had sorted a column in ascending order, and then added a new row. Initially, that row appears at the top. If the user first clicks to sort the column again in ascending order, the table will not re-sort, as it assumes the column is already in ascending order. The user must first sort on descending order and then ascending order.

If you want the data to automatically sort on a specific column in a specific order after inserting a row, then programmatically queue a `SortEvent` after the commit, and implement a handler to execute the sort.

5.5 Modifying the Attributes Displayed in the Table

Once you use the Data Controls panel to create a table, you can then delete attributes, change the order in which they are displayed, change the component used to display them, and change the attribute binding for the component. You can also add new attributes, or rebind the table to a new data control.

For more information about modifying existing UI components and bindings, see the "Modifying the Attributes Displayed in the Table" section of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

Displaying Master-Detail Data

This chapter describes how to use the Data Controls panel to create master-detail objects that are based on ADF Faces components. It shows how to display master-detail data by using prebuilt master-detail widgets, tables, trees and tree tables and how to work with selection events.

This chapter includes the following sections:

- Section 6.1, "Introduction to Displaying Master-Detail Data"
- Section 6.2, "Identifying Master-Detail Objects on the Data Controls Panel"
- Section 6.3, "Using Tables and Forms to Display Master-Detail Objects"
- Section 6.4, "Using Trees to Display Master-Detail Objects"
- Section 6.5, "Using Tree Tables to Display Master-Detail Objects"
- Section 6.6, "Using Selection Events with Trees and Tables"

For information about using a selection list to populate a collection with a key value from a related master or detail collection, see Chapter 7, "Creating Databound Selection Lists."

6.1 Introduction to Displaying Master-Detail Data

When objects have a master-detail relationship, you can declaratively create pages that display the data from both objects simultaneously. For example, the page shown in Figure 6-1 displays a country code in a form at the top of the page and its related states and provinces in a table at the bottom of the page. This is possible because the objects have a master-detail relationship. In this example, the `Country Code` is the master object and `States` is the detail object. ADF iterators automatically manage the synchronization of the detail data objects displayed for a selected master data object. Iterator bindings simplify building user interfaces that allow scrolling and paging through collections of data and drilling-down from summary to detail information.

Figure 6–1 Detail Table

Country Codes
CountryId US

Next Last

States and Provinces

CountryId	StateProvince
US	NV
US	AZ
US	MO
US	IN
US	CA
US	MI
US	NE
US	VA
US	SC
US	TX
US	WI
US	PA
US	NY
US	OR
US	FL

You display master and detail objects in forms and tables. The master-detail form can display these objects on separate pages. For example, you can display the master object in a table on one page and detail objects in a read-only form on another page.

Note: There are some cases when the master-detail UI components that JDeveloper provides cannot provide the functionality you require. For example, you may need to bind components programmatically instead of using the master-detail UI components.

A master object can have many detail objects, and each detail object can in turn have its own detail objects, down to many levels of depth. If one of the detail objects in this hierarchy is dropped from the Application Navigator as a master-detail form on a page, only its immediate parent master object displays on the page. The hierarchy will not display all the way up to the topmost parent object.

If you display the detail object as a tree or tree table object, it is possible to display the entire hierarchy with multiple levels of depth, starting with the topmost master object, and traversing detail children objects at each node.

6.2 Identifying Master-Detail Objects on the Data Controls Panel

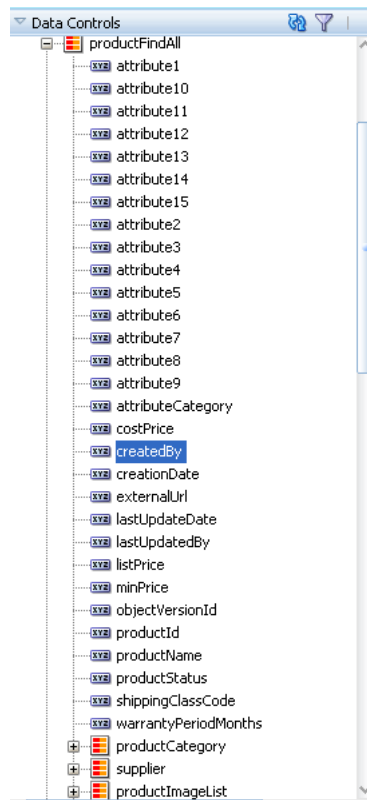
You can declaratively create pages that display master-detail data using the Data Controls panel. The Data Controls panel displays master-detail related objects in a hierarchy that mirrors the data model where the detail objects are children of the master objects.

To display master-detail objects as form or table objects, drag the detail object from the Data Controls panel and drop it on the page. Its master object is automatically created on the page.

Figure 6–2 shows two master-detail related accessor returned collections in the Data Controls panel. `ProductImageList` appears as a child of `ProductFindAll`.

Note: The master-detail hierarchy displayed in the Data Controls panel does not reflect the cardinality of the relationship (that is, one-to-many, one-to-one, many-to-many). The hierarchy simply shows which accessor returned collection (the master) is being used to retrieve one or more objects from another accessor returned collection (the detail).

Figure 6–2 Master-Detail Objects in the Data Controls Panel



In this example, the relationship between `ProductFindAll` and `ProductImageList` is a one-way relationship.

6.3 Using Tables and Forms to Display Master-Detail Objects

You can create a master-detail browse page in a single declarative action using the Data Controls panel. All you have to do is drop the detail accessor returned collection on the page and choose the type of widget you want to use.

The prebuilt master-detail widgets available from the Data Controls panel include range navigation that enables the end user to scroll through the data objects in collections. You can delete unwanted attributes by removing the text field or column from the page.

Figure 6–3 shows an example of a prebuilt master-detail widget, which displays products information in a form at the top of the page and stock levels in a table at the bottom of the page. When the user clicks the **Next** button to scroll through the records in the master data at the top of the page, the page automatically displays the related detail data.

Figure 6–3 Prebuilt Data Controls Panel Master-Detail Widget

Products

ProductId 1
 SupplierId 112
 CategoryId 4
 ProductName Plasma HD Television
 ProductStatus AVAILABLE

Next Last

WarehouseStockLevels

ProductId	WarehouseId	QuantityOnHand
1	101	200
1	102	750
1	103	1500
1	106	81
1	107	186
1	108	386
1	109	106
1	110	103
1	111	469

6.3.1 How to Display Master-Detail Objects in Tables and Forms

If you do not want to use the prebuilt master-detail widgets, you can drag and drop the master and detail objects individually from the Data Controls panel as tables and forms on a single page or on separate pages.

The Data Controls panel enables you to create both the master and detail widgets on one page with a single declarative action using prebuilt master-detail forms and tables.

To create a master-detail page using the prebuilt ADF master-detail forms and tables:

1. From the Data Controls panel, locate the detail object.
2. Drag and drop the detail object onto the JSF page.

Note: If you want to create an editable master-detail form, drop the master object and the detail object separately on the page.

3. In the context menu, choose one of the following **master-details** UI components:
 - **ADF Master Table, Detail Form:** Displays the master objects in a table and the detail objects in a read-only form under the table.
 When a specific data object is selected in the master table, the first related detail data object is displayed in the form below it. The user must use the form navigation to scroll through each subsequent detail data object.
 - **ADF Master Form, Detail Table:** Displays the master objects in a read-only form and the detail objects in a read-only table under the form.
 When a specific master data object is displayed in the form, the related detail data objects are displayed in a table below it.
 - **ADF Master Form, Detail Form:** Displays the master and detail objects in separate forms.

When a specific master data object is displayed in the top form, the first related detail data object is displayed in the form below it. The user must use the form navigation to scroll through each subsequent detail data object.

- **ADF Master Table, Detail Table:** Displays the master and detail objects in separate tables.

When a specific master data object is selected in the top table, the first set of related detail data objects is displayed in the table below it.

6.3.2 What Happens When You Create Master-Detail Tables and Forms

When you drag and drop an accessor returned collection from the Data Controls panel, JDeveloper does many things for you, including adding code to the JSF page and the corresponding entries in the page definition file.

6.3.2.1 Code Generated in the JSF Page

The JSF code generated for a prebuilt master-detail widget is similar to the JSF code generated when you use the Data Controls panel to create a read-only form or table. If you are building your own master-detail widgets, you might want to consider including similar components that are automatically included in the prebuilt master-detail tables and forms.

The tables and forms in the prebuilt master-detail widgets include a `panelHeader` tag that contains the fully qualified name of the data object populating the form or table. You can change this label as needed using a string or an EL expression that binds to a resource bundle.

If there is more than one data object in a collection, a form in a prebuilt master-detail widget includes four `commandButton` tags for range navigation: `First`, `Previous`, `Next`, and `Last`. These range navigation buttons enable the user to scroll through the data objects in the collection. The `actionListener` attribute of each button is bound to a data control operation, which performs the navigation. The `execute` property used in the `actionListener` binding invokes the operation when the button is clicked. (If the form displays a single data object, JDeveloper automatically omits the range navigation components.)

Tip: If you drop an **ADF Master Table, Detail Form** or **ADF Master Table, Detail Table** widget on the page, the parent tag of the detail component (for example, `panelHeader` tag or `table` tag) automatically has the `partialTriggers` attribute set to the `id` of the master component. At runtime, the `partialTriggers` attribute causes only the detail component to be rerendered when the user makes a selection in the master component, which is called *partial rendering*. When the master component is a table, ADF uses partial rendering, because the table does not need to be rerendered when the user simply makes a selection in the facet. Only the detail component needs to be rerendered to display the new data.

6.3.2.2 Binding Objects Defined in the Page Definition File

Example 6-1 shows the page definition file created for a master-detail page that was created by dropping `WarehouseStockLevelList`, which is a detail object under the `ProductFindAll` object, on the page as an **ADF Master Form, Detail Table**.

The `executables` element defines two `accessorIterators`: one for the product (the master object) and one for `WarehouseStockLevels` (the detail object). At runtime, the UI-aware data model and the row set iterator keep the row set of the

detail collection refreshed to the correct set of rows for the current master row as that current row changes.

The `bindings` element defines the value bindings. The attribute bindings that populate the text fields in the form are defined in the `attributeValues` elements. The `id` attribute of the `attributeValues` element contains the name of each data attribute, and the `IterBinding` attribute references an iterator binding to display data from the master object in the text fields.

The attribute bindings that populate the text fields in the form are defined in the `attributeValues` elements. The `id` attribute of the `attributeValues` element contains the name of each data attribute, and the `IterBinding` attribute references an iterator binding to display data from the master object in the text fields.

The range navigation buttons in the form are bound to the action bindings defined in the `action` elements. As in the attribute bindings, the `IterBinding` attribute of the action binding references the iterator binding for the master object.

The table, which displays the detail data, is bound to the table binding object defined in the `table` element. The `IterBinding` attribute references the iterator binding for the detail object.

Example 6–1 Binding Objects Defined in the Page Definition for a Master-Detail Page

```
<executables>
  <variableIterator id="variables"/>
  <iterator Binds="root" RangeSize="25" DataControl="SupplierFacadeLocal"
    id="SupplierFacadeLocalIterator"/>
  <accessorIterator MasterBinding="SupplierFacadeLocalIterator"
    Binds="productFindAll" RangeSize="25"
    DataControl="SupplierFacadeLocal"
    BeanClass="oracle.fodemo.supplier.model.Product"
    id="productFindAllIterator"
    ChangeEventPolicy="ppr"/>
  <accessorIterator MasterBinding="productFindAllIterator"
    Binds="warehouseStockLevelList" RangeSize="25"
    DataControl="SupplierFacadeLocal"
    BeanClass="oracle.fodemo.supplier.model.WarehouseStockLevel"
    id="warehouseStockLevelListIterator"
    ChangeEventPolicy="ppr"/>
</executables>
<bindings>
  <action IterBinding="productFindAllIterator" id="First"
    RequiresUpdateModel="true" Action="first"/>
  <action IterBinding="productFindAllIterator" id="Previous"
    RequiresUpdateModel="true" Action="previous"/>
  ...
  <attributeValues IterBinding="productFindAllIterator" id="attribute1">
    <AttrNames>
      <Item Value="warrantyPeriodMonths"/>
    </AttrNames>
  </attributeValues>
  ...
  <tree IterBinding="warehouseStockLevelListIterator"
    id="warehouseStockLevelList">
    <nodeDefinition DefName="oracle.fodemo.supplier.model.WarehouseStockLevel">
      <AttrNames>
        <Item Value="lastUpdateDate"/>
        <Item Value="objectVersionId"/>
        <Item Value="productId"/>
        <Item Value="quantityOnHand"/>
      </AttrNames>
    </nodeDefinition>
  </tree>
</bindings>
```

```

...
    <Item Value="warehouseId"/>
  </AttrNames>
</nodeDefinition>
</tree>
</bindings>

```

6.3.3 What Happens at Runtime: ADF Iterator for Master-Detail Tables and Forms

At runtime, an ADF iterator determines which row from the master table object to display in the master-detail form. When the form first displays, the first master table object row appears highlighted in the master section of the form. Detail table rows that are associated with the master row display in the detail section of the form.

As described in Section 6.3.2.2, "Binding Objects Defined in the Page Definition File," ADF iterators are associated with underlying `rowsetIterator` objects. These iterators manage which data objects, or rows, currently display on a page. At runtime, the row set iterators manage the data displayed in the master and detail components.

Both the master and detail row set iterators listen to row set navigation events, such as the user clicking the range navigation buttons, and display the appropriate row in the UI. In the case of the default master-detail components, the row set navigation events are the command buttons on a form (**First**, **Previous**, **Next**, **Last**).

The row set iterator for the detail collection manages the synchronization of the detail data with the master data. The detail row set iterator listens for row navigation events in both the master and detail collections. If a row set navigation event occurs in the master collection, the detail row set iterator automatically executes and returns the detail rows related to the current master row.

6.3.4 What You May Need to Know About Displaying Master-Detail Widgets on Separate Pages

The default master-detail components display the master-detail data on a single page. However, using the master and detail objects on the Data Controls panel, you can also display the collections on separate pages, and still have the binding iterators manage the synchronization of the master and detail objects.

To display master-detail objects on separate pages, create two pages, one for the master object and one for the detail object, using the individual tables or forms available from the Data Controls panel. Remember that the detail object iterator manages the synchronization of the master and detail data. Be sure to drag the appropriate detail object from the Data Controls panel when you create the page to display the detail data. For more information, see Section 6.2, "Identifying Master-Detail Objects on the Data Controls Panel."

To handle the page navigation, create an ADF task flow, and then add two view activities to it, one for the master page and one for the detail page. Add command buttons or links to each page, or use the default **Submit** button available when you create a form or table using the Data Controls panel. Each button must specify a navigation rule outcome value in the `action` attribute. In the `task-flow-definition.xml` file, add a navigation rule from the master data page to the detail data page, and another rule to return from the detail data page to the master data page. The `from-outcome` value in the navigation rules must match the outcome value specified in the action attribute of the buttons.

6.4 Using Trees to Display Master-Detail Objects

In addition to tables and forms, you can also display master-detail data in hierarchical trees. The ADF Faces `tree` component is used to display hierarchical data. It can display multiple root nodes that are populated by a binding on a master object. Each root node in the tree may have any number of branches, which are populated by bindings on detail objects. A tree can have multiple levels of nodes, each representing a detail object of the parent node. Each node in the tree is indented to show its level in the hierarchy.

The `tree` component includes mechanisms for expanding and collapsing the tree nodes; however, it does not have focusing capability. If you need to use focusing, consider using the ADF Faces `treeTable` component (for more information, see Section 6.5, "Using Tree Tables to Display Master-Detail Objects"). By default, the icon for each node in the tree is a folder; however, you can use your own icons for each level of nodes in the hierarchy.

Figure 6–4 shows an example of a tree that displays two levels of nodes: root and branch. The root node displays parent product categories such as Media, Office, and Electronics. The branch nodes display and subcategories under each parent category, such as Hardware, Supplies, and Software under the Office parent category.

Figure 6–4 Databound ADF Faces Tree



6.4.1 How to Display Master-Detail Objects in Trees

A *tree* consists of a hierarchy of nodes, where each subnode is a branch off a higher level node. Each node level in a databound ADF Faces `tree` component is populated by a different data collection. In JDeveloper, you define a databound tree using the Edit Tree Binding dialog, which enables you to define the rules for populating each node level in the tree. There must be one rule for each node level in the hierarchy. Each rule defines the following node-level properties:

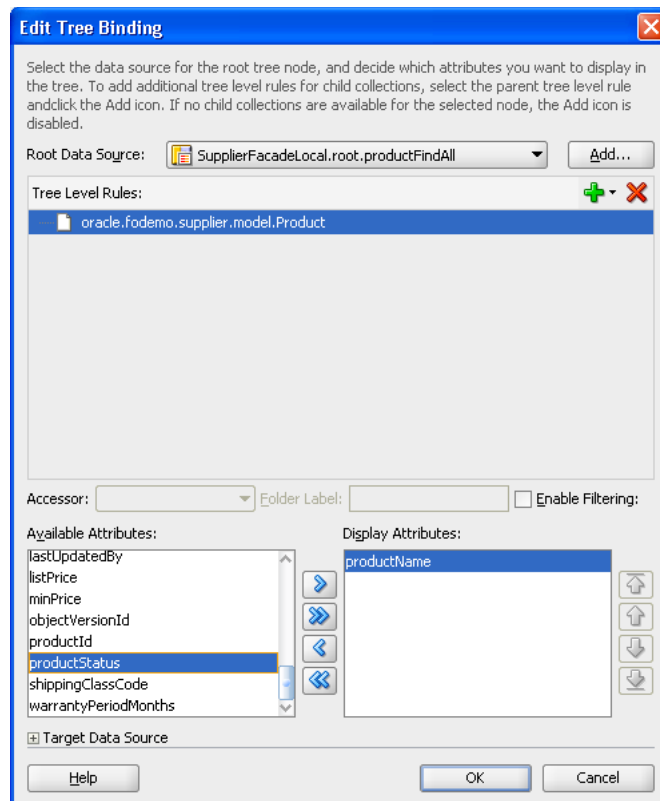
- The accessor returned collection that populates that node level
- The attributes from the accessor returned collection that are displayed at that node level

To display master-detail objects in a tree:

1. Drag the master object from the Data Controls panel, and drop it onto the page. This should be the master data that will represent the root level of the tree.
2. In the context menu, choose **Trees > ADF Tree**.

JDeveloper displays the Edit Tree Binding dialog, as shown in Figure 6–5. You use the binding editor to define a rule for each level that you want to appear in the tree.

Figure 6–5 Edit Tree Binding Dialog



3. In the **Root Data Source** dropdown list, select the accessor returned collection that will populate the root node level.

This will be the master data collection. By default, this is the same collection that you dragged from the Data Controls panel to create the tree, which was a master collection.

Tip: If you don't see the accessor returned collection you want in the **Root Data Source** list, click the **Add** button. In the Add Data Source dialog, select a data control and an iterator name to create a new data source.

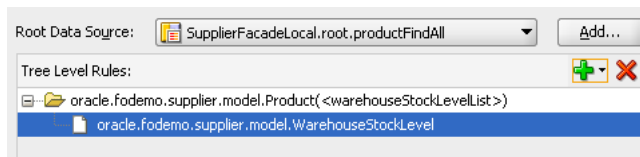
4. Click the **Add** icon to add the root data source you selected to the **Tree Level Rules** list.
5. In the **Tree Level Rules** list, select the data source you just added.
6. Select an attribute in the **Available Attributes** list and move it to the **Display Attributes** list.

The attribute will be used to display nodes at the master level.

After defining a rule for the master level, you must next define a second rule for the detail level that will appear under the master level in the tree.

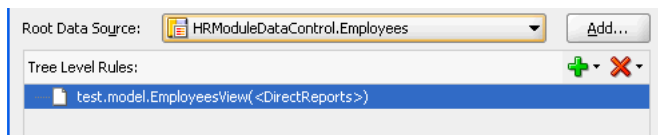
7. To add a second rule, click the **Add** icon above the **Tree Level Rules** list.

A detail data source should appear automatically under the master data source, as shown in Figure 6–6.

Figure 6–6 Master-Detail Tree Level Rules

For example, if you specified `ProductFindAll` as the master root data source, `WarehouseStockLevelList` will automatically appear underneath in the **Tree Level Rules** list, because the two data sources share a master-detail relationship.

If you are creating a tree with a recursive master-detail hierarchy, then you only need to define a rule that specifies a data source with a self-accessor. A recursive tree displays root nodes based on a single collection and displays the child nodes from the attributes of a self-accessor that recursively fetches data from that collection. The recursive tree differs from a typical master-detail tree because it requires only a single rule to define the branches of the tree. A recursive data source should display the data source followed by the name of the self-accessor in brackets, as shown in Figure 6–7.

Figure 6–7 Recursive Tree-Level Rule

For example, in a collection defined by `EmployeesView`, the root node of each branch could be specified by the `ManagerId` for the employee, and the child nodes of the same branch would then be the employees who are related to the `ManagerId`, as specified by the self-accessor `DirectReports`.

8. Click **OK**.
9. You can add data sources to the **Tree Level Rules** list to increase the number of nodes that display in the tree. The order of the remaining data sources should follow the hierarchy of the nodes you want to display in the tree.

6.4.2 What Happens When You Create an ADF Databound Tree

When you drag and drop from the Data Controls panel, JDeveloper does many things for you.

When you create a databound tree using the Data Controls panel, JDeveloper adds binding objects to the page definition file, and it also adds the tree tag to the JSF page. The resulting UI component is fully functional and does not require any further modification.

6.4.2.1 Code Generated in the JSF Page

Example 6–2 shows the code generated in a JSF page when you use the Data Controls panel to create a tree. This sample tree displays the order numbers as the root nodes and the product names as the leaf nodes.

Example 6–2 Code Generated in the JSF Page for a Databound Tree

```
<af:tree
  value="#{bindings.orderItemFindAll.treeModel}"
```

```

var="node"
selectionListener="#{bindings.orderItemFindAll.treeModel.makeCurrent}"
rowSelection="single" id="orderItemsTree">
<f:facet name="nodeStamp">
    <af:outputText value="#{node}" id="ot2"/>
</f:facet>
</af:tree>

```

By default, the `af:tree` tag is created inside a form. The `value` attribute of the `tree` tag contains an EL expression that binds the `tree` component to the `orderItemFindAll` tree binding object in the page definition file. The `treeModel` property in the binding expression refers to an ADF class that defines how the tree hierarchy is displayed, based on the underlying data model. The `var` attribute provides access to the current node.

In the `f:facet` tag, the `nodeStamp` facet is used to display the data for each node. Instead of having a component for each node, the `tree` repeatedly renders the `nodeStamp` facet, similar to the way rows are rendered for the ADF Faces table component.

The ADF Faces `tree` component uses an instance of the `oracle.adf.view.faces.model.PathSet` class to display expanded nodes. This instance is stored as the `treeState` attribute on the component. You may use this instance to programmatically control the expanded or collapsed state of an element in the hierarchy. Any element contained by the `PathSet` instance is deemed expanded. All other elements are collapsed.

6.4.2.2 Binding Objects Defined in the Page Definition File

Example 6-3 shows the binding objects defined in the page definition file for the ADF databound tree.

Example 6-3 Binding Objects Defined in the Page Definition File for a Databound Tree

```

<executables>
    <variableIterator id="variables"/>
    <iterator Binds="root" RangeSize="25" DataControl="SupplierFacadeLocal"
        id="SupplierFacadeLocalIterator"/>
    <accessorIterator MasterBinding="SupplierFacadeLocalIterator"
        Binds="orderItemFindAll" RangeSize="25"
        DataControl="SupplierFacadeLocal"
        BeanClass="oracle.fodemo.supplier.model.OrderItem"
        id="orderItemFindAllIterator"/>
</executables>
<bindings>
    <tree IterBinding="orderItemFindAllIterator" id="orderItemFindAll">
        <nodeDefinition DefName="oracle.fodemo.supplier.model.OrderItem"
            Name="orderItemFindAll0">
            <AttrNames>
                <Item Value="orderId"/>
            </AttrNames>
            <Accessors>
                <Item Value="product"/>
            </Accessors>
        </nodeDefinition>
        <nodeDefinition DefName="oracle.fodemo.supplier.model.Product"
            Name="orderItemFindAll1">
            <AttrNames>
                <Item Value="productName"/>
            </AttrNames>

```

```
        </nodeDefinition>  
    </tree>  
</bindings>
```

The `tree` element is the value binding for all the attributes displayed in the tree. The `iterBinding` attribute of the `tree` element references the iterator binding that populates the data in the tree. The `AttrNames` element within the `tree` element defines binding objects for *all* the attributes in the master collection. However, the attributes that you select to appear in the tree are defined in the `AttrNames` elements within the `nodeDefinition` elements.

The `nodeDefinition` elements define the rules for populating the nodes of the tree. There is one `nodeDefinition` element for each node, and each one contains the following attributes and subelements:

- `DefName`: An attribute that contains the fully qualified name of the data collection that will be used to populate the node
- `id`: An attribute that defines the name of the node
- `AttrNames`: A subelement that defines the attributes that will be displayed in the node at runtime
- `Accessors`: A subelement that defines the accessor attribute that returns the next branch of the tree

The order of the `nodeDefinition` elements within the page definition file defines the order or level of the nodes in the tree, where the first `nodeDefinition` element defines the root node. Each subsequent `nodeDefinition` element defines a subnode of the one before it.

6.4.3 What Happens at Runtime: Displaying an ADF Databound Tree

Tree components use `org.apache.myfaces.trinidad.model.TreeModel` to access data. This class extends `CollectionModel`, which is used by the ADF Faces table component to access data. For more information about the `TreeModel` class, refer to the ADF Faces Javadoc.

When a page with a `tree` is displayed, the iterator binding on the tree populates the root nodes. When a user expands or collapses a node to display or hide its branches, a `DisclosureEvent` event is sent. The `isExpanded` method on this event determines whether the user is expanding or collapsing the node. The `DisclosureEvent` event has an associated listener.

The `DisclosureListener` attribute on the `tree` is bound to the accessor attribute specified in the node rule defined in the page definition file. This accessor attribute is invoked in response to the `DisclosureEvent` event; in other words, whenever a user expands the node the accessor attribute populates the branch nodes.

6.5 Using Tree Tables to Display Master-Detail Objects

Use the ADF Faces `treeTable` component to display a hierarchy of master-detail collections in a table. The advantage of using a `treeTable` component rather than a `tree` component is that the `treeTable` component provides a mechanism that enables users to focus the view on a particular node in the tree.

For example, you can create a tree table that displays three levels of nodes: countries, states or provinces, and cities. Each root node represents an individual country. The

branches off the root nodes display the state or provinces in the country. Each state or province node branches to display the cities contained in it.

As with trees, to create a tree table with multiple nodes, it is necessary to have master-detail relationships between the collections. For example, to create a tree table with three levels of country, state, and city, it was necessary to have a master-detail relationship from the `CountryCodes` collection to the `StatesandProvinces` collection, and a master-detail relationship from the `StatesandProvinces` collection to the `Cities` collection.

A databound ADF Faces `treeTable` displays one root node at a time, but provides navigation for scrolling through the different root nodes. Each root node can display any number of branch nodes. Every node is displayed in a separate row of the table, and each row provides a focusing mechanism in the leftmost column.

You can edit the following `treeTable` component properties in the Property Inspector:

- Range navigation: The user can click the **Previous** and **Next** navigation buttons to scroll through the root nodes.
- List navigation: The list navigation, which is located between the **Previous** and **Next** buttons, enables the user to navigate to a specific root node in the data collection using a selection list.
- Node expanding and collapsing mechanism: The user can open or close each node individually or use the **Expand All** or **Collapse All** command links. By default, the icon for opening and closing the individual nodes is an arrowhead with a plus or minus sign. You can also use a custom icon of your choosing.
- Focusing mechanism: When the user clicks on the focusing icon (which is displayed in the leftmost column) next to a node, the page is redisplayed showing only that node and its branches. A navigation link is provided to enable the user to return to the parent node.

6.5.1 How to Display Master-Detail Objects in Tree Tables

The steps for creating an ADF Faces databound tree table are exactly the same as those for creating an ADF Faces databound tree, except that you drop the accessor returned collection as an **ADF Tree Table** instead of an **ADF Tree**.

6.5.2 What Happens When You Create a Databound Tree Table

When you drag and drop from the Data Controls panel, JDeveloper does many things for you.

When you create a databound tree table using the Data Controls panel, JDeveloper adds binding objects to the page definition file, and it also adds the `treeTable` tag to the JSF page. The resulting UI component is fully functional and does not require any further modification.

6.5.2.1 Code Generated in the JSF Page

Example 6-4 shows the code generated in a JSF page when you use the Data Controls panel to create a tree table. This sample tree table displays two levels of nodes: products and stock levels.

By default, the `treeTable` tag is created inside a form. The `value` attribute of the tree table tag contains an EL expression that binds the tree component to the binding object that will populate it with data. The `treeModel` property refers to an

ADF class that defines how the tree hierarchy is displayed, based on the underlying data model. The `var` attribute provides access to the current node.

Example 6–4 Code Generated in the JSF Page for a Databound ADF Faces Tree Table

```
<af:treeTable value="#{bindings.orderItemFindAll.treeModel}" var="node"
  selectionListener="#{bindings.orderItemFindAll.treeModel.makeCurrent}"
  rowSelection="single" id="tt1">
  <f:facet name="nodeStamp">
    <af:column id="c1">
      <af:outputText value="#{node}" id="ot1"/>
    </af:column>
  </f:facet>
  <f:facet name="pathStamp">
    <af:outputText value="#{node}" id="ot2"/>
  </f:facet>
</af:treeTable>
```

In the `facet` tag, the `nodeStamp` facet is used to display the data for each node. Instead of having a component for each node, the tree repeatedly renders the `nodeStamp` facet, similar to the way rows are rendered for the ADF Faces `table` component. The `pathStamp` facet renders the column and the path links above the table that enable the user to return to the parent node after focusing on a detail node.

6.5.2.2 Binding Objects Defined in the Page Definition File

The binding objects created in the page definition file for a tree table are exactly the same as those created for a tree.

6.5.3 What Happens at Runtime: Events

Tree components use `oracle.adf.view.faces.model.TreeModel` to access data. This class extends `CollectionModel`, which is used by the ADF Faces `table` component to access data. For more information about the `TreeModel` class, refer to the ADF Faces Javadoc.

When a page with a tree table is displayed, the iterator binding on the `treeTable` component populates the root node and listens for a row navigation event (such as the user clicking the **Next** or **Previous** buttons or selecting a row from the range navigator). When the user initiates a row navigation event, the iterator displays the appropriate row.

If the user changes the view focus (by clicking on the component's focus icon), the `treeTable` component generates a focus event (`FocusEvent`). The node to which the user wants to change focus is made the current node before the event is delivered. The `treeTable` component then modifies the `focusPath` property accordingly. You can bind the `FocusListener` attribute on the tree to a method on a managed bean. This method will then be invoked in response to the focus event.

When a user expands or collapses a node, a disclosure event (`DisclosureEvent`) is sent. The `isExpanded` method on the disclosure event determines whether the user is expanding or collapsing the node. The disclosure event has an associated listener, `DisclosureListener`. The `DisclosureListener` attribute on the tree table is bound to the accessor attribute specified in the node rule defined in the page definition file. This accessor attribute is invoked in response to a disclosure event (for example, the user expands a node) and returns the collection that populates that node.

The `treeTable` component includes **Expand All** and **Collapse All** links. When a user clicks one of these links, the `treeTable` sends a `DisclosureAllEvent` event. The `isExpandAll` method on this event determines whether the user is expanding or collapsing all the nodes. The table then expands or collapses the nodes that are children of the root node currently in focus. In large trees, the expand all command will not expand nodes beyond the immediate children. The ADF Faces `treeTable` component uses an instance of the `oracle.adf.view.faces.model.PathSet` class to determine expanded nodes. This instance is stored as the `treeState` attribute on the component. You can use this instance to programmatically control the expanded or collapsed state of a node in the hierarchy. Any node contained by the `PathSet` instance is deemed expanded. All other nodes are collapsed. This class also supports operations like `addAll()` and `removeAll()`.

Like the ADF Faces `table` component, a `treeTable` component provides for range navigation. However, instead of using the `rows` attribute, the `treeTable` component uses a `rowsByDepth` attribute whose value is a space-separated list of non-negative numbers. Each number defines the range size for a node level on the tree. The first number is the root node of the tree, and the last number is for the branch nodes. If there are more branches in the tree than numbers in the `rowsByDepth` attribute, the tree uses the last number in the list for the remaining branches. Each number defines the limit on the number items displayed at one time in each branch. If you want to display all items in a branch, specify 0 in that position of the list.

For example, if the `rowsByDepth` attribute is set to 0 0 3, all root nodes will be displayed, all direct children of the root nodes will be displayed, but only three nodes will display per branch after that. The `treeTable` component includes links to navigate to additional nodes, enabling the user to display the additional nodes.

For more information about the ADF Faces `treeTable` component, refer to the `oracle.adf.view.faces.component.core.data.CoreTreeTable` class in the ADF Faces Javadoc.

6.5.4 Using the `TargetIterator` Property

You can expand a node binding in the page definition editor to view the page's `nodeDefinition` elements. These are the same tree binding rules that you can configure in the tree binding dialog.

For each node definition (rule), you can specify an optional `TargetIterator` property. Its value is an EL expression that is evaluated at runtime when the user selects a row in the tree. The EL expression evaluates an iterator binding in the current binding container. The iterator binding's row key attributes match (in order, number, and data type) the row key of the iterator from which the `nodeDefinition` type's rows are retrieved for the tree.

At runtime, when the tree control receives a `selectionChanged` event, it passes in the list of keys for each level of the tree. These keys uniquely identify the selected node.

The tree binding starts at the top of the tree. For each tree level whose key is present in the `Currently Selected Tree Node Keys` list, if there is a `TargetIterator` property configured for that `nodeDefinition`, the tree binding performs a `setCurrentRowWithKey()` operation on the selected target iterator. It uses the key from the appropriate level of the `Currently Selected Tree Node Keys` list.

6.6 Using Selection Events with Trees and Tables

There may be cases when you need to determine which node in a tree or tree table has been selected in order to handle some processing in your application. For example, when a user selects a category node in a **Browse** tree, a selection event is fired. The listener associated with this event needs to determine the product category of the node selected, and then to return all products whose category attribute matches that value.

6.6.1 How to Use Selection Events with Trees and Tables

To programmatically use selection events, you need to create a listener in a managed bean that will handle the selection event and perform the needed logic. You then need to bind the `selectionListener` attribute of the tree or table to that listener.

To use selection events with trees and tables:

1. If one does not already exist, create a managed bean to contain the needed listener.
2. Create a listener method on the managed bean. For more information about creating listener methods, see the "Using ADF Faces Server Events" section of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*. Your listener should do the following:
 - a. Access the component using the event source. Example 6–5 shows how the `productCategoriesTreeSelectionListener` method on the `HomeBean` managed bean accesses the tree that launched the selection event.

Example 6–5 Getting the Source of an Event

```
public void productCategoriesTreeSelectionListener(SelectionEvent evt) {
    RichTree tree = (RichTree)evt.getSource();
```

For more information about finding the event source component, see the "How to Return the Original Source of the Event" section of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

- b. Access the tree model to get the value of the model, use the `RowKeySet` object to get the currently selected node, and then set that as the current row on the model, as shown in Example 6–6. For more information about `RowKeySet` objects, see Section 6.6.2, "What Happens at Runtime: RowKeySet Objects and SelectionEvent Events."

Example 6–6 Setting the Current Row on a Tree Model

```
TreeModel model = (TreeModel)tree.getValue();
RowKeySet rowKeySet = evt.getAddedSet();
Object key = rowKeySet.iterator().next();
model.setRowKey(key);
```

- c. You can now add logic to execute against the currently selected row. For example, the `productCategoriesTreeSelectionListener` method uses the value binding of the selected row to determine the category ID, and then uses that value as the parameter for another method that, when executed, returns all products with that category ID, as shown in Example 6–7.

Example 6–7 Returning Objects That Match a Given Attribute Value

```
JUCtrlValueBinding nodeBinding =
    (JUCtrlValueBinding)model.getRowData();
```



```
Number catId = (Number)nodeBinding.getAttribute("CategoryId");
_selectedCategory = (String)nodeBinding.getAttribute("CategoryName");

OperationBinding ob =
    ADFUtils.findOperation("ProductsByCategoriesExecuteWithParams");
ob.getParamsMap().put("category", catId);
ob.execute();
```

3. On the associated JSF page, select the tree or table component. In the Property Inspector, expand the **Behavior** section and set the value of the `SelectionListener` attribute to the listener method just created. You can use the **Edit** option from the dropdown method to declaratively select the bean and the method.

6.6.2 What Happens at Runtime: RowKeySet Objects and SelectionEvent Events

Whenever a user selects a node in a tree (or a row in a table), the component triggers selection events. A `selectionEvent` event reports which rows were just deselected and which rows were just selected. The current selection, that is, the selected row or rows, is managed by the `RowKeySet` object, which keeps track of all currently selected nodes by adding and deleting the associated key for the row into or out of the key set. When a user selects a new node, and the tree or table is configured for single selection, then the previously selected key is discarded and the newly selected key is added. If the tree or table is configured for multiple selection, then the newly selected keys are added to the set, and the previously selected keys may or may not be discarded, based on how the nodes were selected. For example, if the user pressed the CTRL key, then the newly selected nodes would be added to the current set.

Creating Databound Selection Lists

This chapter describes how to add selection lists components to pages. It includes instructions for creating selection components with fixed-value lists or dynamically generated lists. It also describes how to add navigation list bindings to let users navigate through a list of objects in a collection.

This chapter includes the following sections:

- Section 7.1, "Introduction to Selection Lists"
- Section 7.2, "Creating a Single Selection List"
- Section 7.3, "Creating a List with Navigation List Binding"

7.1 Introduction to Selection Lists

Selection lists work the same way as do standard JSF list components. ADF Faces list components, however, provide extra functionality such as support for label and message display, automatic form submission, and partial page rendering.

When the user selects an item from a navigation list, a corresponding component bound to the list also changes its value in response to the selection. For example, when the user selects a product from a shopping list, the table that is bound to the products list updates to display the details of the selected product.

7.2 Creating a Single Selection List

ADF Faces Core includes components for selecting a single value and multiple values from a list. For example, `selectOneChoice` allows the user to select an item from a dropdown list, and `selectManyChoice` allow the user to select several items from a list of checkboxes. Selection list components are described in Table 7-1.

Table 7-1 ADF Faces Single and Multiple List Components

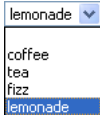
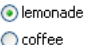
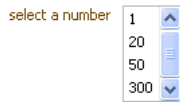
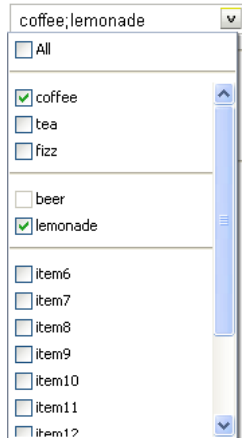

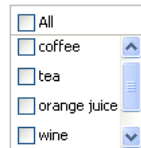
ADF Faces component	Description	Example
<code>SelectOneChoice</code>	Select a single value from a list of items.	
<code>SelectOneRadio</code>	Select a single value from a set of radio buttons.	

Table 7-1 (Cont.) ADF Faces Single and Multiple List Components

ADF Faces component	Description	Example
SelectOneListbox	Select a single value from a scrollable list of items.	
SelectManyChoice	Select multiple values from a scrollable list of checkboxes. Each selection displays at the top of the list.	
SelectManyCheckbox	Select multiple values from a group of checkboxes.	
SelectManyListbox	Select multiple values from a scrollable list of checkboxes.	

You can create selection lists using the `SelectOneChoice` ADF Faces component. The steps are similar for creating other single-value selection lists, such as `SelectOneRadio` and `SelectOneListbox`.

A databound selection list displays values from an accessor returned collection or a static list and updates an attribute in another collection or a method parameter based on the user's selection. When adding a binding to a list, you use an attribute from the data control that will be populated by the selected value in the list.

Note: Using an ADF Model list binding with the `valuePassThru=true` on a `selectOneChoice` component is not supported. The list binding will return indexes, not values.

To create a selection list, you choose a base data source and a list data source in the Edit List Binding dialog:

- Base data source: Select the accessor returned collection that you want to bind to your control and that contains the attributes to be updated from user selections.
- List data source: Select the accessor returned collection that contains the attributes to display.

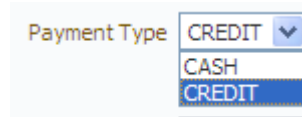
You can create two types of selection lists in the Edit List Binding dialog:

- **Static list:** List selections are based on a fixed list that you create manually by entering values one at a time into the editor.
- **Dynamic list:** List selections are generated dynamically based on one or more databound attribute values.

7.2.1 How to Create a Single Selection List Containing Fixed Values

You can create a selection list containing selections that you code yourself, rather than retrieving the values from another data source.

Figure 7–1 Selection List Bound to a Fixed List of Values



Before you begin:

Prepare a list of values that you will enter into the component as a fixed list.

To create a list bound to a fixed list of values:

1. From the Data Controls panel, drag and drop the attribute onto the JSF page and choose **Create > Single Selections > ADF Select One Choice**.

The Edit List Binding dialog displays. The accessor returned collection containing the attribute you dropped on the JSF page is selected by default in the **Base Data Source** list.

To select a different accessor returned collection, click the **Add** icon next to the list.

2. Select the **Fixed List** radio button.

The **Fixed List** option lets end users choose a value from a static list that you define.

3. In the **Base Data Source Attribute** list, choose an attribute.

The **Base Data Source Attribute** list contains all of the attributes in the collection you selected in the **Base Data Source** list.

4. In the **Set of Values** box, enter each value you want to appear in the list. Press the Enter key to set a value before typing the next value. For example, you could add the payment types `ATM` and `check`.

The order in which you enter the values is the order in which the list items are displayed in the `SelectOneChoice` control at runtime.

The `SelectOneChoice` component supports a `null` value. If the user has not selected an item, the label of the item is shown as blank, and the value of the component defaults to an empty string. Instead of using blank or an empty string, you can specify a string to represent the `null` value. By default, the new string appears at the top of the list.

5. Click **OK**.

7.2.2 How to Create a Single Selection List Containing Dynamically Generated Values

You can populate a selection list component with values dynamically at runtime.

Before you begin:

Define two data sources: one for the list data source that provides the dynamic list of values, and the other for the base data source that is to be updated based on the user's selection.

To create a selection list bound containing dynamically generated values:

1. From the Data Controls panel, drag and drop the attribute onto the JSF page and choose **Create > Single Selections > ADF Select One Choice**.

The Edit List Binding dialog displays. The accessor returned collection containing the attribute you dropped on the JSF page is selected by default in the **Base Data Source** list.

To select a different accessor returned collection, click the **Add** icon next to the list.

2. Select the **Dynamic List** radio button.

The **Dynamic List** option lets you specify one or more base data source attributes that will be updated from another set of bound values.

3. Click the **Add** button next to **List Data Source**.
4. In the Add Data Source dialog, select the accessor returned collection that will populate the values in the selection list.

Note: The list and base collections do not have to form a master-detail relationship, but the attribute in the list collection must have the same type as the base collection attributes.

5. Accept the default iterator name and click **OK**.

The **Data Mapping** section of the Edit List Binding dialog updates with a default data value and list attribute. The **Data Value** control contains the attribute on the accessor returned collection that is updated when the user selects an item in the selection list. The **List Attribute Control** contains the attribute that populates the values in the selection list.

6. You can accept the default mapping or select different attributes items from the **Data Value** and **List Attribute** lists to update the mapping.

To add a second mapping, click **Add**.

7. Click **OK**.

7.2.3 What Happens When You Create a Fixed Selection List

When you add a fixed selection list, JDeveloper adds source code to the JSF page and list and iterator binding objects to the page definition file.

Example 7-1 shows the page source code after you add a fixed `SelectOneChoice` component to it.

Example 7-1 Fixed SelectOneChoice List in JSF Page Source Code

```
<af:selectOneChoice value="#{bindings.paymentType.inputValue}"
    label="#{bindings.paymentType.label}"
    required="#{bindings.paymentType.hints.mandatory}"
    shortDesc="#{bindings.paymentType.hints.tooltip}" id="soc1">
    <f:selectItems value="#{bindings.paymentType.items}" id="si1"/>
</af:selectOneChoice>
```

The `f:selectItems` tag, which provides the list of items for selection, is bound to the `items` property on the `paymentType` list binding object in the binding container

In the page definition file, JDeveloper adds the definitions for the iterator binding objects into the `executables` element, and the list binding object into the `bindings` element, as shown in Example 7–2.

Example 7–2 List Binding Object for the Fixed Selection List in the Page Definition File

```
<executables>
  <variableIterator id="variables"/>
  <iterator Binds="root" RangeSize="25" DataControl="OrdersSessionEJBLocal"
    id="OrdersSessionEJBLocalIterator"/>
  <accessorIterator MasterBinding="OrdersSessionEJBLocalIterator"
    Binds="customerFindAll" RangeSize="25"
    DataControl="OrdersSessionEJBLocal"
    BeanClass="model.Customer" id="customerFindAllIterator"/>
  <accessorIterator MasterBinding="customerFindAllIterator"
    Binds="SOrdList" RangeSize="25"
    DataControl="OrdersSessionEJBLocal"
    BeanClass="model.Ord" id="SOrdListIterator"/>
  <iterator Binds="root" RangeSize="25" DataControl="OrdersSessionEJBLocal"
    id="OrdersSessionEJBLocalIterator1"/>
  <accessorIterator MasterBinding="OrdersSessionEJBLocalIterator1"
    Binds="empFindAll" RangeSize="-1"
    DataControl="OrdersSessionEJBLocal"
    BeanClass="model.Emp" id="empFindAllIterator"/>
</executables>
<bindings>
  <list IterBinding="SOrdListIterator" id="paymentType" DTSupportsMRU="true"
    StaticList="true">
    <AttrNames>
      <Item Value="paymentType"/>
    </AttrNames>
    <ValueList>
      <Item Value="CASH"/>
      <Item Value="CREDIT"/>
    </ValueList>
  </list>
</bindings>
```

7.2.4 What Happens When You Create a Dynamic Selection List

When you add a dynamic selection list to a page, JDeveloper adds source code to the JSF page, and list and iterator binding objects to the page definition file.

Example 7–3 shows the page source code after you add a dynamic `SelectOneChoice` component to it.

Example 7–3 Dynamic `SelectOneChoice` List in JSF Page Source Code

```
<af:selectOneChoice value="#{bindings.salesRepId.inputValue}"
  label="#{bindings.salesRepId.label}"
  required="#{bindings.salesRepId.hints.mandatory}"
  shortDesc="#{bindings.salesRepId.hints.tooltip}" id="soc2">
  <f:selectItems value="#{bindings.salesRepId.items}" id="si2"/>
</af:selectOneChoice>
```

The `f:selectItems` tag, which provides the list of items for selection, is bound to the `items` property on the `salesRepId` list binding object in the binding container.

In the page definition file, JDeveloper adds the definitions for the iterator binding objects into the `executables` element, and the list binding object into the `bindings` element, as shown in Figure 7-4.

Example 7-4 List Binding Object for the Dynamic Selection List in the Page Definition File

```
<executables>
  <variableIterator id="variables"/>
  <iterator Binds="root" RangeSize="25" DataControl="OrdersSessionEJBLocal"
    id="OrdersSessionEJBLocalIterator"/>
  <accessorIterator MasterBinding="OrdersSessionEJBLocalIterator"
    Binds="customerFindAll" RangeSize="25"
    DataControl="OrdersSessionEJBLocal"
    BeanClass="model.Customer" id="customerFindAllIterator"/>
  <accessorIterator MasterBinding="customerFindAllIterator"
    Binds="SOrdList" RangeSize="25"
    DataControl="OrdersSessionEJBLocal"
    BeanClass="model.Ord" id="SOrdListIterator"/>
  <iterator Binds="root" RangeSize="25" DataControl="OrdersSessionEJBLocal"
    id="OrdersSessionEJBLocalIterator1"/>
  <accessorIterator MasterBinding="OrdersSessionEJBLocalIterator1"
    Binds="empFindAll" RangeSize="-1"
    DataControl="OrdersSessionEJBLocal"
    BeanClass="model.Emp" id="empFindAllIterator"/>
</executables>
</bindings>
  <list IterBinding="SOrdListIterator" id="salesRepId" DTSupportsMRU="true"
    StaticList="false"
    ListIter="empFindAllIterator">
    <AttrNames>
      <Item Value="salesRepId"/>
    </AttrNames>
    <ListAttrNames>
      <Item Value="id"/>
    </ListAttrNames>
    <ListDisplayAttrNames>
      <Item Value="firstName"/>
      <Item Value="lastName"/>
    </ListDisplayAttrNames>
  </list>
</bindings>
```

The `AttrNames` element specifies the base data source attributes returned by the base iterator. The `ListAttrNames` element defines the list data source attributes that are mapped to the base data source attributes. The `ListDisplayAttrNames` element specifies the list data source attribute that populates the values users see in the list at runtime.

7.3 Creating a List with Navigation List Binding

Navigation list binding lets users navigate through the objects in a collection. As the user changes the current object selection using the navigation list component, any other component that is also bound to the same collection through its attributes will display from the newly selected object.

In addition, if the collection whose current row you change is the master collection in a data model master-detail relationship, the row set in the detail collection is automatically updated to show the appropriate data for the new current master row.

Before you begin:

Create an accessor returned collection in the Data Controls panel.

To create a list that uses navigation list binding:

1. From the Data Controls panel, drag and drop an accessor returned collection to the page and choose **Create > Navigation > ADF Navigation Lists**.
2. In the Edit List Binding dialog, from the **Base Data Source** dropdown list, select the collection whose members will be used to create the list.

This should be the collection you dragged from the Data Controls panel. If the collection does not appear in the dropdown menu, click the **Add** button to select the collection you want.

3. From the **Display Attribute** dropdown list, select a single attribute, all the attributes, or choose **Select Multiple** to launch a selection dialog.

In the Select Multiple Display Attributes dialog, shuttle the attributes you want to display from the **Available Attributes** pane to the **Attributes to Display** pane. Click **OK** to close the dialog.

4. Click **OK**.

Creating Databound Search Forms

This chapter describes how to create search forms to perform complex searches on multiple attributes and search forms to search on a single attribute. For complex query search forms, it describes how to set up the query search form mode, results table, saved searches list, and personalization. For single attribute search forms, it describes how to configure the form layout. In addition, it includes information on using Query-by-Example (QBE) filtered table searches.

This chapter includes the following sections:

- Section 8.1, "Introduction to Creating Search Forms"
- Section 8.2, "Creating Query Search Forms"
- Section 8.3, "Setting Up Search Form Properties"
- Section 8.4, "Creating Quick Query Search Forms"
- Section 8.5, "Creating Standalone Filtered Search Tables"

8.1 Introduction to Creating Search Forms

You can create search forms that allow users to enter search criteria into input fields for known attributes of an object. The search criteria can be entered via input text fields or selected from a list of values in a popup list picker or dropdown list box. The entered criteria is constructed into a query to be executed. The results of the query can be displayed as a table, a form, or another UI component.

Search forms are region-based components that are reusable and personalizable. They encapsulate and automate many of the actions and iterator management operations required to perform a query. You can create several search forms on the same page without any need to change or create new iterators.

The search forms are based on the model-driven `af:query` and `af:quickQuery` components. Because these underlying components are model-driven, the search form will change automatically to reflect changes in the model. The view layer does not need to be changed.

The *query search form* is a full-featured search form. The *quick query search form* is a simplified form with only one search criteria. Each of these search forms can be combined with a filtered table to display the results, thereby enabling additional search capabilities. You can also create a standalone filtered table to perform searches without the query or quick query search panel.

A *filtered table* is a table that has additional Query-by-Example (QBE) search criteria fields above each searchable column. When the filtering option of a table is enabled, you can enter QBE-style search criteria for each column to filter the query results.

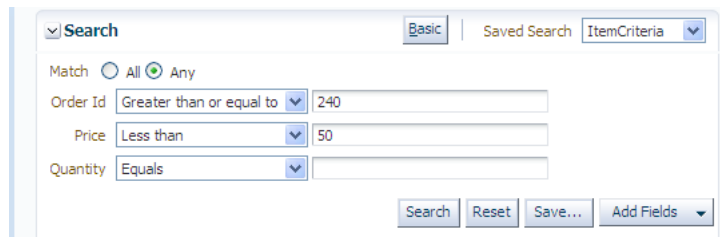
For more information about individual query and table components, see the "Using Query Components" and the "Using Tables and Trees" chapters of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

8.1.1 Query Search Forms

The query search form is the standard form for complex transactional searches. You can build complex search forms with multiple search criteria fields each with a dropdown list of built-in operators. You can also add custom operators and customize the list. The query search form supports lists of values, AND and OR conjunctions, and saving searches for future use.

A query search form has a basic mode and an advanced mode. The user can toggle between the two modes using the basic/advanced button. At design time, you can declaratively specify form properties (such as setting the default state) to be either basic or advanced. Figure 8–1 shows an advanced mode query search form with three search criteria.

Figure 8–1 Advanced Mode Query Search Form with Three Search Criteria Fields



The advanced mode query form features are:

- Selecting search criteria operators from a dropdown list
- Adding custom operators and deleting standard operators
- Selecting WHERE clause conjunctions of either AND or OR (match all or match any)
- Dynamically adding and removing search criteria fields at runtime
- Saving searches for future use
- Personalizing saved searches

Typically, the query search form in either mode is used with an associated results table or tree table. For example, the query results for the search form in Figure 8–1 may be displayed in a table, as shown in Figure 8–2.

Figure 8–2 Results Table for a Query Search

itemId	ordId	price	quantity	quantityShip
4	240	\$8.00	17	17
5	240	\$11.00	18	18
6	240	\$12.00	20	20
7	240	\$16.25	56	56
1	241	\$21.95	88	88
3	240	\$60.00	18	18
4	241	\$28.00	15	15
6	241	\$8.00	54	54
1	242	\$11.00	90	90
2	242	\$21.95	19	19
1	243	\$21.95	17	17
2	243	\$45.00	15	15
3	243	\$11.00	11	11
4	243	\$40.95	14	14
1	244	\$75.00	41	41

The basic mode has all the features of the advanced mode except that it does not allow the user to dynamically add search criteria fields. Figure 8–3 shows a basic mode query search form with three search criteria field. Notice the lack of a dropdown list next to the **Save** button used to add search criteria fields in the advanced mode.

Figure 8–3 Basic Mode Query Form with Three Search Criteria Fields

In either mode, each search criteria field can be modified by selecting operators such as *Greater Than* and *Equal To* from a dropdown list, and the entire search panel can be modified by the **Match All/Any** radio buttons. Partial page rendering is also supported by the search forms in almost all situations. For example, if a *Between* operator is chosen, another input field will be displayed to allow the user to select the upper range.

A *Match All* selection implicitly uses AND conjunctions between the search criteria in the WHERE clause of the query. A *Match Any* selection implicitly uses OR conjunctions in the WHERE clause. Example 8–1 shows how a simplified WHERE clause may appear (the real WHERE in the view criteria is different) when *Match All* is selected for the search criteria shown in Figure 8–1.

Example 8–1 Simplified WHERE Clause Fragment When "Match All" Is Selected

```
WHERE (OrderId=4) AND (Quantity < 50) AND (Price > 100)
```

Example 8–2 shows a simplified WHERE clause if **Match Any** is selected for the search criteria shown in Figure 8–3.

Example 8–2 Simplified WHERE Clause Fragment When "Match Any" Is selected

```
WHERE (orderId=4) OR (Quantity = 20) OR (Price > 100)
```

Advanced mode query forms allow users to dynamically add search criteria fields to the query panel to perform more complicated queries. These user-created search

criteria fields can be deleted, but the user cannot delete existing fields. Figure 8–4 shows how the **Add Fields** dropdown list is used to add the **CategoryId** criteria field to the search form.

Figure 8–4 Dynamically Adding Search Criteria Fields at Runtime

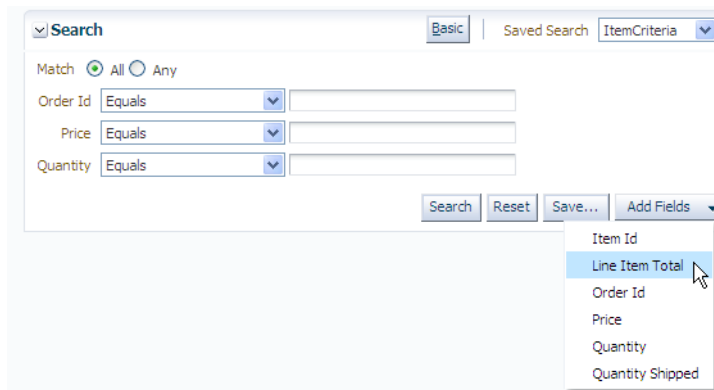
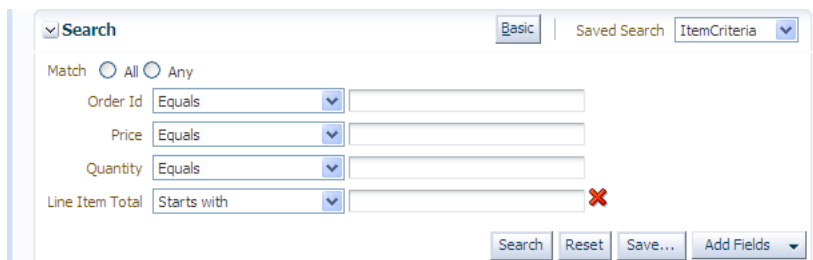


Figure 8–5 shows a user-added search criteria with the delete icon to its right. Users can click the delete icon to remove the criteria.

Figure 8–5 User-Added Search Criteria with Delete Icon



If either **Match All** or **Match Any** is selected and then the user dynamically adds the second instance of a search criteria, then both **Match All** and **Match Any** will be deselected. The user must reselect either **Match All** or **Match Any** before clicking the **Search** button.

If you intend for a query search form to have both a basic and an advanced mode, you can define each search criteria field to appear only for basic, only for advanced, or for both. When the user switches from one mode to the other, only the search criteria fields defined for that mode will appear. For example, suppose three search fields for basic mode (A, B, C) and three search fields for advanced mode (A, B, D) are defined for a query. When the query search form is in basic mode, search criteria fields A, B, and C will appear. When it is in advanced mode, then fields A, B, and D will appear. Any search data that was entered into the search fields will also be preserved when the form returns to that mode. If the user entered 35 into search field C in basic mode, switched to advanced mode, and then switched back to basic, field C would reappear with value 35.

Along with using the basic or advanced mode, you can also determine how much of the search form will display. The default setting displays the whole form. You can also configure the query component to display in compact mode or simple mode. The compact mode has no header or border, and the **Saved Search** dropdown lists moves next to the expand/collapse icon. Figure 8–6 shows a query component set to compact mode.

Figure 8–6 Query Component in Compact Mode

The simple mode displays the component without the header and footer, and without the buttons normally displayed in those areas. Figure 8–7 shows the same query component set to simple mode.

Figure 8–7 Query Component in Simple Mode

Users can also create saved searches at runtime to save the state of a search for future use. The entered search criteria values, the basic/advanced mode state, and the layout of the results table/component can be saved by clicking the **Save** button to open a Save Search dialog, as shown in Figure 8–8. User-created saved searches persist for the session. If they are intended to be available beyond the session, you must configure a persistent data store to store them. For Oracle ADF, you can use an access-controlled data source such as MDS. For more information about using MDS, see the Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework.

Figure 8–8 Runtime Saved Search Dialog Window

Table 8–1 lists the possible scenarios for creators of saved searches, the method of their creation, and their availability.

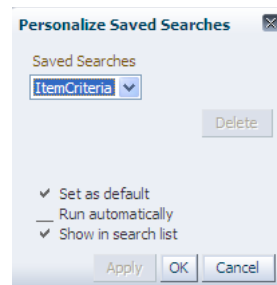
Table 8–1 Design Time and Runtime Saved Searches

Creator	Created at Design time as View Criteria	Created at Runtime with the Save Button
Developer	Developer-created saved searches (system searches) are created during application development and typically are a part of the software release. They are created at design time as view criteria. They are usually available to all users of the application and appear in the lower part of the Saved Search dropdown list.	
Administrator		Administrator-created saved searches are created during predeployment by site administrators. They are created before the site is made available to the general end users. Administrators can create saved searches (or view criteria) using the JDeveloper design time when they are logged in with the appropriate role. These saved searches (or view criteria) appear in the lower part of the Saved Search dropdown list.
End User		End-user saved searches are created at runtime using the query form Save button. They are available only to the user who created them. End-user saved searches appear in the top part of the Saved Search dropdown list.

End users can manage their saved searches by using the Personalize function in the **Saved Search** dropdown list to bring up the Personalize Saved Searches dialog, as shown in Figure 8–9.

End users can use the Personalize function to:

- Update a user-created saved search
- Delete a user-created saved search
- Set a saved search as the default
- Set a saved search to run automatically
- Set the saved search to show or hide from the **Saved Search** dropdown list

Figure 8–9 Personalize Saved Searches Dialog

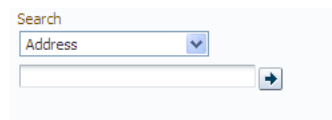
8.1.2 Quick Query Search Forms

A quick query search form is intended to be used in situations where a single search will suffice or as a starting point to evolve into a full query search. Both the query and quick query search forms are ADF Faces components. A quick query search form has one search criteria field with a dropdown list of the available searchable attributes from the associated data collection. Typically, the searchable attributes are all the attributes in the associated view collection. The user can search against the selected attribute or search against all the displayed attributes. The search criteria field type will automatically match the type of its corresponding attribute. An **Advanced** link built into the form offers you the option to create a managed bean to control switching from quick query to advanced mode query search form. For more information, see the "Using Query Components" chapter in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

You can configure the form to have a horizontal layout, as shown in Figure 8–10.

Figure 8–10 Quick Query Search Form in Horizontal Layout

You can also choose a vertical layout, as shown in Figure 8–11.

Figure 8–11 Quick Query Search Form in Vertical Layout

8.1.3 Filtered Table and Query-by-Example Searches

A filtered table can be created standalone or as the results table of a query or quick query search form. Filtered table searches are based on Query-by-Example and use the QBE text or date input field formats. The input validators are turned off to allow for entering characters such as > and <= to modify the search criteria. For example, you can enter >1500 as the search criteria for a number column. Wildcard characters may also be supported. If a column does not support QBE, the search criteria input field will not render for that column.

The filtered table search criteria input values are used to build the query WHERE clause with the AND operator. If the filtered table is associated with a query or quick query

search panel, the composite search criteria values are also combined to create the WHERE clause.

Figure 8–12 shows a query search form with a filtered results table. When the user enters a QBE search criteria, such as >200 for the id field, the query result is the AND of the query search criteria and the filtered table search criteria.

Figure 8–12 Query Search Form with Filtered Table

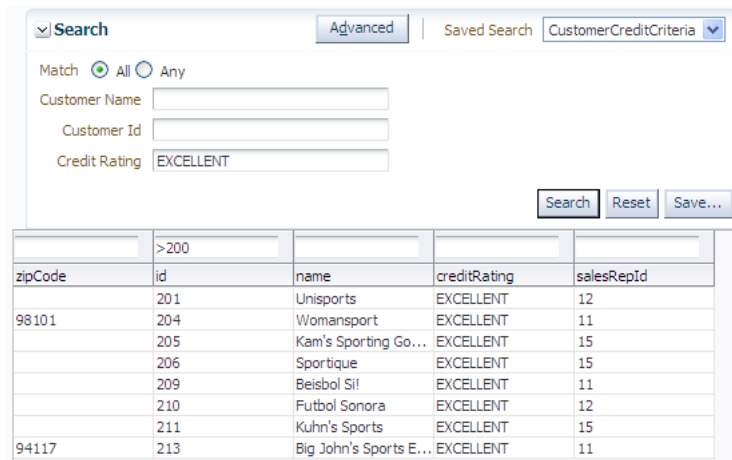


Table 8–2 lists the acceptable QBE search operators that can be used to modify the search value.

Table 8–2 Query-by-Example Search Criteria Operators

Operator	Description
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
AND	And
OR	Or

8.2 Creating Query Search Forms

You create a query search form by dropping a named criteria from the Data Controls panel onto a page. You have a choice of dropping only a search panel, dropping a search panel with a results table, or dropping a search panel with a tree table.

If you choose to drop the search panel with a table, you can select the filtering option in the dialog to turn the table into a filtered table.

Typically, you would drop a query search panel with the results table or tree table. JDeveloper will automatically create and associate a results table or tree table with the query panel.

If you drop a query panel by itself and want a results component or if you already have an existing component for displaying the results, you will need to match the query panel's ResultsComponentId with the results component's Id.

8.2.1 How to Create a Query Search Form with a Results Table or Tree Table

You create a search form by dragging and dropping a named criteria from the Data Controls panel onto the page. You have the option of having a results table or only the query panel.

Before you begin:

You should have created a accessor returned collection in the Data Control panel.

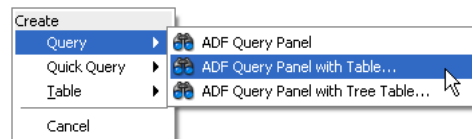
To create a query search form with a results table or tree table:

1. From the Data Controls panel, select the accessor returned collection and expand the **Named Criteria** node to display a list of named criteria.
2. Drag a named criteria item and drop it onto the page or onto the Structure window.

Note: Dropping **All Queriable Attributes** onto the page creates a search form with a search criteria field for each searchable attribute defined in the underlying collection. If you only want to create search criteria fields for some of those attributes, create a named criteria with the attributes you want.

3. From the context menu, choose **Create > Query > ADF Query Panel with Table** or **Create > Query > ADF Query Panel with Tree Table**, as shown in Figure 8–13.

Figure 8–13 Data Controls Panel with Query Context Menu



4. In the Edit Table Columns dialog, you can rearrange any column and select table options. If you choose the filtering option, the table will be a filtered table.

After you have created the form, you may want to set some of its properties or add custom functions. For more information on how to do this, see Section 8.3, "Setting Up Search Form Properties."

8.2.2 How to Create a Query Search Form and Add a Results Component Later

You create a search form by dragging and dropping a named criteria from the Data Controls panel onto the page. You have the option of having a results table or only the query panel.

Before you begin:

You should have created a accessor returned collection in the Data Control panel.

To create a query search form and add a results component in a separate step:

1. From the Data Controls panel, select the accessor returned collection and expand the **Named Criteria** node to display a list of named criteria.
2. Drag a named criteria item and drop it onto the page or onto the Structure window.

Note: Dropping **All Queriable Attributes** onto the page creates a search form with a search criteria field for each searchable attribute defined in the underlying collection. If you only want to create search criteria fields for some of those attributes, create a named criteria for the attributes you want.

3. Choose **Create > Query > ADF Query Panel** from the context menu, as shown in Figure 8–13.
4. If you do not already have a results component, then drag the accessor returned collection and drop it onto the page as a table, tree, or treetable component.
5. In the Property Inspector for the table, copy the value of the **Id** field.
6. In the Property Inspector for the query panel, paste the value of the table's ID into the query's **ResultsComponentId** field.

After you have created the search form, you may want to set some of its properties or add custom functions. See Section 8.3, "Setting Up Search Form Properties," for more information.

8.2.3 How to Persist Saved Searches into MDS

If you want saved searches to be persisted to MDS, you need to define the `/persdef` namespace in the `adf-config.xml` file. In addition, you need to perform the regular MDS configuration, such as specifying `metadatapath`. Example 8–3 shows an `adf-config.xml` file with the `/persdef` namespace defined.

Example 8–3 Sample adf-config.xml with /persdef Namespace

```
<persistence-config>
  <metadata-namespaces>
    <namespace path="/persdef" metadata-store-usage="mdsstore"/>
  </metadata-namespaces>
  <metadata-store-usages>
    <metadata-store-usage id="mdsstore" deploy-target="true"
      default-cust-store="true"/>
  </metadata-store-usage>
</metadata-store-usages>
</persistence-config>
```

In order for the added saved searches to be available the next time the user logs in, `cust-config` needs to be defined as part of the MDS configuration. For more information about setting `cust-config` and MDS, see "How to Create Customization Classes" of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

If you are also saving the layout of the results component, the application must have the ADF PageFlow Runtime and ADF Controller Runtime libraries installed. Set the project's technology scope to include **ADF Page Flow**.

8.2.4 What Happens When You Create a Query Form

When you drop a query search form onto a page, JDeveloper creates an `af:query` tag on the page. If you drop a query with table or tree table, then an `af:table` tag or `af:treeTable` tag will follow the `af:query` tag.

Under the `af:query` tag are several attributes that define the query properties. They include:

- The `id` attribute, which uniquely identifies the query.
- The `resultsComponentId` attribute, which identifies the component that will display the results of the query. Typically, this will be the table or tree table that was dropped onto the page together with the query. You can change this value to be the `id` of a different results component. For more information, see Section 8.2.2, "How to Create a Query Search Form and Add a Results Component Later."

In the page definition file, JDeveloper creates an iterator, accessorIterators, and a `searchRegion` entry in the `executables` section. Example 8–4 shows the sample code for a page definition file.

In the page definition file `executable` section:

- The iterator `RangeSize` property is set to a default value of 25. If you want a different page size, you must edit this value.
- The iterator `id` property is set to the root iterator. In the example, the value is set to `OrdersSessionEJBLocalIterator`
- The accessorIterator `Binds` property is set to the accessor. In the example, the value is set to `itemFindAll`.
- The accessorIterator `id` property is set to the accessor iterator. In the example, the value is set to `itemFindAllIterator`.
- The `searchRegion` `Criteria` property is set to `ItemCriteria`.
- The `searchRegion` `Binds` property is set to the search iterator. In the example, the value is set to `itemFindAllIterator`
- The `searchRegion` `id` property is set to `ItemCriteriaQuery`.

If the query was dropped onto the page with a table or tree, then in the page definition file `bindings` section, a tree element is added with the `IterBinding` property set to the search iterator. In this example, the value is set to `itemFindAllIterator`.

Example 8–4 Search Form Code in the Page Definition File

```
<executables>
  <variableIterator id="variables"/>
  <iterator Binds="root" RangeSize="25" DataControl="OrdersSessionEJBLocal"
    id="OrdersSessionEJBLocalIterator"/>
  <accessorIterator MasterBinding="OrdersSessionEJBLocalIterator"
    Binds="itemFindAll" RangeSize="25"
    DataControl="OrdersSessionEJBLocal"
    BeanClass="model.Item"
    id="itemFindAllIterator"/>
  <searchRegion Criteria="ItemCriteria"
    Customizer="oracle.jbo.uicli.binding.JUSearchBindingCustomizer"
    Binds="itemFindAllIterator"
    id="ItemCriteriaQuery"/>
</executables>
<bindings>
  <tree IterBinding="itemFindAllIterator" id="itemFindAll">
    <nodeDefinition DefName="model.Item">
      <AttrNames>
        <Item Value="itemId"/>
        <Item Value="ordId"/>
        <Item Value="price"/>
      </AttrNames>
    </nodeDefinition>
  </tree>
</bindings>
```

```
<Item Value="quantity"/>
<Item Value="quantityShipped"/>
</AttrNames>
</nodeDefinition>
</tree>
</bindings>
```

8.2.5 What Happens at Runtime: Search Forms

At runtime, the search form displays as a search panel on the page. The search panel will display in either basic mode or advanced mode.

After the user enters the search criteria and clicks **Search**, a query is executed and the results are displayed in the associated table, tree table, or component.

8.3 Setting Up Search Form Properties

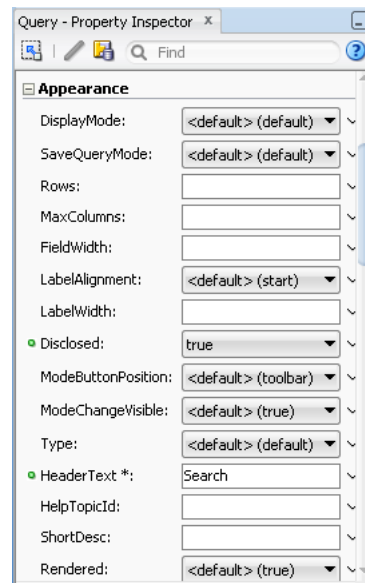
Search form properties that can be set after the query component has been added to the JSF page include:

- `id` of the results table or results component
- Show or hide of the basic/advanced button
- Position of the mode button
- Default, simple, or compact mode for display

8.3.1 How to Set Search Form Properties on the Query Component

After you have dropped the query search form onto a page, you can edit other form properties in the Property Inspector, as shown in Figure 8–14. Some of the common properties you may set are:

- Enabling or disabling the basic/advanced mode button
- Setting the ID of the query search form
- Setting the ID of the results component (for example, a results table)
- Selecting the default, simple, or compact mode for display

Figure 8–14 Property Inspector for a Query Component

One common option is to show or hide the basic/advanced button.

To enable or hide the basic/advanced button in the query form:

1. In the Structure window, double-click **af:query**.
2. In the Property Inspector, click the **Appearance** tab.
3. To enable the basic/advanced mode button, select **true** from the **ModeChangeVisible** field. To hide the basic/advance mode button, select **false** from the **ModeChangeVisible** field.

8.4 Creating Quick Query Search Forms

You can use quick query search forms to let users search on a single attribute of a collection. Quick query search form layout can be either horizontal or vertical. Because they occupy only a small area, quick query search forms can be placed in different areas of a page. You can create a managed bean to enable users to switch from a quick query to a full query search. For more information about switching from quick query to query using a managed bean, see the "Using Query Components" chapter in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

If you drop a quick query panel with a results table or tree, JDeveloper will automatically create the results table, as described in Section 8.4.1, "How to Create a Quick Query Search Form with a Results Table or Tree Table." If you drop a quick query panel by itself and subsequently want a results table or component or if you already have one, you will need to match the quick query `id` with the results component's `partialTrigger` value, as described in Section 8.4.2, "How to Create a Quick Query Search Form and Add a Results Component Later."

8.4.1 How to Create a Quick Query Search Form with a Results Table or Tree Table

You can create quick query searches using the full set of searchable attributes and simultaneously add a table or tree table as the results component.

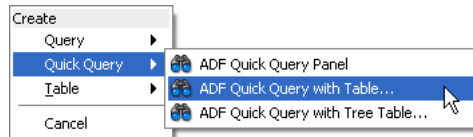
Before you begin:

Create an accessor returned collection in the Data Control panel.

To create a quick query search form with a results table:

1. From the Data Controls panel, select the accessor returned collection and expand the **Named Criteria** node to display a list of named criteria.
2. Drag a named criteria item and drop it onto the page or onto the Structure window.
3. From the context menu, choose **Create > Quick Query > ADF Quick Query Panel with Table** or **Create > Quick Query > ADF Quick Query Panel with Tree Table**, as shown in Figure 8–15.
4. In the Edit Table Columns dialog, you can rearrange any column and select table options. If you choose the filtering option, the table will be a filtered table.

Figure 8–15 Data Control Panel with Quick Query Context Menu



8.4.2 How to Create a Quick Query Search Form and Add a Results Component Later

You can create quick query searches using the full set of searchable attributes and add a table or tree table as the results component later.

Before you begin:

Create an accessor returned collection in the Data Control panel.

To create a quick query search form and add a results component in a separate step:

1. From the Data Controls panel, select the accessor returned collection and expand the **Named Criteria** node to display a list of named criteria.
2. Drag a named criteria item and drop it onto the page or onto the Structure window.
3. From the context menu, choose **Create > Quick Query > ADF Quick Query Panel**.
4. If you do not already have a results component, then drag the accessor returned collection and drop it onto the page as a table, tree, or treetable component.
5. In the Property Inspector for the quick query, copy the value of the **Id** field.
6. In the Property Inspector for the results component (for example, a table), paste or enter the value into the **PartialTriggers** field.

8.4.3 How to Set the Quick Query Layout Format

The default layout of the form is horizontal. You can change the layout option using the Property Inspector.

To set the layout:

1. In the Structure window, double-click **af:quickQuery**.

2. In the Property Inspector, on the Commons page, select the **Layout** property using the dropdown list to specify **default**, **horizontal**, or **vertical**.

8.4.4 What Happens When You Create a Quick Query Search Form

When you drop a quick query search form onto a page, JDeveloper creates an `af:quickQuery` tag. If you have dropped a quick query with table or tree table, then an `af:table` tag or `af:treeTable` tag is also added.

Under the `af:quickQuery` tag are several attributes and facets that define the quick query properties. Some of the tags are:

- The `id` attribute, which uniquely identifies the quick query. This value should be set to match the results table or component's `partialTriggers` value. JDeveloper will automatically assign these values when you drop a quick query with table or tree table. If you want to change to a different results component, see Section 8.4.2, "How to Create a Quick Query Search Form and Add a Results Component Later."
- The `layout` attribute, which specifies the quick query layout to be default, horizontal, or vertical.
- The `end` facet, which specifies the component to be used to display the **Advanced** link (that changes the mode from quick query to the query). For more information about creating this function, see the "Using Query Components" chapter of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

8.4.5 What Happens at Runtime: Quick Query

At runtime, the quick query search form displays a single search criteria field with a dropdown list of selectable search criteria items. If there is only one searchable criteria item, then the dropdown list box will not be rendered. An input component that is compatible with the selected search criteria type will be displayed, as shown in Table 8–3. For example, if the search criteria type is date, then `inputDate` will be rendered.

Table 8–3 Quick Query Search Criteria Field Components

Attribute Type	Rendered Component
DATE	<code>af:inputDate</code>
VARCHAR	<code>af:inputText</code>
NUMBER	<code>af:inputNumberSpinBox</code>

In addition, a **Search** button is rendered to the right of the input field. If the `end` facet is specified, then any components in the `end` facet are displayed. By default, the `end` facet contains an **Advanced** link.

8.5 Creating Standalone Filtered Search Tables

You use query search forms for complex searches, but you can also perform simple QBE searches using the filtered table. You can create a standalone ADF-filtered table without the associated search panel and perform searches using the QBE-style search criteria input fields. For more information about filtered tables, see Section 8.1.3, "Filtered Table and Query-by-Example Searches."

When creating a table, you can make almost any table a filtered table by selecting the filtering option if the option is enabled. There are three ways to create a standalone filtered table:

- You can drop a table onto a page from the Component Palette, bind it to a data collection, and set the filtering option. For more information, see the "Using Query Components" chapter of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.
- You can create a filtered table by dragging and dropping an accessor returned collection onto a page and setting the filtering option.
- You can also create a filtered table or a read-only filtered table by dropping **All Queriable Attributes** or a named criteria onto the page. The resulting filtered table will have a column for each searchable attribute and an input search field above each column.

You can set the QBE search criteria for each filterable column to be a case-sensitive or case-insensitive search using the `filterFeature` attribute of `af:column` in the `af:table` component. For more information, see the "Enable Filtering in Tables" section of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

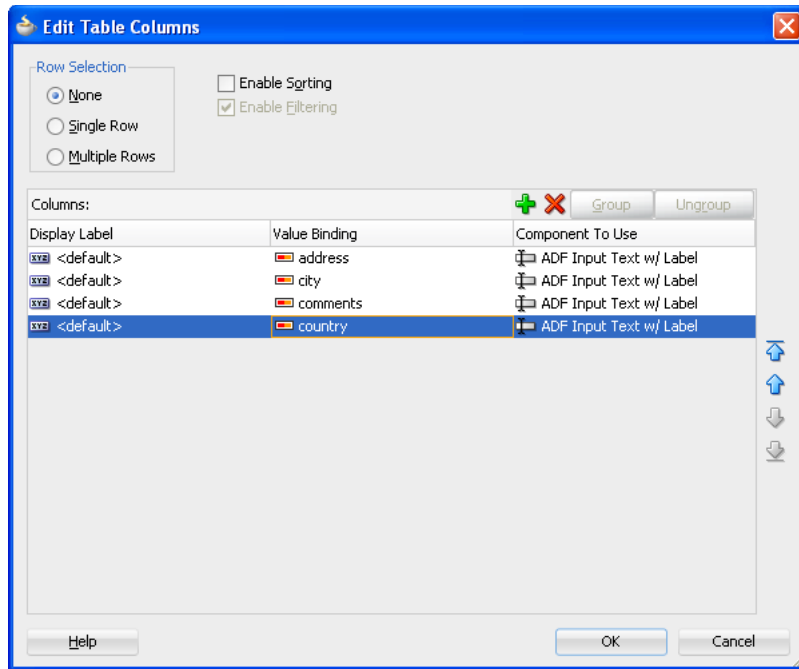
Before you begin:

Create an accessor returned collection in the Data Control panel.

To create a filtered table:

1. From the Data Controls panel, select the accessor returned collection and expand the **Named Criteria** node to display a list of all named criteria.
2. Drag a named criteria item and drop it onto the page or onto the Structure window.
3. From the context menu, choose **Create > Tables > ADF Filtered Table** or **Create > Tables > ADF Read-Only Filtered Table**.
4. In the Edit Table Columns dialog, you can rearrange any column and select table options. Because the table is created by JDeveloper during quick query creation, the filtering option is automatically enabled and not user-selectable, as shown in Figure 8-16.

Figure 8–16 Edit Table Columns Dialog for Filtered Table



Deploying an ADF Java EE Application

This chapter describes how to deploy Oracle ADF Java EE applications to a target application server. It describes how to create deployment profiles, how to create deployment descriptors, and how to load ADF runtime libraries. It includes instructions for running an application in the Integrated WebLogic Server as well as deploying to a standalone Oracle WebLogic Server or IBM WebSphere Application Server.

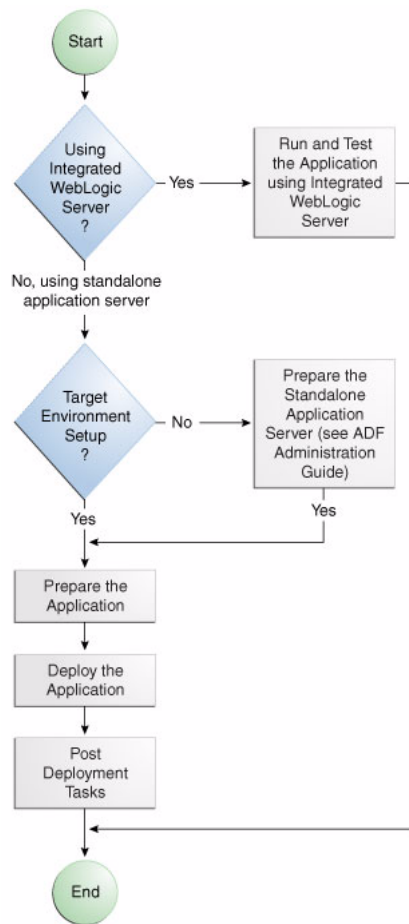
This chapter includes the following sections:

- Section 9.1, "Introduction to Deploying ADF Java EE Web Applications"
- Section 9.2, "Running a Java EE Application in Integrated WebLogic Server"
- Section 9.3, "Preparing the Application"
- Section 9.4, "Deploying the Application"
- Section 9.5, "Postdeployment Configuration"
- Section 9.6, "Testing the Application and Verifying Deployment"

9.1 Introduction to Deploying ADF Java EE Web Applications

Deployment is the process of packaging application files as an archive file and transferring it to a target application server. You can use JDeveloper to deploy Oracle ADF Java EE web applications directly to the application server (such as Oracle WebLogic Server or IBM WebSphere Application Server), or indirectly to an archive file as the deployment target, and then install this archive file to the target server. For application development, you can also use JDeveloper to run an application in Integrated WebLogic Server. JDeveloper supports deploying to server clusters. You cannot use JDeveloper to deploy to individual Managed Servers within a cluster.

Figure 9–1 shows the flow diagram that describes the overall deployment process. Note that preparing the target application server for deployment by installing the ADF runtime is described in the *Oracle Fusion Middleware Administrator's Guide for Oracle Application Development Framework*

Figure 9–1 Deployment Overview Flow Diagram

Note: Normally, you use JDeveloper to deploy applications for development and testing purposes. If you are deploying Oracle ADF Java EE web applications for production purposes, you can use Enterprise Manager or scripts to deploy to production-level application servers.

For more information about deployment to later-stage testing or production environments, see the *Oracle Fusion Middleware Administrator's Guide for Oracle Application Development Framework*.

ADF Java EE applications are based on standardized, modular components and can be deployed to the following application servers:

- Oracle WebLogic Server

Oracle WebLogic Server provides a complete set of services for those modules and handles many details of application behavior automatically, without requiring programming. For information about which versions of Oracle WebLogic Server are compatible with JDeveloper, see the certification information website at <http://www.oracle.com/technetwork/developer-tools/jdev/documentation/index.html>.

- IBM WebSphere Application Server

For information about which versions of IBM WebSphere are compatible, see the *Oracle Fusion Middleware Third-Party Application Server Guide*.

JSF applications that contain ADF Faces components have a few additional deployment requirements:

- ADF Faces requires Sun's JSF Reference Implementation 1.2 and MyFaces 1.0.8 (or later).

You can use JDeveloper to:

- Run applications in Integrated WebLogic Server

You can run and debug applications using Integrated WebLogic Server and then deploy to a standalone WebLogic Server or to WebSphere.

Integrated IBM WebSphere Application Server is not supported for this release.

- Deploy to Oracle WebLogic Server

You can deploy applications directly to the standalone application server by creating a connection to the server and choosing the name of that server as the deployment target.

- Deploy to an archive file

You can deploy applications indirectly by choosing an EAR file as the deployment target. The archive file can subsequently be installed on a target Oracle WebLogic Server instance.

9.1.1 Developing Applications with Integrated WebLogic Server

If you are developing an application in JDeveloper and you want to run the application in Integrated WebLogic Server, you do not need to perform the tasks required for deploying directly to Oracle WebLogic Server or to an archive file. JDeveloper has a default connection to Integrated WebLogic Server and does not require any deployment profiles or descriptors. Integrated WebLogic Server has a preconfigured domain that includes the ADF libraries, as well as the `-Djps.app.credential.override.allowed=true` setting, that are required to run Oracle ADF applications. You can run an application by choosing **Run** from the JDeveloper main menu.

9.1.2 Developing Applications for Standalone Application Server

Typically, you test and develop your application by running it in Integrated WebLogic Server. You can then test the application further by deploying it to standalone Oracle WebLogic Server in development mode to more closely simulate the production environment.

In general, you use JDeveloper to prepare the application or project for deployment by:

- Creating a connection to the target application server
- Creating deployment profiles (if necessary)
- Creating deployment descriptors (if necessary)
- Updating `application.xml` and `web.xml` to be compatible with Oracle WebLogic Server (if required)
- Enabling the application for Real User Experience Insight (RUEI) in `web.xml` (if desired)

- Migrating application-level security policy data to a domain-level security policy store
- Configuring the Oracle Single Sign-On (Oracle SSO) service and properties in the domain `jps-config.xml` file when you intend the web application to run using Oracle SSO

You must already have an installed application server. For Oracle WebLogic Server, you can use the Oracle 11g Installer or the Oracle Fusion Middleware 11g Application Developer Installer to install one. For other applications servers, follow the instructions in the applications server documentation to obtain and install the server.

You also must prepare the application server for ADF application deployment. For more information, see the "Preparing the Standalone Application Server for Deployment" section of the *Oracle Fusion Middleware Administrator's Guide for Oracle Application Development Framework*.

- Installing the ADF runtime into the application server installation:
 - For WebLogic Server
 - If you installed Oracle WebLogic Server together with JDeveloper using the Oracle 11g Installer for JDeveloper, the ADF runtime should already be installed.
 - If the ADF runtime is not installed and you want to use Oracle Enterprise Manager to manage standalone ADF applications (which are applications without Oracle SOA Suite or Oracle WebCenter components), use the Oracle Fusion Middleware 11g Application Developer Installer. This installer will install the necessary Oracle Enterprise Manager components into the Oracle WebLogic installation.
 - If the ADF runtime is not installed and you do not need to install Enterprise Manager, use the Oracle 11g Installer for JDeveloper.
 - For WebSphere
 - * Use the Oracle Fusion Middleware 11g Application Developer Installer to install the ADF runtime and the necessary Oracle Enterprise Manager components into the WebSphere installation. For information about installing WebSphere, see the *Oracle Fusion Middleware Third-Party Application Server Guide*.
- Extending Oracle WebLogic Server domains or WebSphere Cells to be ADF-compatible using the ADF runtime
- For WebLogic, setting the Oracle WebLogic Server credential store overwrite setting as required (`-Djps.app.credential.override.allowed=true` setting)
- Creating a global JDBC data source for applications that require a connection to a data source

After the application and the application server have been prepared, you can:

- Use JDeveloper to:
 - Directly deploy to the application server using the deployment profile and the application server connection.
 - Deploy to an EAR file using the deployment profile. For ADF applications, WAR and MAR files can be deployed only as part of an EAR file.

- Use Enterprise Manager, scripts, or the application's administration tool to deploy the EAR file created in JDeveloper. For more information, see the *Oracle Fusion Middleware Administrator's Guide for Oracle Application Development Framework*.

9.2 Running a Java EE Application in Integrated WebLogic Server

JDeveloper is installed with Integrated WebLogic Server, which you can use to test and develop your application. For most development purposes, Integrated WebLogic Server will suffice. When your application is ready to be tested, you can select the run target and then choose the **Run** command from the main menu.

Note: The first time you run an application in Integrated WebLogic Server, the Configure Default Domain dialog appears for you to define an administrative password for the new domain.

When you run the application target, JDeveloper detects the type of Java EE module to deploy based on artifacts in the projects and workspace. JDeveloper then creates an in-memory deployment profile for deploying the application to Integrated WebLogic Server. JDeveloper copies project and application workspace files to an "exploded EAR" directory structure. This file structure closely resembles the EAR file structure that you would have if you were to deploy the application to an EAR file. JDeveloper then follows the standard deployment procedures to register and deploy the "exploded EAR" files into Integrated WebLogic Server. The "exploded EAR" strategy reduces the performance overhead of packaging and unpacking an actual EAR file.

In summary, when you select the run target and run the application in Integrated WebLogic Server, JDeveloper:

- Detects the type of Java EE module to deploy based on the artifacts in the project and application
- Creates a deployment profile in memory
- Copies project and application files into a working directory with a file structure that would simulate the "exploded EAR" file of the application.
- Performs the deployment tasks to register and deploy the simulated EAR into Integrated WebLogic Server
- Automatically migrates identities, credentials, and policies

Later on, if you plan to deploy the application to a standalone WebLogic Server instance, you will need to migrate this security information. For more information, see Section 9.3.4, "How to Deploy Applications with ADF Security Enabled."

Note: JDeveloper ignores the deployment profiles that were created for the application when you run the application in Integrated WebLogic Server.

The application will run in the base domain in Integrated WebLogic Server. This base domain has the same configuration as a base domain in a standalone WebLogic Server instance. In other words, this base domain will be the same as if you had used the Oracle Fusion Middleware Configuration Wizard to create a base domain with the default options in a standalone WebLogic Server instance.

JDeveloper will extend this base domain with the necessary domain extension templates, based on the JDeveloper technology extensions. For example, if you have installed JDeveloper Studio, JDeveloper will automatically configure the Integrated WebLogic Server environment with the ADF runtime template (JRF Fusion Middleware runtime domain extension template).

You can explicitly create a default domain for Integrated WebLogic Server. You can use these domains to run and test your applications. Open the Application Server Navigator, right-click **IntegratedWebLogicServer** and choose **Create Default Domain**.

JDeveloper has a default connection to Integrated WebLogic Server. You do not need to create a connection to run an application. If you do want to manually create an application server connection to Integrated WebLogic Server, use the instructions in Section 9.3.1, "How to Create a Connection to the Target Application Server," to create the connection, selecting **Integrated Server** instead of **Standalone Server** in Step 2.

9.2.1 How to Run an Application in Integrated WebLogic Server

You can test an application by running it in Integrated WebLogic Server. You can also set breakpoints and then run the application within the ADF Declarative Debugger.

To run an application in Integrated WebLogic Server:

1. In the Application Navigator, select the project, unbounded task flow, JSF page, or file as the run target.
2. Right-click the run target and choose **Run** or **Debug**.
3. The Configure Default Domain dialog displays the first time you run your application and start a new domain in Integrated WebLogic Server. Use the dialog to define an administrator password for the new domain. Passwords you enter can be eight characters or more and must have a numeric character.

9.2.2 How to Run an Application with Metadata in Integrated WebLogic Server

When an application is running in Integrated WebLogic Server, the metadata archive (MAR) profile itself will not be deployed to a repository, but a simulated Oracle Metadata Services (MDS) repository will be configured for the application that reflects the metadata information contained in the MAR. This metadata information is simulated, and the application runs based on this location in source control.

By default, only the customizations in ADF view and ADF Model are included in the MAR. If the Java EE application has customizations in other directories, you must create a custom MAR profile that includes these directories.

Any customizations or documents created by the application that are not configured to be stored in other MDS repositories are written to this simulated MDS repository directory. For example, if you customize an object, the customization is written to the simulated MDS repository. If you execute code that creates a new metadata object, then this new metadata object is also written to the same location in the simulated MDS repository. You can keep the default location for this directory (`ORACLE_HOME\jdeveloper\systemXX.XX\o.mds.dt\adrs\Application\AutoGeneratedMar\mds_adrs_writedir`), or you can set it to a different directory. You also have the option to preserve this directory across different application runs, or to delete it before each application run.

If your workspace has different working sets, only the metadata from the projects defined in the working set and their dependent projects will be included in the MAR. You can view and change a project's dependencies by right-clicking the project in the

Application Navigator, choosing **Project Properties**, and then selecting **Dependencies**. For instance, an application may have several projects but `workingsetA` is defined to be `viewController2` and `viewController5`; and `viewController5` has a dependency on `modelproject1`. When you run or debug `workingsetA`, only the metadata for `viewController2`, `viewController5`, and `modelproject1` will be included in the MAR for deployment.

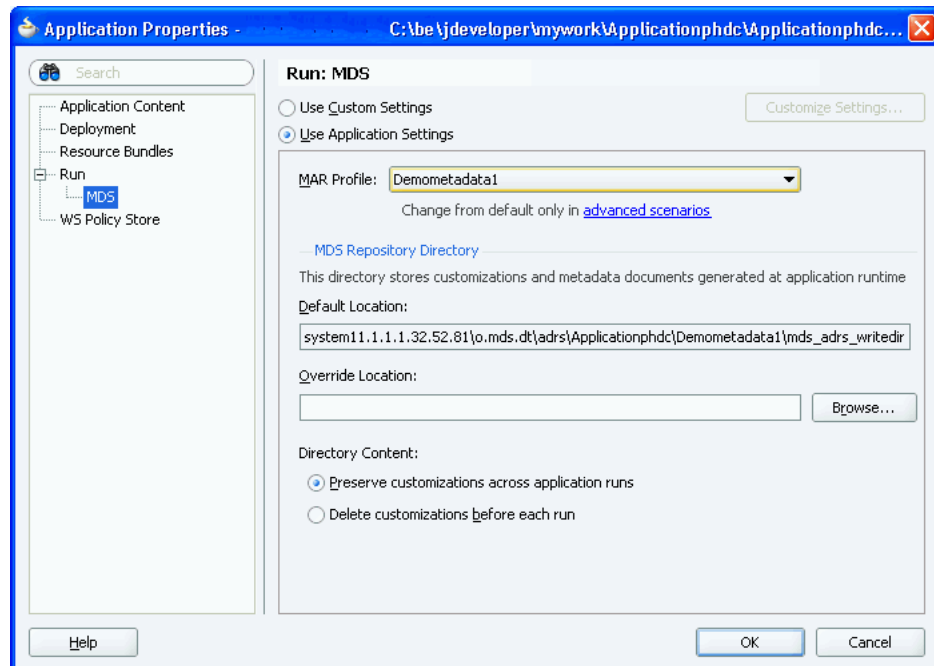
There should already be a MAR profile, either generated automatically by JDeveloper, or manually generated by a user.

To deploy the MAR profile to Integrated WebLogic Server:

1. In the Application Navigator, right-click the application and choose **Application Properties**.
2. In the Application Properties dialog, expand **Run** and choose **MDS**.
3. In the Run MDS page:
 - Select the MAR profile from the **MAR Profile** dropdown list
 - Enter a directory path in **Override Location** if you want to customize the location of the simulated MDS repository.
 - Select the **Directory Content** option. You can choose to preserve the customizations across application runs or delete customizations before each run.

Select the MAR profile from the **MAR Profile** dropdown list. Figure 9–2 shows **Demometadata1** selected as the MAR profile.

Figure 9–2 Setting the Run MDS options

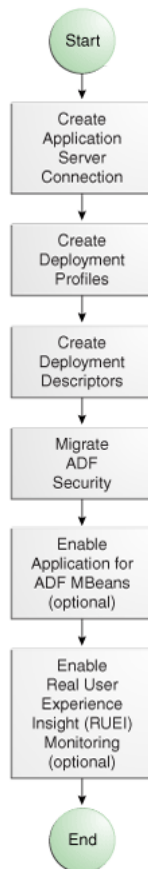


9.3 Preparing the Application

Before you deploy an ADF application to a standalone Oracle WebLogic Server instance, you must perform prerequisite tasks within JDeveloper to prepare the application for deployment.

Figure 9–3 show the process flow to prepare the application for deployment. After the application has been prepared and the application server has been prepared as described in the *Oracle Fusion Middleware Administrator's Guide for Oracle Application Development Framework*, you can proceed to deploy the application as described in Section 9.4, "Deploying the Application."

Figure 9–3 Preparing the Application for Deployment Flow Diagram



9.3.1 How to Create a Connection to the Target Application Server

You can deploy applications to Oracle WebLogic Server via JDeveloper application server connections. If your application involves customization using MDS, you should register your MDS repository with the Oracle WebLogic Server domain.

For more information about registering MDS, see the *Oracle Application Server Administrator's Guide*.

To create a connection to an application server:

1. Launch the Application Server Connection wizard.

You can:

- In the Application Server Navigator, right-click **Application Servers** and choose **New Application Server Connection**.
 - In the New Gallery, expand **General**, select **Connections** and then **Application Server Connection**, and click **OK**.
 - In the Resource Palette, choose **New > New Connections > Application Server**.
2. In the Create AppServer Connection dialog Usage page, select **Standalone Server**.
 3. In the Name and Type page, enter a connection name.
 4. In the **Connection Type** dropdown list, choose:
 - **WebLogic 10.3** to create a connection to Oracle WebLogic Server
 - **WebSphere Server 7.x** to create a connection to IBM WebSphere Server
 5. Click **Next**.
 6. On the Authentication page, enter a user name and password for the administrative user authorized to access the application server.
 7. Click **Next**.
 8. On the Configuration page, enter the information for your server:

For WebLogic:

- The Oracle WebLogic host name is the name of the WebLogic Server instance containing the TCP/IP DNS where your application (.jar, .war, .ear) will be deployed.
- In the **Port** field, enter a port number for the Oracle WebLogic Server instance on which your application (.jar, .war, .ear) will be deployed.

If you don't specify a port, the port number defaults to 7001.

- In the **SSL Port** field, enter an SSL port number for the Oracle WebLogic Server instance on which your application (.jar, .war, .ear) will be deployed.

Specifying an SSL port is optional. It is required only if you want to ensure a secure connection for deployment.

If you don't specify an SSL port, the port number defaults to 7002.

- Select **Always Use SSL** to connect to the Oracle WebLogic Server instance using the SSL port.
- Optionally enter a **WebLogic Domain** only if Oracle WebLogic Server is configured to distinguish nonadministrative server nodes by name.

For WebSphere:

- In the **Host Name** field, enter the name of the WebSphere server containing the TCP/IP DNS where your Java EE applications (.jar, .war, .ear) are deployed. If no name is entered, the name defaults to localhost.
- In the **SOAP Connector Port** field, enter the port number. The host name and port are used to connect to the server for deployment. The default SOAP connector port is 8879.
- In the **Server Name** field, enter the name assigned to the target application server for this connection.

- In the **Target Node** field, enter the name of the target node for this connection. A node is a grouping of Managed Servers. The default is `machineNode01`, where `machine` is the name of the machine the node resides on
 - In the **Target Cell** field, enter the name of the target cell for this connection. A cell is a group of processes that host runtime components. The default is `machineNode01Cell`, where `machine` is the name of the machine the node resides on.
 - In the **Wsadmin script location** field, enter, or browse to, the location of the `wsadmin` script file to be used to define the system login configuration for your IBM WebSphere application server connection. The default location is `websphere-home/bin/wsadmin.sh` for Unix/Linux and `websphere-home/bin/wsadmin.bat` for Windows.
9. Click **Next**.
 10. If you have chosen WebSphere, the JMX page appears. On the JMX page, enter the JMX information (optional):

Note: JMX configuration is optional and is not required for connecting to the WebSphere Application Server. JMX is only needed for deploying SOA applications.

- Select **Enable JMX** for this connection to enable JMX.
 - In the **RMI Port** field, enter the port number of WebSphere's RMI connector port. The default is 2809.
 - In the **WebSphere Runtime Jars Location** field, enter or browse to the location of the WebSphere runtime JARs.
 - In the **WebSphere Properties Location (for secure MBEAN access)** field, enter or browse to the location of the file that contains the properties for the security configuration and the mbeans that are enabled. This field is optional.
11. Click **Next**.
 12. If the SSL Signer Exchange Prompt dialog appears, click **Y**.
 13. On the Test page, click **Test Connection** to test the connection.

JDeveloper performs several types of connections tests. The JSR-88 test must pass for the application to be deployable. If the test fails, return to the previous pages of the wizard to fix the configuration.
 14. Click **Finish**.

9.3.2 How to Create Deployment Profiles

A *deployment profile* defines the way the application is packaged into the archive that will be deployed to the target environment. The deployment profile:

- Specifies the format and contents of the archive file that will be created
- Lists the source files, deployment descriptors, and other auxiliary files that will be packaged
- Describes the type and name of the archive file to be created
- Highlights dependency information, platform-specific instructions, and other information

You need a WAR deployment profile for each web view-controller project that you want to deploy in your application. If you want to package seeded customizations or place base metadata in the MDS repository, you need an application-level metadata archive (MAR) deployment profile as well. For more information about seeded customizations, see the "Customizing Applications with MDS" section of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*. If the application has customization classes, you need a JAR file for those classes and you need to add that JAR when you create the EAR file. If you are using EJB, you need an EJB JAR profile. Finally, you need an application-level EAR deployment profile and you must select the projects you want to include from a list, such as WAR and MAR profiles and customization classes JAR files. When the application is deployed, the EAR file will include all the projects that were selected in the deployment profile.

For Oracle ADF applications, you can deploy the application only as an EAR file. The WAR and MAR files that are part of the application should be included in the EAR file when you create the deployment profile.

9.3.2.1 Creating a WAR Deployment Profile

You will need to create a WAR deployment profile for each web-based project you want to package into the application. Typically, the WAR profile will include the dependent data model projects it requires.

To create WAR deployment profiles for an application:

1. In the Application Navigator, right-click the web project that you want to deploy and choose **New**.

You will create a WAR profile for each web project.

2. In the New Gallery, expand **General**, select **Deployment Profiles** and then **WAR File**, and click **OK**.

If you don't see **Deployment Profiles** in the **Categories** tree, click the **All Features** tab.

3. In the Create Deployment Profile -- WAR File dialog, enter a name for the project deployment profile and click **OK**.
4. In the Edit WAR Deployment Profile Properties dialog, choose items in the left pane to open dialog pages in the right pane. Configure the profile by setting property values in the pages of the dialog.
 - If you have customization classes in your application, they must be loaded from the EAR-level application class loader and not from the WAR. You will later add these customization classes to the EAR.

By default, customization classes are added to the data model project's WAR class path. So for each WAR, you must exclude the customization classes.

If you created your customization classes in an extension project of the application, be sure to deselect any customization class archive on the **Library Dependencies** page of the WAR deployment profile for each user interface project.

If you created your customization classes in the data model project of the application, deselect any customization classes on the **File Groups > WEB-INF/classes > Filters** page of the WAR deployment profile for each user interface project. If you are using a `customization.properties` file, it should also be deselected.

- You might also want to change the Java EE web context root setting (choose **General** in the left pane).

By default, when **Use Project's Java EE Web Context Root** is selected, the associated value is set to the project name, for example, `Application1-Project1-context-root`. You need to change this if you want users to use a different name to access the application.

If you are using custom JAAS LoginModule for authentication with JAZN, the context root name also defines the application name that is used to look up the JAAS LoginModule.

5. Click **OK** to exit the Edit WAR Deployment Profile Properties dialog.
6. Click **OK** again to exit the Project Properties dialog.
7. Repeat Steps 1 through 7 for all web projects that you want to deploy.

9.3.2.2 Creating a MAR Deployment Profile

If you have seeded customizations or base metadata that you want to place in the MDS repository, you need to create a MAR deployment profile.

The namespace configuration under `<mds-config>` for MAR content in the `adf-config.xml` file is generated based on your selections in the MAR Deployment Profile Properties dialog.

Although uncommon, an enterprise application (packaged in an EAR) can contain multiple web application projects (packaged in multiple WARs), but the metadata for all these web applications will be packaged into a single metadata archive (MAR). The metadata contributed by each of these individual web applications can be global (available for all the web applications) or local to that particular web application.

To avoid name conflicts for metadata with global scope, make sure that all metadata objects and elements have unique names across all the web application projects that form part of the enterprise application.

To avoid name conflicts and to ensure that the metadata for a particular web application remains local to that application, you can define a `web-app-root` for that web application project.

The `web-app-root` is an element in the `adf-settings.xml` file for a web application project. The `adf-settings.xml` file should be kept in the `META-INF` directory under the `public_html` directory for the web project. Example 9-1 shows the contents of a sample `adf-settings.xml` file.

Example 9-1 *web-app-root Element in the adf-settings.xml File*

```
<?xml version="1.0" encoding="UTF-8" ?>
<adf-settings xmlns="http://xmlns.oracle.com/adf/settings"
  xmlns:wap="http://xmlns.oracle.com/adf/share/http/config">
  <wap:adf-web-config xmlns="http://xmlns.oracle.com/adf/share/http/config">
    <web-app-root rootName="order"/>
  </wap:adf-web-config>
</adf-settings>
```

In this example, the `adf-settings.xml` file has a `web-app-root` element that defines `rootName` as `order`.

If your enterprise application has only one web application project, there is no need to define a `web-app-root` element. If your enterprise application has multiple web application projects, you should supply a `web-app-root` for all the web applications

except one, without which the deployment will fail. For example, if you have `web-application1`, `web-application2`, and `web-application3`, two of these web application projects must define a `web-app-root` to preclude any name conflicts.

JDeveloper creates an auto-generated MAR when the **Enable User Customizations** and **Across Sessions using MDS** options are selected in the ADF View page of the Project Properties dialog or when you explicitly specify the deployment target directory in the `adf-config.xml` file.

By default, only the customizations in ADF view and ADF Model are included in the MAR. If the Java EE application has customizations in other directories, you must create a custom MAR profile that includes those directories.

Before you begin:

Create an MDS repository for your customization requirements to deploy metadata using the MAR deployment profile.

To create a MAR deployment profile:

1. In the Application Navigator, right-click the application and choose **New**.

You will create a MAR profile if you want to include customizations.

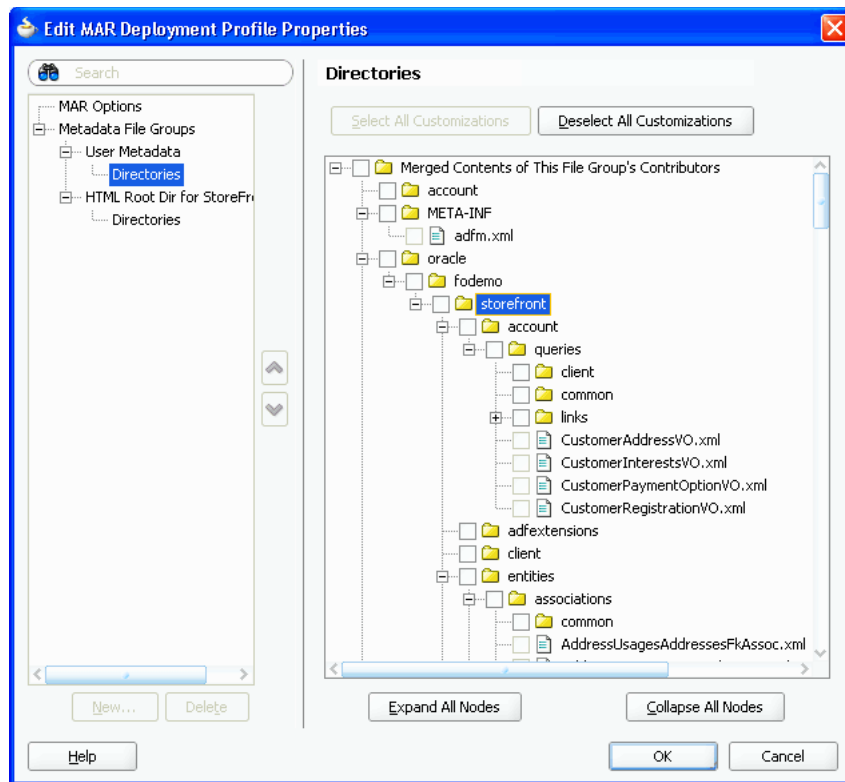
2. In the New Gallery, expand **General**, select **Deployment Profiles** and then **MAR File**, and click **OK**.

If you don't see **Deployment Profiles** in the **Categories** tree, click the **All Features** tab.

3. In the Create Deployment Profile -- MAR File dialog, enter a name for the MAR deployment profile and click **OK**.
4. In the Edit MAR Deployment Profile Properties dialog, choose items in the left pane to open dialog pages in the right pane.

Figure 9-4 shows a sample **User Metadata** directory tree.

Figure 9–4 Selecting Items for the MAR Deployment Profiles



Note the following important points:

- To include all customizations, you need only create a file group with the desired directories.
- To include files from other than ADF Model and ADF view, create a new file group under **User Metadata** with the desired directories and explicitly select the required content in the Directories page.
- ADF Model and ADF view directories are added by default. No further action is required to package the ADF Model and ADF view customizations into the MAR. ADF view content is added to **HTML Root dir**, while ADF Model content is added to **User Metadata**. If your application has other customization directories, such as from an EJB project, you must add those directories.
- To include the base metadata in the MDS repository, you need to explicitly select these directories in the dialog.

When you select the base document to be included in the MAR, you also select specific packages. When you select one package, all the documents (including subpackages) under that package will be used. When you select a package, you cannot deselect individual items under that package.

- If a dependent ADF library JAR for the project contains seeded customizations, they will automatically be added to the MAR during MAR packaging. They will not appear in the MAR profile.
- If ADF Library customizations were created in the context of the consuming project, those customizations would appear in the MAR profile dialog by default.

5. Click **OK** to exit the Edit MAR Deployment Profile Properties dialog.
6. Click **OK** again to exit the Application Properties dialog.

9.3.2.3 Creating an EJB JAR Deployment Profile

If you are using an EJB module in the data model project, you need to create an EJB JAR deployment profile.

Before you begin:

Create a data model project that has an EJB module.

To create an EJB JAR deployment profile for an application:

1. In the Application Navigator, right-click the web project that you want to deploy and choose **New**.
2. In the New Gallery, expand **General**, select **Deployment Profiles** and then **EJB JAR File**, and click **OK**.

If you don't see **Deployment Profiles** in the **Categories** tree, click the **All Features** tab.

3. In the Create Deployment Profile -- EJB JAR File dialog, enter a name for the deployment profile and click **OK**.
4. In the Edit EJB JAR Deployment Profile Properties dialog, choose items in the left pane to open dialog pages in the right pane. Configure the profile by setting property values in the pages of the dialog.
5. Click **OK** to exit the Edit EJB JAR Deployment Profile Properties dialog.
6. Click **OK** again to exit the Project Properties dialog.

9.3.2.4 Creating an Application-Level EAR Deployment Profile

The EAR file contains all the necessary application artifacts for the application to run in the application server. For more information about the EAR file, see Section 9.4.6, "What You May Need to Know About EAR Files and Packaging."

Before you begin:

- Add classes into a JAR file, as described in Section 9.3.2.7, "Adding Customization Classes into a JAR."
- Create the WAR deployment profiles, as described in Section 9.3.2.1, "Creating a WAR Deployment Profile."

To create an EAR deployment profile for an application:

1. In the Application Navigator, right-click the application and choose **New**.
You will create an EAR profile for the application.

2. In the New Gallery, expand **General**, select **Deployment Profiles** and then **EAR File**, and click **OK**.

If you don't see **Deployment Profiles** in the **Categories** tree, click the **All Features** tab.

3. In the Create Deployment Profile -- EAR File dialog, enter a name for the application deployment profile and click **OK**.

4. In the Edit EAR Deployment Profile Properties dialog, choose items in the left pane to open dialog pages in the right pane. Configure the profile by setting property values in the pages of the dialog.

Be sure that you:

- Select **Application Assembly** and then in the **Java EE Modules** list, select all the project profiles that you want to include in the deployment, including any **WAR** or **MAR** profiles.
- Select **Platform**, and select the application server you are deploying to, and then select the target application connection from the **Target Connection** dropdown list.

Note: If you are using a custom JAAS LoginModule for authentication with JAZN, the context root name also defines the application name that is used to look up the JAAS LoginModule.

5. If you have customization classes in your application, configure these classes so that they load from the EAR-level application class loader.
 - a. In the Edit EAR Deployment Profile Properties dialog, select **Application Assembly**.
 - b. Select the JAR deployment profile that contains the customization classes, and enter `lib` in the **Path in EAR** field at the bottom of the dialog.

Note: You should have created this JAR as described in Section 9.3.2.7, "Adding Customization Classes into a JAR."

The JAR file containing the customization classes is added to the EAR file's `lib` directory.

Note: If you have customization classes in your application, you must also make sure they are not loaded from the WAR. By default, customization classes that are added to the data model project's Libraries & Classpath are packaged to the WAR class path.

To make sure customization classes from an extension project are not duplicated in the WAR, be sure to deselect any customization class archive on the **Library Dependencies** page for the WAR.

If you created your customization classes in the data model project of the consuming application, deselect any customization classes on the **File Groups > WEB-INF/classes > Filters** page for the WAR.

6. Click **OK** to exit the Deployment Profile Properties dialog.
7. Click **OK** again to exit the Application Properties dialog.

9.3.2.5 Delivering Customization Classes as a Shared Library

As an alternative to adding your customization classes to the EAR, as described in Section 9.3.2.4, "Creating an Application-Level EAR Deployment Profile," you can also include the customization classes in the consuming application as a shared library.

Before you begin:

With the application containing the customization classes open in JDeveloper in the Studio Developer role, use the procedure described in Section 9.3.2.7, "Adding Customization Classes into a JAR," making sure that you select **Shared Library JAR File** as the type of archive to create.

Note: This procedure describes how to create and use a shared library if you are deploying to Oracle Weblogic Server.

To create and use a shared library for your customization classes:

1. In the Application Navigator, right-click the customization classes project, and choose **Deploy > deployment-profile**.
2. In the Deploy wizard, select **Deploy to a Weblogic Application Server** and click **Next**.
3. Select the appropriate application server, and click **Finish**.
This makes the shared library available on the application server. You must now add a reference to the shared library from the consuming application.
4. Open the application you want to customize in JDeveloper in the Studio Developer role.
5. In the Application Resources panel of the Application Navigator, double-click the `weblogic-application.xml` file to open it.
6. In the overview editor, click the Libraries tab.
7. In the **Shared Library References** section, click the add icon.
8. In the **Library Name** field of the newly created row in the **Shared Library References** table, enter the name of the customization classes shared library you deployed, and save your changes.

9.3.2.6 Viewing and Changing Deployment Profile Properties

After you have created a deployment profile, you can view and change its properties.

To view, edit, or delete a project's deployment profile:

1. In the Application Navigator, right-click the project and choose **Project Properties**.
2. In the Project Properties dialog, click **Deployment**.
The **Deployment Profiles** list displays all profiles currently defined for the project.
3. In the list, select a deployment profile.
4. To edit or delete a deployment profile, click **Edit** or **Delete**.

9.3.2.7 Adding Customization Classes into a JAR

If your application has customization classes, create a JAR that contains only these customization classes. When you create your EAR, you can add the JAR to the EAR assembly. And when you create WAR profiles for your web projects, you must make sure they don't include the customization classes JAR.

Before you begin:

Make sure that your project has customization classes. You do not need to perform this procedure if the application does not have customization classes. For more

information about customization classes, see the "How to Create Customization Classes" section of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

To add customization classes into a JAR:

1. In the Application Navigator, right-click the data model project that contains the customization classes you want to create a JAR for, and choose **New**.
2. In the New Gallery, expand **General**, select **Deployment Profiles** and then **JAR File**, and click **OK**.

Alternatively, if you want to create a shared library, select **Shared Library JAR File** from the list of profile types, and click **OK**.

Note: If you don't see **Deployment Profiles** in the **Categories** tree, click the **All Features** tab.

3. In the Create Deployment Profile -- JAR File dialog, enter a name for the project deployment profile (for example, CCArchive) and click **OK**.
4. In the Edit JAR Deployment Profile Properties dialog, select **JAR Options**.
5. Enter the location for the JAR file.
6. Expand **Files Groups > Project Output > Filters**.
7. In the **Files** tab, select the customization classes you want to add to the JAR file. If you are using a `customization.properties` file, it needs to be in the same class loader as the JAR file. You can select the `customization.properties` file to package it along with the customization classes in the same JAR.
8. Click **OK** to exit the Edit JAR Deployment Profile Properties dialog.
9. Click **OK** again to exit the Project Properties dialog.
10. In the Application Navigator, right-click the project containing the JAR deployment profile, and choose **Deploy > deployment profile > to JAR file**.

Note: If this is the first time you deploy to a JAR from this deployment profile, you choose **Deploy > deployment profile** and select **Deploy to JAR** in the wizard.

9.3.3 How to Create and Edit Deployment Descriptors

Deployment descriptors are server configuration files that define the configuration of an application for deployment and that are deployed with the Java EE application as needed. The deployment descriptors that a project requires depend on the technologies the project uses and on the type of the target application server. Deployment descriptors are XML files that can be created and edited as source files, but for most descriptor types, JDeveloper provides dialogs or an overview editor that you can use to view and set properties. If you cannot edit these files declaratively, JDeveloper opens the XML file in the source editor for you to edit its contents.

You can specify deployment descriptors that are specific to your target Oracle WebLogic application server instance.

In addition to the standard Java EE deployment descriptors (for example, `application.xml` and `web.xml`), you can also have deployment descriptors that

are specific to your target application server. For example, if you are deploying to Oracle WebLogic Server, you can also have `weblogic.xml`, `weblogic-application.xml`, and `weblogic-ear-jar.xml`.

Make sure that the application EAR file includes a `weblogic-application.xml` file that contains a reference to `adf.oracle.domain`, and that it includes an `ADFApplicationLifecycleListener` to clean up application resources between deployment and undeployment actions. Example 9-2 shows a sample `weblogic-application.xml` file.

Example 9-2 Sample `weblogic-application.xml`

```
<weblogic-application xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.bea.com/ns/weblogic/weblogic-application.xsd"
  xmlns="http://www.bea.com/ns/weblogic/weblogic-application">
  <listener>
    <listener-class>oracle.adf.share.weblogic.listeners.
      ADFApplicationLifecycleListener</listener-class>
  </listener>
  <listener>
    <listener-class>oracle.mds.lcm.weblogic.WLLifecycleListener</listener-class>
  </listener>
  <library-ref>
    <library-name>adf.oracle.domain</library-name>
    <implementation-version>
    </implementation-version>
  </library-ref>
</weblogic-application>
```

If you are deploying web services, you may need to modify your `weblogic-application.xml` and `web.xml` files as described in the "How to Deploy Web Services to Oracle WebLogic Server" section of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

If you want to enable the application for Real User Experience Insight (RUEI) monitoring, you must add a parameter to the `web.xml` file, as described in Section 9.3.3.5, "Enabling the Application for Real User Experience Insight."

During deployment, the application's security properties are written to the `weblogic-application.xml` file to be deployed with the application in the EAR file.

Because Oracle WebLogic Server runs on Java EE 1.5, you may need to modify the `application.xml` and `web.xml` files to be compatible with the server.

9.3.3.1 Creating Deployment Descriptors

JDeveloper automatically creates many of the required deployment descriptors for you. If they are not present, or if you need to create additional descriptors, you can explicitly create them.

Before you begin:

Check to see whether JDeveloper has already generated deployment descriptors.

To create a deployment descriptor:

1. In the Application Navigator, right-click the project for which you want to create a descriptor and choose **New**.

2. In the New Gallery, expand **General**, select **Deployment Descriptors** and then a descriptor type, and click **OK**.

If you can't find the item you want, make sure that you chose the correct project, and then choose the **All Features** tab or use the **Search** field to find the descriptor. If the item is not enabled, check to make sure that the project does not already have a descriptor of that type. A project is allowed only one instance of a descriptor.

JDeveloper starts the Create Deployment Descriptor wizard and then opens the file in the overview or source editor, depending on the type of deployment descriptor you choose.

Note: For EAR files, do not create more than one deployment descriptor per application or workspace. These files are assigned to projects, but have application workspace scope. If multiple projects in an application have the same deployment descriptor, the one belonging to the launched project will supersede the others. This restriction applies to `application.xml`, `weblogic-jdbc.xml`, `jazn-data.xml`, and `weblogic.xml`.

The best place to create an application-level descriptor is in the Descriptors node of the Application Resources panel in the Application Navigator. This ensures that the application is created with the correct descriptors.

9.3.3.2 Viewing or Modifying Deployment Descriptor Properties

After you have created a deployment descriptor, you can change its properties by using JDeveloper dialogs or by editing the file in the source editor. The deployment descriptor is an XML file (for example, `application.xml`) typically located under the Application Sources node.

To view or change deployment descriptor properties:

1. In the Application Navigator or in the Application Resources panel, double-click the deployment descriptor.
2. In the editor window, select either the **Overview tab** or the **Source tab**, and configure the descriptor by setting property values.

If the overview editor is not available, JDeveloper opens the file in the source editor.

9.3.3.3 Configuring the application.xml File for WebLogic Compatibility

You may need to configure your `application.xml` file to be compliant with Java EE 1.5.

Note: Typically, your project has an `application.xml` file that is compatible and you would not need to perform this procedure.

To configure the application.xml file:

1. In the Application Navigator, right-click the project and choose **New**.
2. In the New Gallery, expand **General**, select **Deployment Descriptors** and then **Java EE Deployment Descriptor Wizard**, and click **OK**.

3. In the Select Descriptor page of the Create Java EE Deployment Descriptor dialog, select **application.xml** and click **Next**.
4. In the Select Version page, select **5.0** and click **Next**.
5. In the Summary page, click **Finish**.
6. Edit the `application.xml` file with the appropriate values.

9.3.3.4 Configuring the `web.xml` File for WebLogic Compatibility

You may need to configure your `web.xml` file to be compliant with Java EE 1.5 (which corresponds to servlet 2.5 and JSP 1.2).

Note: Typically, your project has a `web.xml` file that is compatible and you would not need to perform this procedure. JDeveloper creates a starter `web.xml` file when you create a project.

To configure the `web.xml` file:

1. In the Application Navigator, right-click the project and choose **New**.
2. In the New Gallery, expand **General**, select **Deployment Descriptors** and then **Java EE Deployment Descriptor Wizard**, and click **OK**.
3. In the Select Descriptor page of the Create Java EE Deployment Descriptor dialog, select **web.xml** and click **Next**.
4. In the Select Version page, select **2.5** and click **Next**.
5. In the Summary page, click **Finish**.

9.3.3.5 Enabling the Application for Real User Experience Insight

Real User Experience Insight (RUEI) is a web-based utility to report on real-user traffic requested by, and generated from, your network. It measures the response times of pages and transactions at the most critical points in the network infrastructure. Session diagnostics allow you to perform root-cause analysis.

RUEI enables you to view server and network times based on the real-user experience, to monitor your Key Performance Indicators (KPIs) and Service Level Agreements (SLAs), and to trigger alert notifications on incidents that violate their defined targets. You can implement checks on page content, site errors, and the functional requirements of transactions. Using this information, you can verify your business and technical operations. You can also set custom alerts on the availability, throughput, and traffic of all items identified in RUEI.

For more information about RUEI, see the *Oracle Real User Experience Insight User's Guide* at http://download.oracle.com/docs/cd/E16339_01/doc.60/e16359/toc.htm.

You must enable an application for RUEI by adding the `context-param` tag to the `web.xml` file shown in Example 9-3.

Example 9-3 Enabling RUEI Monitoring for an Application in `web.xml`

```
<context-param>
  <description>This parameter notifies ADF Faces that the
    ExecutionContextProvider service provider is enabled.
    When enabled, this will start monitoring and aggregating
    user activity information for the client initiated
    requests. By default this param is not set or is false.
```

```

</description>
<param-name>
    oracle.adf.view.faces.context.ENABLE_ADF_EXECUTION_CONTEXT_PROVIDER
</param-name>
<param-value>true</param-value>
</context-param>

```

9.3.4 How to Deploy Applications with ADF Security Enabled

If you are developing an application in JDeveloper using Integrated WebLogic Server, application security deployment properties are configured by default, which means that the application and security credentials and policies will be overwritten each time you redeploy for development purposes. However, the application security deployment properties are the same for Integrated WebLogic Server and the standalone WebLogic Server.

9.3.4.1 Applications That Will Run Using Oracle Single Sign-On (SSO)

Before you can deploy and run the web application with Oracle ADF Security enabled on Oracle WebLogic Server, the administrator of the target server must configure the domain-level `jps-config.xml` file for the Oracle Access Manager (OAM) security provider. To assist with this configuration task, an Oracle WebLogic Scripting Tool (WLST) script has been provided with the JDeveloper install. For details about running this configuration script (with command `addOAMSSOProvider(loginuri, logouturi, autologinuri)`), see the procedure for configuring Oracle WebLogic Server for a web application using ADF Security, OAM SSO, and OPSS SSO in the *Oracle Fusion Middleware Security Guide*.

Running the configuration script ensures that the ADF Security framework defers to the OAM service provider to clear the `ObSSOCookie` token. OAM uses this token to save the identity of authenticated users and, unless it is cleared during logout, the user will be unable to log out.

After the system administrator runs the script on the target server, the domain `jps-config.xml` file will contain the following security provider definition that is specific for ADF Security:

```

<propertySet name="props.auth.uri">
    <property name="login.url.FORM" value="/${app.context}/adfAuthentication"/>
    <property name="logout.url" value=""/>
</propertySet>

```

Additionally, the authentication type required by SSO is `CLIENT-CERT`. The `web.xml` authentication configuration for the deployed application must specify the `<auth-method>` element as one of the following `CLIENT-CERT` types.

WebLogic supports two types of authentication methods:

- For FORM-type authentication method, specify the elements like this:

```

<login-config>
    <auth-method>CLIENT-CERT,FORM</auth-method>
    <realm-name>myrealm</realm-name>
    <form-login-config>
        <form-login-page>/login.html</form-login-page>
        <form-error-page>/error.html</form-error-page>
    </form-login-config>
</login-config>

```

- For BASIC-type authentication method, specify the elements like this:

```
<login-config>
  <auth-method>CLIENT-CERT, BASIC</auth-method>
  <realm-name>myrealm</realm-name>
</login-config>
```

WebSphere supports a single authentication method. Specify the elements like this:

```
<login-config>
  <auth-method>CLIENT-CERT</auth-method>
  <realm-name>myrealm</realm-name>
  <form-login-config>
    <form-login-page>/login.html</form-login-page>
    <form-error-page>/error.html</form-error-page>
  </form-login-config>
</login-config>
```

You can configure the `web.xml` file either before or after deploying the web application. For further details about setting up the authentication method for Single Sign-On, see the *Oracle Fusion Middleware Security Guide*.

9.3.4.2 Configuring Security for WebLogic Server

In a development environment, JDeveloper will automatically migrate application-level credentials, identities, and policies to the remote WebLogic Server instance only if the server is set up to be in development mode. Integrated WebLogic Server is set up in development mode by default. You can set up a remote WebLogic Server to be in development mode during Oracle WebLogic Server domain creation using the Oracle Fusion Middleware Configuration Wizard. For more information about configuring Oracle WebLogic Server domains, see *Oracle Fusion Middleware Creating Domains Using the Configuration Wizard*.

JDeveloper will not migrate application-level security credentials to WebLogic Server setup in production mode. Typically, in a production environment, administrators will use Enterprise Manager or WLST scripts to deploy an application, including its security requirements.

When you deploy an application to WebLogic Server, credentials (in the `cwallet.sso` and `jazn-data.xml` files) will either overwrite or merge with the WebLogic Server domain-level credential store, depending on whether a property in `weblogic-application.xml` is set to `OVERWRITE` or `MERGE`. In production-mode WebLogic Server, to avoid security risks, only `MERGE` is allowed. For development-mode WebLogic Server, you can set to `OVERWRITE` to test user names and passwords. You can also set the property by running `setDomainEnv.cmd` or `setDomainEnv.sh` with the following option added to the command (usually located in `ORACLE_HOME/user_projects/domains/MyDomain/bin`).

For `setDomainEnv.cmd`:

```
set EXTRA_JAVA_PROPERTIES=-Djps.app.credential.override.allowed=true
  %EXTRA_JAVA_PROPERTIES%
```

For `setDomainEnv.sh`:

```
EXTRA_JAVA_PROPERTIES="-Djps.app.credential.override.allowed=true
  ${EXTRA_JAVA_PROPERTIES}"
export EXTRA_JAVA_PROPERTIES
```

If the Administration Server is already running, you must restart it for this setting to take effect.

You can check to see whether WebLogic Server is in production mode by using the Oracle WebLogic Server Administration Console or by verifying the following line in the WebLogic Server `config.xml` file:

```
<production-mode-enabled>true</production-mode-enabled>
```

By default, JDeveloper sets the application's credentials, identities, and policies to `OVERWRITE` mode. That is, the **Application Policies**, **Credentials**, and **Users and Groups** options are selected by default in the Application Properties dialog Deployment page. However, an application's credentials will be migrated only if the target WebLogic Server instance is set to development mode with `-Djps.app.credential.override.allowed=true`.

Policy migration only works in development-mode. Identity migration only works when using JDeveloper to directly deploy to WebLogic Server regardless of whether it is in development or production-mode.

When your application is ready for deployment to a production environment, you should remove the identities from the `jazn-data.xml` file or disable the migration of identities by deselecting **Users and Groups** from the Application Properties dialog. Application credentials must be manually migrated outside of JDeveloper.

Note: Before you migrate the `jazn-data.xml` file to a production environment, check that the policy store does not contain duplicate permissions for a grant. If a duplicate permission (one that has the same name and class) appears in the file, the administrator migrating the policy store will receive an error and the migration of the policies will be halted. You should manually edit the `jazn-data.xml` file to remove any duplicate permissions from a grant definition.

For more information about migrating application credentials and other `jazn-data` user credentials, see the *Oracle Containers for J2EE Security Guide*.

9.3.4.2.1 Applications with JDBC Data Source for WebLogic

If your application uses application-level JDBC data sources with password indirection for database connections, you may need to create credential maps in WebLogic Server to enable the database connection. For more information, see Section 9.3.7, "What You May Need to Know About JDBC Data Source for Oracle WebLogic Server."

9.3.4.3 Configuring Security for WebSphere Server

Applications with credentials and policies in the `jazn-data.xml` and `cwallet.sso` files can be migrated to WebSphere. You will need to perform additional tasks in WebSphere. Be aware that the `opss-application.xml` file is not included in the application EAR file if it is intended for WebSphere.

Note: Before you migrate the `jazn-data.xml` file to a production environment, check that the policy store does not contain duplicate permissions for a grant. If a duplicate permission (one that has the same name and class) appears in the file, the administrator migrating the policy store will receive an error and the migration of the policies will be halted. You should manually edit the `jazn-data.xml` file to remove any duplicate permissions from a grant definition.

For more information about setting up WebSphere to accept credentials and policies, see the *Oracle Fusion Middleware Third-Party Application Server Guide*.

9.3.4.3.1 Applications with JDBC Data Source for WebSphere

If your application uses application-level JDBC data sources with password indirection for database connections, you will need to create a JDBC data source in WebSphere. For more information, see the IBM WebSphere documentation.

9.3.4.3.2 Editing the web.xml File to Protect the Application Root for WebSphere

When you enable ADF Security for your web application, the `web.xml` file includes the Java EE security constraint `allPages` to protect the Java EE application root. By default, to support deploying to Oracle WebLogic Server, JDeveloper specifies the URL pattern for the security constraint as `/` (backslash). If you intend to deploy the application to IBM WebSphere, the correct URL pattern is `/*` (backslash-asterisk). Before you deploy the application to WebSphere, manually edit the `web.xml` file for your application to change the `allPages` security constraint as follows:

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>allPages</web-resource-name>
    <url-pattern>/*</url-pattern>
  </web-resource-collection>
  . . .
</security-constraint>
```

9.3.5 How to Replicate Memory Scopes in a Clustered Environment

If you are deploying an application that is intended to run in a clustered environment, you need to ensure that all managed beans with a lifespan longer than one request are serializable, and that the ADF framework is aware of changes to managed beans stored in ADF scopes (view scope and page flow scope).

For more information, see the "How to Set Managed Bean Memory Scopes in a Server-Cluster Environment" section of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

9.3.6 How to Enable the Application for ADF MBeans

An ADF application uses many XML files for setting configuration information. Some of these configuration files have ADF MBean counterparts that are deployed with the application. After the application has been deployed, you can change configuration properties by accessing the ADF MBeans using the Enterprise Manager Fusion Middleware Control MBean browser.

To enable ADF MBeans, you must register them in the `web.xml` file. Example 9-4 shows a `web.xml` file with listener entries for connections and configuration.

Example 9-4 Enabling ADF MBeans in the web.xml File

```
<listener>
  <listener-class>
    oracle.adf.mbean.share.connection.ADFConnectionLifecycleCallBack
  </listener-class>
</listener>
<listener>
  <listener-class>
    oracle.adf.mbean.share.config.ADFConfigLifecycleCallBack</listener-class>
</listener>
```

Additionally, the `ADFConnection` and `ADFConfig` MBeans require the application to be configured with an MDS repository. MDS configuration entries in the `adf-config.xml` file for a file-based MDS are shown in Example 9–5. For more information about configuring MDS, see the *Oracle Application Server Administrator's Guide*.

Example 9–5 MDS Configuration Entries in the `adf-config.xml` File

```
<adf-mds-config xmlns="http://xmlns.oracle.com/adf/mds/config">
  <mds-config xmlns="http://xmlns.oracle.com/mds/config" version="11.1.1.000">
    <persistence-config>
      <metadata-store-usages>
        <metadata-store-usage
          default-cust-store="true" deploy-target="true" id="myFileStore">
          <metadata-store name="myReposName"
            class-name=
              "oracle.mds.persistence.stores.file.FileMetadataStore">
            <property name="metadata-path" value="C:\Temp\"/>
            <property name="partition-name" value="myAppName"/>
          </metadata-store>
        </metadata-store-usage>
      </metadata-store-usages>
    </persistence-config>
  </mds-config>
</adf-mds-config>
```

In a production environment, an MDS repository that uses a database is required. You can use JDeveloper, Enterprise Manager Fusion Middleware Control, or WLST commands to switch from a file-based repository to a database MDS repository.

Additionally, if several applications are sharing the same MDS configuration, you can ensure that each application has distinct customization layers by defining a `adf:adf-properties-child` property in the `adf-config.xml` file. JDeveloper automatically generates this entry when creating applications. If your `adf-config.xml` file does not have this entry, add it to the file with code similar to that of Example 9–6.

Example 9–6 Adding MDS Partition Code to the `adf-config.xml` File

```
<adf:adf-properties-child xmlns="http://xmlns.oracle.com/adf/config/properties">
  <adf-property name="adfAppUID" value="Application3-4434"/>
  <adf-property name="partition_customizations_by_application_id"
    value="true"/>
</adf:adf-properties-child>
```

The `value` attribute is either generated by JDeveloper or you can set it to any unique identifier within the server farm where the application is deployed. This value can be set to the `value` attribute of the `adfAppUID` property.

When `adf-property name` is set to `adfAppUID`, then the corresponding `value` property should be set to the name of the application. By default, JDeveloper generates the `value` property using the application's package name. If the package name is not specified, JDeveloper generates the `value` property by using the workspace name and a four-digit random number.

For more information about configuring Oracle ADF applications using ADF MBeans, see the *Oracle Fusion Middleware Administrator's Guide for Oracle Application Development Framework*.

9.3.7 What You May Need to Know About JDBC Data Source for Oracle WebLogic Server

An Oracle ADF Java EE application can use a JDBC data source to connect to the database. You use the Oracle WebLogic Server Administration Console to configure a JDBC data source. A JDBC data source has three types: global, application level, and application level with password indirection. You generally set up a global JDBC data source in WebLogic Server. Any application that requires access to that database can use that JDBC data source. An application can also include application-level JDBC data sources. When the application is packaged for deployment, if the **Auto Generate and Synchronize weblogic-jdbc.xml Descriptor During Deployment** option is selected, JDeveloper creates a `connection_name-jdbc.xml` file for each connection that was defined. Each connection's information is written to the corresponding `connection_name-jdbc.xml` file (entries are also changed in `weblogic-application.xml` and `web.xml`). When the application is deployed to WebLogic Server, the server looks for application-level data source information before it looks for the global data source.

If the application is deployed with password indirection set to `true`, WebLogic Server will look for the `connection_name-jdbc.xml` file for user name information and it will then attempt to locate application-level credential maps for these user names to obtain the password. If you are using JDeveloper to directly deploy the application to WebLogic Server, JDeveloper automatically creates the credential map and populates the map to the server using an MBean call.

However, if you are deploying to an EAR file, JDeveloper will not be able to make the MBean call to WebLogic Server. You must set up the credential maps using the Oracle WebLogic Administration Console. Even if you have a global JDBC data source set up, if you do not also have credential mapping set up, WebLogic Server will not be able to map the credentials with passwords and the connection will fail. For more information about JDBC data sources, password indirection, and how to set up application credential mappings, see "JDBC Data Sources" in the "Deploying Applications" section of the JDeveloper online help.

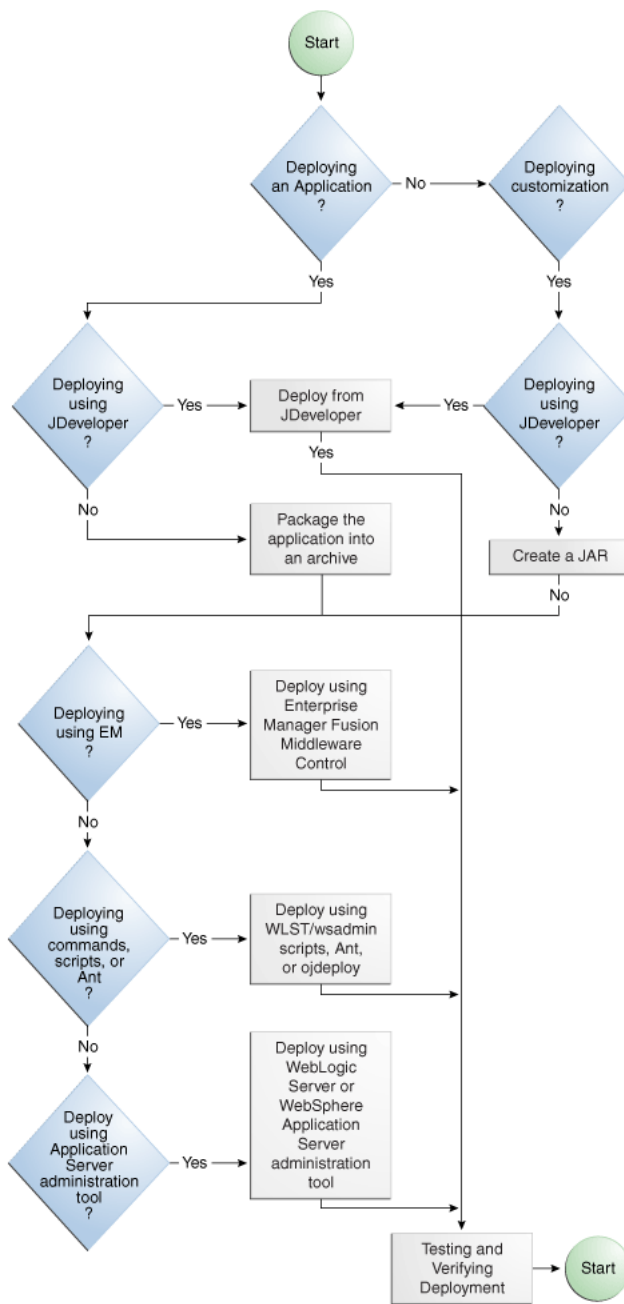
For more information, see the "Preparing the Standalone Application Server for Deployment" section of the *Oracle Fusion Middleware Administrator's Guide for Oracle Application Development Framework*.

9.4 Deploying the Application

You can use JDeveloper to deploy applications directly to the standalone application server or create an archive file and use other tools to deploy to the application server.

Figure 9–5 show the process flow for deploying an application and also for deploying customizations to the target standalone application server.

Figure 9–5 Application Deployment Flow Diagram



Note: Before you begin to deploy applications that use Oracle ADF to the standalone application server, you need to prepare the application server environment by performing tasks such as installing the ADF runtime and creating and extending domains or cells. For more information, see the "Preparing the Standalone Application Server for Deployment" section of the *Oracle Fusion Middleware Administrator's Guide for Oracle Application Development Framework*.

Table 9–1 describes some common deployment techniques that you can use during the application development and deployment cycle. The deployment techniques are listed

in order from deploying on development environments to deploying on production environments. It is likely that in the production environment, the system administrators deploy applications by using Enterprise Manager or the WLST scripting tool.

Table 9–1 Deployment Techniques for Development or Production Environments

Deployment Technique	Environment	When to Use
Run directly from JDeveloper	Test or Development	When you are developing your application. You want deployment to be quick because you will be repeating the editing and deploying process many times. JDeveloper contains Integrated WebLogic Server, on which you can run and test your application.
Use JDeveloper to directly deploy to the target application server	Test or Development	When you are ready to deploy and test your application on an application server in a test environment. On the test server, you can test features (such as LDAP and Oracle Single Sign-On) that are not available on the development server. You can also use the test environment to develop your deployment scripts, for example, using Ant.
Use JDeveloper to deploy to an EAR file, then use the target application server's tools for deployment	Test or Development	When you are ready to deploy and test your application on an application server in a test environment. As an alternative to deploying directly from JDeveloper, you can deploy to an EAR file and then use other tools to deploy to the application server. On the test server, you can test features (such as LDAP and Oracle Single Sign-On) that are not available on the development server. You can also use the test environment to develop your deployment scripts, for example, using Ant.
Use Enterprise Manager or WLST scripts to deploy applications	Production	When your application is in a test and production environment. In production environments, system administrators usually use Enterprise Manager or run WLST scripts to deploy applications.

Any necessary MDS repositories must be registered with the application server. If the MDS repository is a database, the repository maps to a data source with MDS-specific requirements.

If you are deploying the application to Oracle WebLogic Server, make sure to target this data source to the WebLogic Administration Server and to all Managed Servers to which you are deploying the application. For more information about registering MDS, see the *Oracle Application Server Administrator's Guide*.

If you are using the application server's administrative consoles or scripts to deploy an application packaged as an EAR file that requires MDS repository configuration in `adf-config.xml`, you must run the `getMDSArchiveConfig` command to configure MDS before deploying the EAR file. MDS configuration is required if the EAR file contains a MAR file or if the application is enabled for DT@RT (Design Time At Run Time).

For more information about WLST commands, see the *Oracle Fusion Middleware WebLogic Scripting Tool Command Reference*. For more information about `wsadmin` commands, see the *Oracle Fusion Middleware Third-Party Application Server Guide* and the *Oracle Fusion Middleware Configuration Guide for WebSphere*.

If you plan to configure ADF connection information or `adf-config.xml` using ADF MBeans after the application has been deployed, make sure that the application is configured with MDS and that you have the MBean listeners enabled in the `web.xml` file. For more information, see Section 9.3.6, "How to Enable the Application for ADF MBeans."

9.4.1 How to Deploy to WebLogic Server from JDeveloper

Before you begin:

Create an application-level deployment profile that deploys to an EAR file.

Note: When you are deploying to Oracle WebLogic Server from JDeveloper, ensure that the HTTP Tunneling property is enabled in the Oracle WebLogic Server Administration Console. This property is located under **Servers > ServerName > Protocols**. *ServerName* refers to the name of Oracle WebLogic Server.

Note: JDeveloper does not support deploying applications to individual Managed Servers that are members of a cluster. You may be able to target one or more Managed Servers within a cluster using the Oracle WebLogic Server Administration Console or other Oracle WebLogic tools; however, the cluster can be negatively affected. For more information about deploying to Oracle WebLogic Server clusters, see the *Oracle Application Server Administrator's Guide*.

To deploy to the target application server from JDeveloper:

1. In the Application Navigator, right-click the application and choose **Deploy > deployment profile**.
2. In the Deploy wizard Deployment Action page, select **Deploy to Application Server** and click **Next**.
3. In the Select Server page, select the application server connection.
4. If you are deploying to a WebLogic Server instance, the WebLogic Options page appears. Select a deploy option and click **Next**.

Note: If you are deploying an ADF application, do not use the **Deploy to all instances in the domain** option.

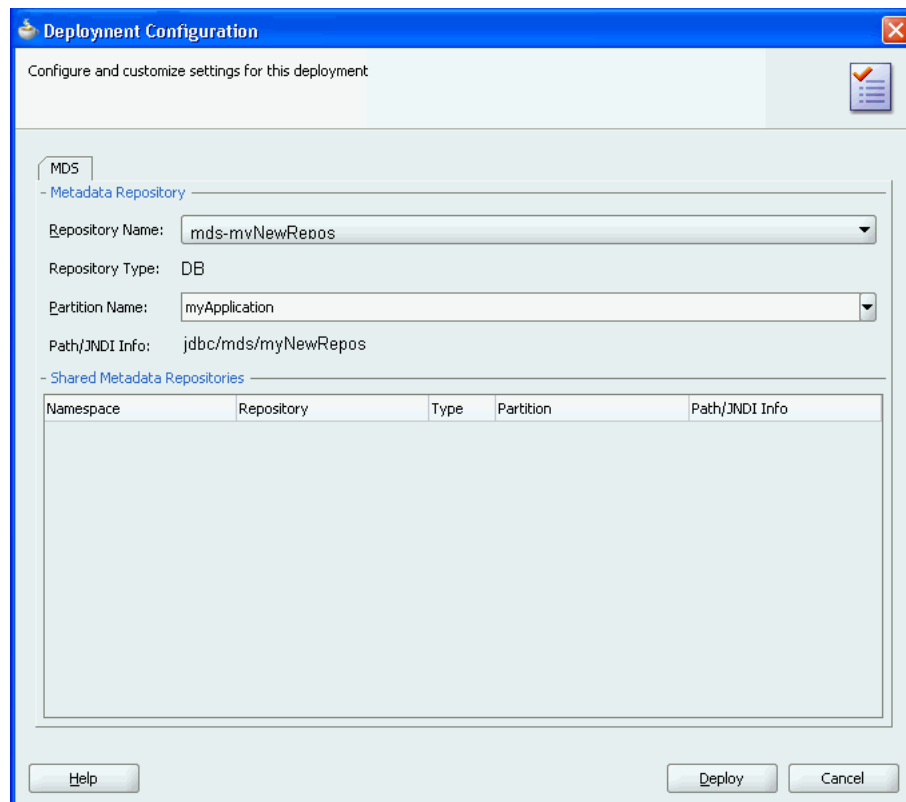
5. Click **Finish**.

During deployment, you can see the process steps displayed in the deployment Log window. You can inspect the contents of the modules (archives or exploded EAR) being created by clicking on the links that are provided in the log window. The archive or exploded EAR file will open in the appropriate editor or directory window for inspection.

If the `adf-config.xml` file in the EAR file requires MDS repository configuration, the Deployment Configuration dialog appears for you to choose the target metadata repository or shared metadata repositories, as shown in Figure 9-6. The **Repository Name** dropdown list allows you to choose a target

metadata repository from a list of metadata repositories registered with the Administration Server. The **Partition Name** dropdown list allows you to choose the metadata repository partition to which the application's metadata will be imported during deployment. You can use WLST/wsadmin scripts, Oracle WebLogic Server Administration Tool, or WebSphere Administrative Tool, respectively, to configure and register MDS. For more information about managing the MDS repository, see the *Oracle Application Server Administrator's Guide*.

Figure 9–6 MDS Configuration and Customization for Deployment



Note: If you are deploying a Java EE application, click the application menu next to the Java EE application in the Application Navigator.

For more information on creating application server connections, see Section 9.3.1, "How to Create a Connection to the Target Application Server."

Tip: You may get an exception in JDeveloper when trying to deploy large EAR files. The workaround is to deploy the application using the Oracle WebLogic Server Administration Console.

9.4.2 How to Create an EAR File for Deployment

You can also use the deployment profile to create an archive file (EAR file). You can then deploy the archive file using Enterprise Manager, WLST scripts, or Oracle WebLogic Server Administration Console.

Although an Oracle ADF Java EE application is encapsulated in an EAR file (which usually includes WAR, MAR, and JAR components), it may have parts that are not deployed with the EAR.

To create an EAR archive file:

- In the Application Navigator, right-click the application containing the deployment profile, and choose **Deploy > deployment profile > to EAR file**.

If an EAR file is deployed at the application level, and it has dependencies on a JAR file in the data model project and dependencies on a WAR file in the view-controller project, then the files will be located in the following directories by default:

- `ApplicationDirectory/deploy/EARdeploymentprofile.EAR`
- `ApplicationDirectory/ModelProject/deploy/JARdeploymentprofile.JAR`
- `ApplicationDirectory/ViewControllerProject/deploy/WARdeploymentprofile.WAR`

Tip: Choose **View > Log** and select the Deployment tab to see messages generated during the creation of the archive file.

9.4.3 What You May Need to Know About ADF Libraries

An ADF Library is a JAR file that contains JAR services registered for ADF components such as ADF task flows, pages, or application modules. If you want the ADF components in a project to be reusable, you create an ADF Library deployment profile for the project and then create an ADF Library JAR based on that profile.

An application or project can consume the ADF Library JAR when you add it using the Resource Palette or manually by adding it to the library classpath. When the ADF Library JAR is added to a project, it will be included in the project's WAR file if the **Deployed by Default** option is selected.

For more information, see the "Reusing Application Components" section of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

9.4.4 How to Deploy New Customizations Applied to ADF Library

If you have created new customizations for an ADF Library, you can use the MAR profile to deploy these customizations to any deployed application that consumes that ADF Library. For instance, `applicationA`, which consumes `ADFLibraryB`, was deployed to a remote application server. Later on, when new customizations are added to `ADFLibraryB`, you only need to deploy the updated customizations into `applicationA`. You do not need to repackage and redeploy the whole application, nor do you need to manually patch the MDS repository.

Note: This procedure is for applying ADF Library customization changes to an application that has already been deployed to a remote application server. It is not for the initial packaging of customizations into a MAR that will eventually be a part of an EAR. For information about the initial packaging of the customization using a MAR, see Section 9.3.2.2, "Creating a MAR Deployment Profile."

To deploy ADF Library customizations, create a new MAR profile that includes only the customizations to be deployed and then use JDeveloper to:

- Deploy the customizations directly into the MDS repository in the remote application server.
- Deploy the customizations to a JAR. And then import the JAR into the MDS repository using tools such as the Fusion Middleware Control.

9.4.4.1 Exporting Customization to a Deployed Application

You can export the customizations directly from JDeveloper into the MDS repository for the deployed application on the remote application server.

Before you begin:

Create new customizations to the ADF Library using JDeveloper.

To export the customizations directly into the application server:

1. In the Application Navigator, right-click the application and choose **Deploy > metadata**.
2. In the Deploy Metadata dialog Deployment Action page, select **Export to a Deployed Application** and click **Next**.

If the MAR profile is included in any of the application's EAR profiles, **Export to a Deployed Application** will be dimmed and disabled.

3. In the Application Server page, select the application server connection and click **Next**.
4. For WebLogic Server, the Server Instance page appears. In this page, select the server instance where the deployed application is located and click **Next**.
5. In the Deployed Application page, select the application you want to apply the customizations to and click **Next**.
6. In the Sandbox Instance page, if you want to deploy to a sandbox, select **Deploy to an associated sandbox**, choose the sandbox instance and click **Next**.
7. In the Summary page, verify the information and click **Finish**.

9.4.4.2 Deploying Customizations to a JAR

When you deploy the ADF Library customizations to a JAR, you are packaging the contents as defined by the MAR profile.

Before you begin:

Create new customizations to the ADF Library using JDeveloper.

To deploy the customizations as a JAR

1. In the Application Navigator, right-click the application and choose **Deploy > metadata**.
2. In the Deploy Metadata dialog Deployment Action page, select **Deploy to MAR**.
3. In the Summary page, click **Finish**.
4. Use Enterprise Manager Fusion Middleware Control or the application server's administration tool to import the JAR into the MDS repository.

9.4.5 What You May Need to Know About ADF Libraries

An ADF Library is a JAR file that contains JAR services registered for ADF components such as ADF task flows or pages. If you want the ADF components in a project to be reusable, you create an ADF Library deployment profile for the project and then create an ADF Library JAR based on that profile.

An application or project can consume the ADF Library JAR when you add it using the Resource Palette or manually by adding it to the library classpath. When the ADF Library JAR is added to a project, it will be included in the project's WAR file if the **Deployed by Default** option is selected.

For more information, see the "Reusing Application Components" section of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

9.4.6 What You May Need to Know About EAR Files and Packaging

When you package an Oracle ADF application into an EAR file, it can contain the following:

- WAR files: Each web-based view-controller project should be packaged into a WAR file.
- MAR file: If the application has customizations that are deployed with the application, it should be packaged into a MAR.
- ADF Library JAR files: If the application consumes ADF Library JARs, these JAR files may be packaged within the EAR.
- Other JAR files: The application may have other dependent JAR files that are required. They can be packaged within the EAR.

9.4.7 How to Deploy the Application Using Scripts and Ant

You can deploy the application using commands and automate the process by putting those commands in scripts. The `ojdeploy` command can be used to deploy an application without JDeveloper. You can also use Ant scripts to deploy the application. JDeveloper has a feature to help you build Ant scripts. Depending on your requirements, you may be able to integrate regular scripts with Ant scripts.

For more information about commands, scripts, and Ant, see the *Oracle Fusion Middleware Administrator's Guide for Oracle Application Development Framework*.

9.4.8 What You May Need to Know About JDeveloper Runtime Libraries

When an application is deployed, it includes some of its required libraries with the application. The application may also require shared libraries that have already been loaded to WebLogic Server as JDeveloper runtime libraries. It may be useful to know which JDeveloper libraries are packaged within which WebLogic Server shared library. For a listing of the contents of the JDeveloper runtime libraries, see the *Oracle Fusion Middleware Administrator's Guide for Oracle Application Development Framework*.

9.5 Postdeployment Configuration

After you have deployed your application to WebLogic Server, you can perform configuration tasks.

9.5.1 How to Migrate an Application

If you want to migrate an Oracle ADF Java EE application from one WebLogic Server to another WebLogic Server, you may need to perform some of the same steps you did for a first time deployment.

In general, to migrate an application, you would:

- Load the ADF runtime (if it is not already installed) to the target application server. For more information, see the "Preparing the Standalone Application Server for Deployment" section of the *Oracle Fusion Middleware Administrator's Guide for Oracle Application Development Framework*.
- Configure the target WebLogic Server instance with the correct database or URL connection information.
- Migrate security information from the source to the target. For instructions, see Section 9.3.4, "How to Deploy Applications with ADF Security Enabled."
- Deploy the application using Enterprise Manager, Oracle WebLogic Administration Console, or scripts. For more information, see the *Oracle Fusion Middleware Administrator's Guide for Oracle Application Development Framework*.

9.5.2 How to Configure the Application Using ADF MBeans

If ADF MBeans were enabled and packaged with the deployed application, you can configure ADF properties using the Enterprise Manager Fusion Middleware Control MBean Browser. For instructions to enable an application for MBeans, see Section 9.3.6, "How to Enable the Application for ADF MBeans."

For information on how to configure ADF applications using ADF MBeans, see the *Oracle Fusion Middleware Administrator's Guide for Oracle Application Development Framework*.

9.5.3 How to Configure WebSphere for Result Sets Reuse

ADF applications deployed to WebSphere Application Server use shared database connections. You need to set the `non-transactional_datasource` and `DisableMultiThreadedServletConnectionMgmt` properties in WebSphere to allow the application to reuse result sets across requests.

If you do not set these properties, WebSphere will close the result sets between requests.

For more information, see the "Configuring WebSphere Application Server" section in the *Oracle Fusion Middleware Administrator's Guide for Oracle Application Development Framework*.

9.6 Testing the Application and Verifying Deployment

After you deploy the application, you can test it from Oracle WebLogic Server. To test-run your ADF application, open a browser window and enter a URL:

- For non-Faces pages: `http://<host>:port/<context root>/<page>`
- For Faces pages: `http://<host>:port/<context root>/faces/<view_id>`

where `<view_id>` is the view ID of the ADF task flow view activity.

Tip: The context root for an application is specified in the view-controller project settings by default as *ApplicationName/ProjectName/context-root*. You can shorten this name by specifying a name that is unique across the target application server. Right-click the view-controller project, and choose **Project Properties**. In the Project Properties dialog, select **Java EE Application** and enter a unique name for the context root.

Note: */faces* has to be in the URL for Faces pages. This is because JDeveloper configures your *web.xml* file to use the URL pattern of */faces* in order to be associated with the Faces Servlet. The Faces Servlet does its per-request processing, strips out */faces* part in the URL, then forwards the URL to the JSP. If you do not include the */faces* in the URL, then the Faces Servlet is not engaged (since the URL pattern doesn't match). Your JSP is run without the necessary JSF per-request processing.
