Oracle® Application Development Framework

Developer's Guide

10*g* Release 3 (10.1.3.0)

B28967-02

June 2008

ORACLE®

Oracle Application Development Framework Developer's Guide, 10*g* Release 3 (10.1.3.0)

B28967-02

Primary Author: Ken Chu, Orlando Cordero, Ralph Gordon, Rosslynne Hefferan, Mario Korf, Robin Merrin, Steve Muench, Kathryn Munn, Barbara Ramsey, Jon Russell, Deborah Steiner, Odile Sullivan-Tarazi, Poh Lee Tan, Robin Whitmore, Martin Wykes

Contents

Preface xix

Audience xix
Documentation Accessibility xix
Related Documents xx
Conventions xx

1 Introduction to Oracle ADF Applications

1.1 Overview of Oracle Application Development Framework 1-1
1.1.1 Framework Architecture and Supported Technologies 1-1
1.1.1.1 View Layer Technologies Supported 1-2
1.1.1.2 Controller Layer Technologies Supported 1-2
1.1.1.3 Business Services Technologies Supported by ADF Model 1-2
1.1.1.4 Recommended Technologies for J2EE Enterprise Developers 1-3
1.1.2 Declarative Development with Oracle ADF and JavaServer Faces 1-3
1.1.2.1 Declarative J2EE Technologies You May Have Already Used 1-4
1.1.2.2 JSF Offers Dependency Injection, Page Handling, EL and More 1-4
1.1.2.3 Oracle ADF Further Raises the Level of Declarative Development for JSF 1-6
1.1.3 Key ADF Binding Features for JSF Development 1-7
1.1.3.1 Comprehensive JDeveloper Design-Time Support 1-8
1.1.3.2 More Sophisticated UI Functionality Without Coding 1-8
1.1.3.3 Centralize Common Functionality in Layered Model Metadata 1-9
1.1.3.4 Simplified Control Over Page Lifecycle 1-9
1.2 Development Process with Oracle ADF and JavaServer Faces 1-10
1.2.1 Overview of the Steps for Building an Application 1-10
1.2.1.1 Starting by Creating a New Application 1-11
1.2.1.2 Building the Business Service in the Model Project 1-12
1.2.1.3 Creating a Data Control for Your Service to Enable Data Binding 1-12
1.2.1.4 Dragging and Dropping Data to Create a New JSF Page 1-14
1.2.1.5 Examining the Binding Metadata Files Involved 1-15
1.2.1.6 Understanding How Components Reference Bindings via EL 1-16
1.2.1.7 Configuring Binding Properties If Needed 1-18
1.2.1.8 Understanding How Bindings Are Created at Runtime 1-18

1.2.2 Making the Display More Data-Driven ... 1-19
1.2.2.1 Hiding and Showing Groups of Components Based on Binding Properties ... 1-19
1.2.2.2 Toggling Between Alternative Sets of Components Based on Binding Properties ... 1-20

2 Oracle ADF Service Request Demo Overview

2.1 Introduction to the Oracle ADF Service Request Demo ... 2-1
2.1.1 Requirements for Oracle ADF Service Request Application ... 2-2
2.1.2 Overview of the Schema ... 2-2
2.2 Setting Up the Oracle ADF Service Request Demo ... 2-4
2.2.1 Downloading and Installing the Oracle ADF Service Request Application ... 2-4
2.2.2 Installing the Oracle ADF Service Request Schema ... 2-5
2.2.3 Creating the Oracle JDeveloper Database Connection ... 2-7
2.2.4 Running the Oracle ADF Service Request Demo in JDeveloper ... 2-8
2.2.5 Running the Oracle ADF Service Request Demo Unit Tests in JDeveloper ... 2-9
2.3 Quick Tour of the Oracle ADF Service Request Demo ... 2-11
2.3.1 Customer Logs In and Reviews Existing Service Requests ... 2-11
2.3.2 Customer Creates a Service Request ... 2-13
2.3.3 Manager Logs In and Assigns a Service Request ... 2-16
2.3.4 Manager Views Reports and Updates Technician Skills ... 2-18
2.3.5 Technician Logs In and Updates a Service Request ... 2-21

3 Building and Using Application Services

3.1 Introduction to Business Services ... 3-1
3.2 Implementing Services with EJB Session Beans ... 3-2
3.2.1 How to Create a Session Bean ... 3-2
3.2.1.1 Remote and Local Interfaces ... 3-2
3.2.1.2 Generating Session Facade Methods ... 3-2
3.2.2 What Happens When You Create a Session Bean ... 3-3
3.2.3 What You May Need to Know When Creating a Session Bean ... 3-7
3.2.4 How to Update an Existing Session Bean With New Entities ... 3-8
3.3 Creating Classes to Map to Database Tables ... 3-8
3.3.1 How to Create Classes ... 3-8
3.3.2 What Happens when you Create a Class ... 3-9
3.3.3 What You May Need to Know ... 3-10
3.3.3.1 Associating Descriptors with Different Database Tables ... 3-10
3.3.3.2 Using Amendment Methods ... 3-10
3.3.3.3 Modifying the Generated Code ... 3-11
3.4 Mapping Classes to Tables ... 3-11
3.4.1 Types of Mappings ... 3-12
3.4.2 Direct Mappings ... 3-12
3.4.3 How to Create Direct Mappings ... 3-12
3.4.4 What Happens when you Create a Direct Mapping ... 3-13
3.4.5 What You May Need to Know ... 3-13

3.5 Mapping Related Classes with Relationships ... 3-14
3.5.1 How to Create Relationship Mappings ... 3-15
3.5.2 What Happens when you Create a Relationship ... 3-15
3.5.3 What You May Need to Know ... 3-15
3.6 Finding Objects by Primary Key ... 3-17
3.7 Querying Objects ... 3-17
3.7.1 How to Create a Query ... 3-17
3.7.2 What You May Need to Know ... 3-18
3.7.2.1 Using a Query By Example ... 3-18
3.7.2.2 Sorting Query Results ... 3-18
3.8 Creating and Modifying Objects with a Unit of Work ... 3-18
3.8.1 How to Create a Unit of Work ... 3-19
3.8.1.1 Creating Objects with Unit of Work ... 3-19
3.8.1.2 Typical Unit of Work Usage ... 3-20
3.8.2 What Happens when you Modify a Unit of Work ... 3-21
3.8.2.1 Deleting Objects ... 3-22
3.8.3 What You May Need to Know ... 3-22
3.8.3.1 Unit of Work and Change Policy ... 3-22
3.8.3.2 Nested and Parallel Units of Work ... 3-22
3.9 Interacting with Stored Procedures ... 3-23
3.9.1 Specifying an Input Parameter ... 3-23
3.9.2 Specifying an Output Parameter ... 3-24
3.9.3 Specifying an Input / Output Parameter ... 3-25
3.9.4 Using an Output Parameter Event ... 3-25
3.9.5 Using a StoredFunctionCall ... 3-25
3.9.6 Query Sequencing ... 3-26
3.10 Exposing Services with ADF Data Controls ... 3-26
3.10.1 How to Create ADF Data Controls ... 3-26
3.10.2 Understanding the Data Control Files ... 3-27
3.10.2.1 About the DataControls.dcx File ... 3-27
3.10.2.2 About the Structure Definition Files ... 3-27
3.10.2.3 About the Entity XML Files ... 3-27
3.10.2.4 About the Design-time XML Files ... 3-28
3.10.3 Understanding the Data Control Palette ... 3-28
3.10.3.1 Overview of the Data Control Business Objects ... 3-29
3.10.3.2 Refreshing ADF Data Controls After Modifying Business Services ... 3-30

4 Getting Started with ADF Faces

4.1 Introduction to ADF Faces ... 4-1
4.2 Setting Up a Workspace and Project ... 4-3
4.2.1 What Happens When You Use an Application Template to Create a Workspace ... 4-3
4.2.1.1 Starter web.xml File ... 4-5
4.2.1.2 Starter faces-config.xml File ... 4-6
4.2.2 What You May Need to Know About the ViewController Project ... 4-7
4.2.3 What You May Need to Know About Multiple JSF Configuration Files ... 4-8

4.3 Creating a Web Page 4-9
4.3.1 How to Add a JSF Page 4-9
4.3.2 What Happens When You Create a JSF Page 4-11
4.3.3 What You May Need to Know About Using the JSF Navigation Diagram 4-12
4.3.4 What You May Need to Know About ADF Faces Dependencies and Libraries 4-13
4.4 Laying Out a Web Page 4-13
4.4.1 How to Add UI Components to a JSF Page 4-14
4.4.2 What Happens When You First Insert an ADF Faces Component 4-15
4.4.2.1 More About the web.xml File 4-17
4.4.2.2 More About the faces-config.xml File 4-18
4.4.2.3 Starter adf-faces-config.xml File 4-18
4.4.3 What You May Need to Know About Creating JSF Pages 4-20
4.4.3.1 Editing in the Structure Window 4-21
4.4.3.2 Displaying Errors 4-21
4.4.4 Using the PanelPage Component 4-22
4.4.4.1 PanelPage Facets 4-24
4.4.4.2 Page Body Contents 4-27
4.5 Creating and Using a Backing Bean for a Web Page 4-28
4.5.1 How to Create and Configure a Backing Bean 4-29
4.5.2 What Happens When You Create and Configure a Backing Bean 4-29
4.5.3 How to Use a Backing Bean in a JSF Page 4-30
4.5.4 How to Use the Automatic Component Binding Feature 4-31
4.5.5 What Happens When You Use Automatic Component Binding in JDeveloper 4-32
4.5.6 What You May Need to Know About Backing Beans and Managed Beans 4-33
4.5.7 Using ADF Data Controls and Backing Beans 4-34
4.6 Best Practices for ADF Faces 4-35

5 Displaying Data on a Page

5.1 Introduction to Displaying Data on a Page 5-1
5.2 Using the Data Control Palette 5-2
5.2.1 How to Understand the Items on the Data Control Palette 5-3
5.2.2 How to Use the Data Control Palette 5-5
5.2.3 What Happens When You Use the Data Control Palette 5-7
5.2.4 What Happens at Runtime 5-8
5.3 Working with the DataBindings.cpx File 5-9
5.3.1 How to Create a DataBindings.cpx File 5-9
5.3.2 What Happens When You Create a DataBindings.cpx File 5-10
5.4 Configuring the ADF Binding Filter 5-10
5.4.1 How to Configure the ADF Binding Filter 5-10
5.4.2 What Happens When You Configure an ADF Binding Filter 5-11
5.4.3 What Happens at Runtime 5-12
5.5 Working with Page Definition Files 5-12
5.5.1 How to Create a Page Definition File 5-12
5.5.2 What Happens When You Create a Page Definition File 5-13
5.5.2.1 Binding Objects Defined in the parameters Element 5-14
5.5.2.2 Binding Objects Defined in the executables Element 5-15
5.5.2.3 Binding Objects Defined in the bindings Element 5-17

5.5.3 What Happens at Runtime ... 5-19
5.5.4 What You May Need to Know About Binding Container Scope ... 5-19
5.6 Creating ADF Data Binding EL Expressions ... 5-19
5.6.1 How to Create an ADF Data Binding EL Expression ... 5-19
5.6.2 How to Use the Expression Builder ... 5-21
5.6.3 What Happens When You Create ADF Data Binding Expressions ... 5-23
5.6.3.1 EL Expressions That Reference Attribute Binding Objects ... 5-23
5.6.3.2 EL Expressions That Reference Table Binding Objects ... 5-24
5.6.3.3 EL Expressions That Reference Action Binding Objects ... 5-25
5.6.4 What You May Need to Know About ADF Binding Properties ... 5-27
5.6.5 What You May Need to Know About Binding to Values in Other Pages ... 5-27

6 Creating a Basic Page

6.1 Introduction to Creating a Basic Page ... 6-1
6.2 Using Attributes to Create Text Fields ... 6-2
6.2.1 How to Use the Data Control Palette to Create a Text Field ... 6-2
6.2.2 What Happens When You Use the Data Control Palette to Create a Text Field ... 6-3
6.2.2.1 Creating and Using Iterator Bindings ... 6-3
6.2.2.2 Creating and Using Value Bindings ... 6-4
6.2.2.3 Using EL Expressions to Bind UI Components ... 6-5
6.2.3 What Happens at Runtime: The JSF and ADF Lifecycles ... 6-6
6.3 Creating a Basic Form ... 6-9
6.3.1 How to Use the Data Control Palette to Create a Form ... 6-9
6.3.2 What Happens When You Use the Data Control Palette to Create a Form ... 6-11
6.3.2.1 Using Facets ... 6-12
6.4 Incorporating Range Navigation into Forms ... 6-13
6.4.1 How to Insert Navigation Controls into a Form ... 6-13
6.4.2 What Happens When Command Buttons Are Created Using the Data Control Palette ... 6-14
6.4.2.1 Using Action Bindings for Built-in Navigation Operations ... 6-14
6.4.2.2 Iterator RangeSize Attribute ... 6-15
6.4.2.3 Using EL Expressions to Bind to Navigation Operations ... 6-16
6.4.3 What Happens at Runtime: About Action Events and Action Listeners ... 6-17
6.4.4 What You May Need to Know About the Browser Back Button ... 6-17
6.5 Modifying the UI Components and Bindings on a Form ... 6-18
6.5.1 How to Modify the UI Components and Bindings ... 6-18
6.5.1.1 Changing the Value Binding for a UI Component ... 6-20
6.5.1.2 Changing the Action Binding for a UI Component ... 6-20
6.5.2 What Happens When You Modify Attributes and Bindings ... 6-20

7 Adding Tables

7.1 Introduction to Adding Tables 7-1
7.2 Creating a Basic Table 7-2
7.2.1 How to Create a Basic Table 7-2
7.2.2 What Happens When You Use the Data Control Palette to Create a Table 7-4
7.2.2.1 Iterator and Value Bindings for Tables 7-4
7.2.2.2 Code on the JSF Page for an ADF Faces Table 7-5
7.3 Incorporating Range Navigation into Tables 7-6
7.3.1 How to Use Navigation Controls in a Table 7-7
7.3.2 What Happens When You Use Navigation Controls in a Table 7-7
7.3.3 What Happens at Runtime 7-8
7.3.4 What You May Need to Know About the Browser Back Button 7-8
7.4 Modifying the Attributes Displayed in the Table 7-9
7.4.1 How to Modify the Displayed Attributes 7-9
7.4.2 How to Change the Binding for a Table 7-11
7.4.3 What Happens When You Modify Bindings or Displayed Attributes 7-11
7.5 Adding Hidden Capabilities to a Table 7-11
7.5.1 How to Use the DetailStamp Facet 7-12
7.5.2 What Happens When You Use the DetailStamp Facet 7-13
7.5.3 What Happens at Runtime 7-13
7.6 Enabling Row Selection in a Table 7-14
7.6.1 How to Use the TableSelectOne Component in the Selection Facet 7-16
7.6.2 What Happens When You Use the TableSelectOne Component 7-17
7.6.3 What Happens at Runtime 7-17
7.6.4 How to Use the TableSelectMany Component in the Selection Facet 7-18
7.6.5 What Happens When You Use the TableSelectMany Component 7-19
7.6.6 What Happens at Runtime 7-21
7.7 Setting the Current Object Using a Command Component 7-22
7.7.1 How to Manually Set the Current Row 7-22
7.7.2 What Happens When You Set the Current Row 7-23
7.7.3 What Happens At Runtime 7-23

8 Displaying Master-Detail Data

8.1 Introduction to Displaying Master-Detail Data 8-1
8.2 Identifying Master-Detail Objects on the Data Control Palette 8-2
8.3 Using Tables and Forms to Display Master-Detail Objects 8-4
8.3.1 How to Display Master-Detail Objects in Tables and Forms 8-5
8.3.2 What Happens When You Create Master-Detail Tables and Forms 8-6
8.3.2.1 Code Generated in the JSF Page 8-6
8.3.2.2 Binding Objects Defined in the Page Definition File 8-7
8.3.3 What Happens at Runtime 8-8
8.3.4 What You May Need to Know About Master-Detail on Separate Pages 8-8

8.4 Using Trees to Display Master-Detail Objects ... 8-9
8.4.1 How to Display Master-Detail Objects in Trees ... 8-10
8.4.2 What Happens When You Create ADF Databound Trees ... 8-13
8.4.2.1 Code Generated in the JSF Page ... 8-13
8.4.2.2 Binding Objects Defined in the Page Definition File ... 8-13
8.4.3 What Happens at Runtime ... 8-15
8.5 Using Tree Tables to Display Master-Detail Objects ... 8-15
8.5.1 How to Display Master-Detail Objects in Tree Tables ... 8-16
8.5.2 What Happens When You Create a Databound Tree Table ... 8-16
8.5.2.1 Code Generated in the JSF Page ... 8-17
8.5.2.2 Binding Objects Defined in the Page Definition File ... 8-17
8.5.3 What Happens at Runtime ... 8-17
8.6 Using an Inline Table to Display Detail Data in a Master Table ... 8-18
8.6.1 How to Display Detail Data Using an Inline Table ... 8-19
8.6.2 What Happens When You Create an Inline Detail Table ... 8-20
8.6.2.1 Code Generated in the JSF Page ... 8-20
8.6.2.2 Binding Objects Defined in the Page Definition File ... 8-21
8.6.3 What Happens at Runtime ... 8-22

9 Adding Page Navigation

9.1 Introduction to Page Navigation ... 9-1
9.2 Creating Navigation Rules ... 9-2
9.2.1 How to Create Page Navigation Rules ... 9-2
9.2.1.1 About Navigation Rule Elements ... 9-2
9.2.1.2 Using the Navigation Modeler to Define Navigation Rules ... 9-3
9.2.1.3 Using the JSF Configuration Editor ... 9-5
9.2.2 What Happens When You Create a Navigation Rule ... 9-8
9.2.3 What Happens at Runtime ... 9-10
9.2.4 What You May Need to Know About Navigation Rules and Cases ... 9-11
9.2.4.1 Defining Rules in Multiple Configuration Files ... 9-11
9.2.4.2 Overlapping Rules ... 9-11
9.2.4.3 Conflicting Navigation Rules ... 9-12
9.2.4.4 Splitting Navigation Cases Over Multiple Rules ... 9-12
9.2.5 What You May Need to Know About the Navigation Modeler ... 9-13
9.3 Using Static Navigation ... 9-14
9.3.1 How to Create Static Navigation ... 9-14
9.3.2 What Happens When You Create Static Navigation ... 9-15
9.4 Using Dynamic Navigation ... 9-16
9.4.1 How to Create Dynamic Navigation ... 9-17
9.4.2 What Happens When You Create Dynamic Navigation ... 9-18
9.4.3 What Happens at Runtime ... 9-20
9.4.4 What You May Need to Know About Using Default Cases ... 9-20
9.4.5 What You May Need to Know About Action Listener Methods ... 9-21
9.4.6 What You May Need to Know About Data Control Method Outcome Returns ... 9-21

10 Creating More Complex Pages

10.1 Introduction to More Complex Pages ... 10-1
10.2 Using a Managed Bean to Store Information ... 10-2
10.2.1 How to Use a Managed Bean to Store Information ... 10-2
10.2.2 What Happens When You Create a Managed Bean ... 10-3
10.3 Creating Command Components to Execute Methods ... 10-4
10.3.1 How to Create a Command Component Bound to a Service Method ... 10-5
10.3.2 What Happens When You Create Command Components Using a Method ... 10-6
10.3.2.1 Using Parameters in a Method ... 10-6
10.3.2.2 Using EL Expressions to Bind to Methods ... 10-7
10.3.3 What Happens at Runtime ... 10-8
10.4 Setting Parameter Values Using a Command Component ... 10-8
10.4.1 How to Set Parameters Using Command Components ... 10-8
10.4.2 What Happens When You Set Parameters ... 10-9
10.4.3 What Happens at Runtime ... 10-9
10.5 Overriding Declarative Methods ... 10-10
10.5.1 How to Override a Declarative Method ... 10-10
10.5.2 What Happens When You Override a Declarative Method ... 10-13
10.6 Creating a Form or Table Using a Method that Takes Parameters ... 10-14
10.6.1 How to Create a Form or Table Using a Method That Takes Parameters ... 10-15
10.6.2 What Happens When You Create a Form Using a Method that Takes Parameters 10-15
10.6.3 What Happens at Runtime ... 10-16
10.7 Creating an Input Form for a New Record ... 10-16
10.7.1 How to Use Constructors to Create an Input Form ... 10-17
10.7.2 What Happens When You Use a Constructor ... 10-18
10.7.3 How to Use a Custom Method to Create an Input Form ... 10-20
10.7.4 What Happens When You Use Methods to Create a Parameter Form ... 10-20
10.7.4.1 Using Variables and Parameters ... 10-20
10.7.5 What Happens at Runtime ... 10-22
10.8 Creating Search Pages ... 10-23
10.8.1 How to Create a Search Form ... 10-23
10.8.2 What Happens When You Use Parameter Methods ... 10-24
10.8.3 What Happens at Runtime ... 10-26
10.8.4 Creating a QBE Search Form With Results on a Separate Page ... 10-26
10.8.4.1 How to Create a Search Form and Separate Results Page ... 10-27
10.8.4.2 What Happens When You Create A Search Form ... 10-28
10.8.4.3 What You May Need to Know ... 10-29
10.8.5 Creating Search and Results on the Same Page ... 10-29
10.8.5.1 How To Create Search and Results on the Same Page ... 10-30
10.8.5.2 What Happens When Search and Results are on the Same Page ... 10-31
10.9 Conditionally Displaying the Results Table on a Search Page ... 10-32
10.9.1 How to Add Conditional Display Capabilities ... 10-33
10.9.2 What Happens When you Conditionally Display the Results Table ... 10-34

11 Using Complex UI Components

11.1 Introduction to Complex UI Components 11-1
11.2 Using Dynamic Menus for Navigation 11-2
11.2.1 How to Create Dynamic Navigation Menus 11-2
11.2.1.1 Creating a Menu Model 11-3
11.2.1.2 Creating the JSF Page for Each Menu Item 11-13
11.2.1.3 Creating the JSF Navigation Rules 11-16
11.2.2 What Happens at Runtime 11-17
11.2.3 What You May Need to Know About Menus 11-18
11.3 Using Popup Dialogs 11-19
11.3.1 How to Create Popup Dialogs 11-22
11.3.1.1 Defining a JSF Navigation Rule for Launching a Dialog 11-22
11.3.1.2 Creating the JSF Page That Launches a Dialog 11-23
11.3.1.3 Creating the Dialog Page and Returning a Dialog Value 11-25
11.3.1.4 Handling the Return Value 11-28
11.3.1.5 Passing a Value into a Dialog 11-29
11.3.2 How the SRDemo Popup Dialogs Are Created 11-30
11.3.3 What You May Need to Know About ADF Faces Dialogs 11-35
11.3.4 Other Information 11-35
11.4 Enabling Partial Page Rendering 11-35
11.4.1 How to Enable PPR 11-36
11.4.2 What Happens at Runtime 11-38
11.4.3 What You May Need to Know About PPR and Screen Readers 11-38
11.5 Creating a Multipage Process 11-38
11.5.1 How to Create a Process Train 11-40
11.5.1.1 Creating a Process Train Model 11-40
11.5.1.2 Creating the JSF Page for Each Train Node 11-45
11.5.1.3 Creating the JSF Navigation Rules 11-47
11.5.2 What Happens at Runtime 11-48
11.5.3 What You May Need to Know About Process Trains and Menus 11-48
11.6 Providing File Upload Capability 11-49
11.6.1 How to Support File Uploading on a Page 11-50
11.6.2 What Happens at Runtime 11-54
11.6.3 What You May Need to Know About ADF Faces File Upload 11-54
11.6.4 Configuring File Uploading Initialization Parameters 11-55
11.6.5 Configuring a Custom Uploaded File Processor 11-55
11.7 Creating Databound Dropdown Lists 11-56
11.7.1 How to Create a Dropdown List with a Fixed List of Values 11-56
11.7.2 What Happens When You Create a Dropdown List Bound to a Fixed List 11-58
11.7.3 How to Create a Dropdown List with a Dynamic List of Values 11-59
11.7.4 What Happens When You Create a Dropdown List Bound to a Dynamic List 11-60
11.7.5 How to Use Variables with Dropdown Lists 11-61
11.8 Creating a Databound Shuttle 11-62
11.8.1 How to Create a Databound Shuttle 11-63
11.8.2 What Happens at Runtime 11-70

12 Using Validation and Conversion

12.1 Introduction to Validation and Conversion 12-1
12.2 Validation, Conversion, and the Application Lifecycle 12-2
12.3 Adding Validation 12-3
12.3.1 How to Add Validation 12-3
12.3.1.1 Adding ADF Faces Validation 12-3
12.3.1.2 Adding ADF Model Validation 12-7
12.3.2 What Happens When You Create Input Fields Using the Data Control Palette 12-8
12.3.3 What Happens at Runtime 12-10
12.3.4 What You May Need to Know 12-11
12.4 Creating Custom JSF Validation 12-11
12.4.1 How to Create a Backing Bean Validation Method 12-12
12.4.2 What Happens When You Create a Backing Bean Validation Method 12-12
12.4.3 How to Create a Custom JSF Validator 12-13
12.4.4 What Happens When You Use a Custom JSF Validator 12-16
12.5 Adding Conversion 12-16
12.5.1 How to Use Converters 12-18
12.5.2 What Happens When You Create Input Fields Using the Data Control Palette 12-18
12.5.3 What Happens at Runtime 12-19
12.6 Creating Custom JSF Converters 12-19
12.6.1 How to Create a Custom JSF Converter 12-19
12.6.2 What Happens When You Use a Custom Converter 12-21
12.7 Displaying Error Messages 12-22
12.7.1 How to Display Server-Side Error Messages on a Page 12-23
12.7.2 What Happens When You Choose to Display Error Messages 12-23
12.8 Handling and Displaying Exceptions in an ADF Application 12-23
12.8.1 How to Change Exception Handling 12-24
12.8.2 What Happens When You Change the Default Error Handling 12-31

13 Adding ADF Bindings to Existing Pages

13.1 Introduction to Adding ADF Bindings to Existing Pages 13-1
13.2 Designing Pages for ADF Bindings 13-2
13.2.1 Creating the Page 13-2
13.2.2 Adding Components to the Page 13-3
13.2.3 Other Design Considerations 13-4
13.2.3.1 Creating Text Fields in Forms 13-4
13.2.3.2 Creating Tables 13-4
13.2.3.3 Creating Buttons and Links 13-4
13.2.3.4 Creating Lists 13-5
13.2.3.5 Creating Trees or Tree Tables 13-5
13.3 Using the Data Control Palette to Bind Existing Components 13-5
13.3.1 How to Add ADF Bindings Using the Data Control Palette 13-5
13.3.2 What Happens When You Use the Data Control Palette to Add ADF Bindings 13-7
13.4 Adding ADF Bindings to Text Fields 13-7
13.4.1 How to Add ADF Bindings to Text Fields 13-7
13.4.2 What Happens When You Add ADF Bindings to a Text Field 13-8

13.5 Adding ADF Bindings to Tables ... 13-8
13.5.1 How to Add ADF Bindings to Tables ... 13-8
13.5.2 What Happens When You Add ADF Bindings to a Table ... 13-10
13.6 Adding ADF Bindings to Actions ... 13-11
13.6.1 How to Add ADF Bindings to Actions ... 13-12
13.6.2 What Happens When You Add ADF Bindings to an Action ... 13-12
13.7 Adding ADF Bindings to Selection Lists ... 13-13
13.7.1 How to Add ADF Bindings to Selection Lists ... 13-13
13.7.2 What Happens When You Add ADF Bindings to a Selection List ... 13-13
13.8 Adding ADF Bindings to Trees and Tree Tables ... 13-14
13.8.1 How to Add ADF Bindings to Trees and Tree Tables ... 13-15
13.8.2 What Happens When You Add ADF Bindings to a Tree or Tree Table ... 13-15

14 Changing the Appearance of Your Application

14.1 Introduction to Changing ADF Faces Components ... 14-1
14.2 Changing the Style Properties of a Component ... 14-2
14.2.1 How to Set a Component's Style Attributes ... 14-2
14.2.2 What Happens When You Format Text ... 14-3
14.3 Using Skins to Change the Look and Feel ... 14-3
14.3.1 How to Use Skins ... 14-5
14.3.1.1 Creating a Custom Skin ... 14-6
14.3.1.2 Configuring an Application to Use a Skin ... 14-10
14.4 Internationalizing Your Application ... 14-11
14.4.1 How to Internationalize an Application ... 14-14
14.4.2 How to Configure Optional Localization Properties for ADF Faces ... 14-19

15 Optimizing Application Performance with Caching

15.1 About Caching ... 15-1
15.2 Using ADF Faces Cache to Cache Content ... 15-2
15.2.1 How to Add Support for ADF Faces Cache ... 15-6
15.2.2 What Happens When You Cache Fragments ... 15-7
15.2.2.1 Logging ... 15-7
15.2.2.2 AFC Statistics Servlet ... 15-7
15.2.2.3 Visual Diagnostics ... 15-8
15.2.3 What You May Need to Know ... 15-9

16 Testing and Debugging Web Applications

16.1 Getting Started with Oracle ADF Model Debugging ... 16-1
16.2 Correcting Simple Oracle ADF Compilation Errors ... 16-2
16.3 Correcting Simple Oracle ADF Runtime Errors ... 16-4
16.4 Understanding a Typical Oracle ADF Model Debugging Session ... 16-6
16.4.1 Turning on Diagnostic Logging ... 16-7
16.4.2 Creating an Oracle ADF Debugging Configuration ... 16-7
16.4.3 Understanding the Different Kinds of Breakpoints ... 16-8
16.4.4 Editing Breakpoints to Improve Control ... 16-9
16.4.5 Filtering Your View of Class Members ... 16-9

16.4.6 Communicating Stack Trace Information to Someone Else 16-10
16.5 Debugging the Oracle ADF Model Layer 16-10
16.5.1 Correcting Failures to Display Pages 16-11
16.5.1.1 Fixing Binding Context Creation Errors 16-11
16.5.1.2 Fixing Binding Container Creation Errors 16-12
16.5.2 Correcting Failures to Display Data 16-15
16.5.2.1 Fixing Executable Errors 16-15
16.5.2.2 Fixing Render Value Errors Before Submit 16-19
16.5.3 Correcting Failures to Invoke Actions and Methods 16-22
16.5.4 Correcting Page Validation Failures 16-25
16.6 Tracing EL Expressions 16-27

17 Working Productively in Teams

17.1 Using CVS with an ADF Project 17-1
17.1.1 Choice of Internal or External CVS Client 17-1
17.1.2 Preference Settings 17-1
17.1.3 File Dependencies 17-1
17.1.4 Use Consistent Connection Definition Names 17-2
17.1.5 General Advice for Committing ADF Work to CVS 17-2
17.1.5.1 Other Version Control Tips and Techniques 17-2
17.1.6 Check Out or Update from the CVS Repository 17-3
17.1.7 Special Consideration when Manually Adding Navigation Rules to the faces-config.xml File 17-3
17.2 General Advice for Using CVS with JDeveloper 17-3
17.2.1 Team-Level Activities 17-3
17.2.2 Developer-Level Activities 17-4
17.2.2.1 Typical Workflow When Checking Your Work Into CVS 17-4
17.2.2.2 Handling CVS Repository Configuration Files 17-5

18 Adding Security to an Application

18.1 Introduction to Security in Oracle ADF Web Applications 18-1
18.2 Specifying the JAZN Resource Provider 18-2
18.2.1 How To Specify the Resource Provider 18-2
18.2.2 What You May Need to Know About Oracle ADF Security and Resource Providers 18-3
18.3 Configuring Authentication Within the web.xml File 18-4
18.3.1 How to Enable J2EE Container-Managed Authentication 18-4
18.3.2 What Happens When You Use Security Constraints without Oracle ADF Security 18-8
18.3.3 How to Enable Oracle ADF Authentication 18-9
18.3.4 What Happens When You Use Security Constraints with Oracle ADF 18-11
18.4 Creating a Login Page 18-12
18.4.1 Wiring the Login and Error Pages 18-15
18.4.2 What Happens When You Wire the Login and Error Pages 18-16
18.5 Creating a Logout Page 18-17
18.5.1 Wiring the Logout Action 18-18
18.5.2 What Happens When You Wire the Logout Action 18-20

18.6 Implementing Authorization Using Oracle ADF Security 18-20
18.6.1 Configuring the Application to Use Oracle ADF Security Authorization 18-22
18.6.1.1 How to Configure Oracle ADF Security Authorization 18-22
18.6.1.2 What Happens When You Configure An Application to Use Oracle ADF Security 18-22
18.6.1.3 What You May Need to Know About the Authorization Property 18-23
18.6.2 Setting Authorization on ADF Binding Containers 18-23
18.6.3 Setting Authorization on ADF Iterator Bindings 18-23
18.6.4 Setting Authorization on ADF Attribute and MethodAction Bindings 18-24
18.6.5 What Happens When Oracle ADF Security Handles Authorization 18-24
18.7 Implementing Authorization Programmatically 18-25
18.7.1 Making User Information EL Accessible 18-25
18.7.1.1 Creating a Class to Manage Roles 18-26
18.7.1.2 Creating a Managed Bean for the Security Information 18-28
18.7.2 Integrating the Managed Bean with Oracle ADF Model 18-31
18.7.2.1 Creating a TopLink Named Query To Return a User Object 18-31
18.7.2.2 Create a Session Facade Method to Wrap the Named Query 18-32
18.7.2.3 Create a Page Definition to Make the Method an EL Accessible Object 18-32
18.7.2.4 Executing the Session Facade Method from the UserInfo Bean 18-34

19 Advanced TopLink Topics

19.1 Introduction to Advanced TopLink Topics 19-1
19.2 Using Advanced Parameters (databindings.cpx) 19-1
19.2.1 Performing Deletes First 19-2
19.2.2 Specifying the TopLink Session File 19-2
19.2.3 Specifying the Sequencing 19-3
19.3 Configuring Method Access for Relationship 19-3
19.4 Using sessions.xml with a TopLink Data Control 19-4
19.5 Using Multiple Maps with a TopLink Data Control 19-5
19.6 Compiling TopLink Classes with Specific JDK Versions 19-7

20 Creating Data Control Adapters

20.1 Introduction to the Simple CSV Data Control Adapter 20-1
20.2 Overview of Steps to Create a Data Control Adapter 20-2
20.3 Implement the Abstract Adapter Class 20-3
20.3.1 Location of JAR Files 20-3
20.3.2 Abstract Adapter Class Outline 20-4
20.3.3 Complete Source for the SampleDCAdapter Class 20-4
20.3.4 Implementing the initialize Method 20-7
20.3.5 Implementing the invokeUI Method 20-7
20.3.6 Implementing the getDefinition Method 20-8
20.4 Implement the Data Control Definition Class 20-9
20.4.1 Location of JAR Files 20-9
20.4.2 Data Control Definition Class Outline 20-9
20.4.3 Complete Source for the SampleDCDef Class 20-10
20.4.4 Creating a Default Constructor 20-13

20.4.5 Collecting Metadata from the User 20-13
20.4.6 Defining the Structure of the Data Control 20-15
20.4.7 Creating an Instance of the Data Control 20-16
20.4.8 Setting the Metadata for Runtime 20-16
20.4.9 Setting the Name for the Data Control 20-17
20.5 Implement the Data Control Class 20-17
20.5.1 Location of JAR Files 20-18
20.5.2 Data Control Class Outline 20-18
20.5.3 Complete Source for the SampleDataControl Class 20-19
20.5.4 Implementing the invokeOperation Method 20-21
20.5.4.1 About Calling processResult 20-23
20.5.4.2 Return Value for invokeOperation 20-23
20.5.5 Implementing the getName Method 20-23
20.5.6 Implementing the release Method 20-24
20.5.7 Implementing the getDataProvider Method 20-24
20.6 Create any Necessary Supporting Classes 20-24
20.7 Create an XML File to Define Your Adapter 20-25
20.8 Build Your Adapter 20-26
20.9 Package and Deploy Your Adapter to JDeveloper 20-26
20.10 Location of Javadoc Information 20-28
20.11 Contents of Supporting Files 20-29
20.11.1 sampleDC.xsd 20-29
20.11.2 CSVHandler Class 20-29
20.11.3 CSVParser 20-37

21 Working with Web Services

21.1 What are Web Services 21-1
21.1.1 SOAP 21-2
21.1.2 WSDL 21-2
21.1.3 UDDI 21-2
21.1.4 Web Services Interoperability 21-3
21.2 Creating Web Service Data Controls 21-4
21.2.1 How to Create a Web Service Data Control 21-4
21.3 Securing Web Service Data Controls 21-5
21.3.1 WS-Security Specification 21-5
21.3.2 Creating and Using Keystores 21-6
21.3.2.1 How to Create a Keystore 21-6
21.3.2.2 How to Request a Certificate 21-7
21.3.2.3 How to Export a Public Key Certificate 21-8
21.3.3 Defining Web Service Data Control Security 21-9
21.3.3.1 How to Set Authentication 21-9
21.3.3.2 How to Set Digital Signatures 21-12
21.3.3.3 How to Set Encryption and Decryption 21-13
21.3.3.4 How to Use a Key Store 21-13

22 Deploying ADF Applications

22.1 Introduction to Deploying ADF Applications ... 22-1
22.2 Deployment Steps ... 22-2
22.3 Deployment Techniques ... 22-7
22.4 Deploying Applications Using Ant ... 22-8
22.5 Deploying the SRDemo Application ... 22-8
22.6 Deploying to Oracle Application Server ... 22-9
22.6.1 Oracle Application Server Versions Supported ... 22-9
22.6.2 Oracle Application Server Release 2 (10.1.2) Deployment Notes ... 22-9
22.6.3 Oracle Application Server Deployment Methods ... 22-10
22.6.4 Oracle Application Server Deployment to Test Environments ("Automatic Deployment") ... 22-10
22.6.5 Oracle Application Server Deployment to Clustered Topologies ... 22-11
22.7 Deploying to JBoss ... 22-11
22.7.1 JBoss Versions Supported ... 22-11
22.7.2 JBoss Deployment Notes ... 22-11
22.7.3 JBoss Deployment Methods ... 22-13
22.8 Deploying to WebLogic ... 22-13
22.8.1 WebLogic Versions Supported ... 22-13
22.8.2 WebLogic Versions 8.1 and 9.0 Deployment Notes ... 22-13
22.8.3 WebLogic 8.1 Deployment Notes ... 22-13
22.8.4 WebLogic 9.0 Deployment Notes ... 22-14
22.8.5 WebLogic Deployment Methods ... 22-14
22.9 Deploying to WebSphere ... 22-14
22.9.1 WebSphere Versions Supported ... 22-14
22.9.2 WebSphere Deployment Notes ... 22-15
22.9.3 WebSphere Deployment Methods ... 22-15
22.10 Deploying to Tomcat ... 22-15
22.10.1 Tomcat Versions Supported ... 22-15
22.10.2 Tomcat Deployment Notes ... 22-15
22.11 Deploying to Application Servers That Support JDK 1.4 ... 22-16
22.11.1 Switching Embedded OC4J to JDK 1.4 ... 22-16
22.12 Installing ADF Runtime Library on Third-Party Application Servers ... 22-17
22.12.1 Installing the ADF Runtime Libraries from JDeveloper ... 22-17
22.12.2 Configuring WebSphere 6.0.1 to Run ADF Applications ... 22-20
22.12.2.1 Source for install_adflibs_1013.sh Script ... 22-21
22.12.2.2 Source for install_adflibs_1013.cmd Script ... 22-23
22.12.3 Installing the ADF Runtime Libraries Manually ... 22-24
22.12.3.1 Installing the ADF Runtime Libraries from a Zip File ... 22-26
22.12.4 Deleting the ADF Runtime Library ... 22-27
22.13 Verifying Deployment and Troubleshooting ... 22-27
22.13.1 How to Test Run Your Application ... 22-28
22.13.2 "Class Not Found" or "Method Not Found" Errors ... 22-28
22.13.3 Application Is Not Using data-sources.xml File on Target Application Server ... 22-28
22.13.4 Using jazn-data.xml with the Embedded OC4J Server ... 22-28

A Reference ADF XML Files

A.1 About the ADF Metadata Files ... A-1
A.2 ADF File Overview Diagram ... A-2
A.2.1 Oracle ADF Data Control Files ... A-3
A.2.2 Oracle ADF Data Binding Files ... A-3
A.2.3 Oracle ADF Faces and Web Configuration Files ... A-4
A.3 ADF File Syntax Diagram ... A-4
A.4 DataControls.dcx ... A-7
A.4.1 Syntax of the DataControls.dcx File ... A-7
A.4.2 Sample of the DataControls.dcx File ... A-9
A.4.3 Sample of the adfm.xml File ... A-11
A.5 Structure Definition Files ... A-11
A.5.1 Syntax for the Structure Definition for a JavaBean ... A-12
A.5.2 Sample Structure Definition for the <sessionbeanname>.xml File ... A-14
A.5.3 Sample Structure Definition for the <entitybeanname>.xml File ... A-16
A.5.4 Collection and SingleValue Sample Files ... A-17
A.6 DataBindings.cpx ... A-18
A.6.1 DataBindings.cpx Syntax ... A-18
A.6.2 DataBindings.cpx Sample ... A-20
A.7 <pageName>PageDef.xml ... A-21
A.7.1 PageDef.xml Syntax ... A-21
A.7.2 PageDef.xml Sample for a Method That Returns a String ... A-30
A.7.3 PageDef.xml Sample for a Method that Returns a Collection ... A-30
A.8 web.xml ... A-31
A.8.1 Tasks Supported by the web.xml File ... A-33
A.8.1.1 Configuring for State Saving ... A-33
A.8.1.2 Configuring for Application View Caching ... A-34
A.8.1.3 Configuring for Debugging ... A-34
A.8.1.4 Configuring for File Uploading ... A-35
A.8.1.5 Configuring for ADF Model Binding ... A-35
A.8.1.6 Other Context Configuration Parameters for JSF ... A-36
A.8.1.7 What You May Need to Know ... A-36
A.9 j2ee-logging.xml ... A-37
A.9.1 Tasks Supported by the j2ee-logging.xml ... A-37
A.9.1.1 Change the Logging Level for Oracle ADF Packages ... A-37
A.9.1.2 Redirect the Log Output ... A-37
A.9.1.3 Change the Location of the Log File ... A-38
A.10 faces-config.xml ... A-38
A.10.1 Tasks Supported by the faces-config.xml ... A-38
A.10.1.1 Registering a Render Kit for ADF Faces Components ... A-39
A.10.1.2 Registering a Phase Listener for ADF Binding ... A-39
A.10.1.3 Registering a Message Resource Bundle ... A-40
A.10.1.4 Configuring for Supported Locales ... A-40
A.10.1.5 Creating Navigation Rules and Cases ... A-40
A.10.1.6 Registering Custom Validators and Converters ... A-42
A.10.1.7 Registering Managed Beans ... A-42

A.11 adf-faces-config.xml A-44
A.11.1 Tasks Supported by adf-faces-config.xml A-44
A.11.1.1 Configuring Accessibility Levels A-45
A.11.1.2 Configuring Currency Code and Separators for Number Groups and Decimals A-45
A.11.1.3 Configuring For Enhanced Debugging Output A-46
A.11.1.4 Configuring for Client-Side Validation and Conversion A-46
A.11.1.5 Configuring the Language Reading Direction A-46
A.11.1.6 Configuring the Skin Family A-47
A.11.1.7 Configuring the Output Mode A-47
A.11.1.8 Configuring the Number of Active ProcessScope Instances A-47
A.11.1.9 Configuring the Time Zone and Year Offset A-47
A.11.1.10 Configuring a Custom Uploaded File Processor A-48
A.11.1.11 Configuring the Help Site URL A-48
A.11.1.12 Retrieving Configuration Property Values From adf-faces-config.xml A-48
A.12 adf-faces-skins.xml A-49
A.12.1 Tasks Supported by adf-faces-skins.xml A-49

B Reference ADF Binding Properties

B.1 EL Properties of Oracle ADF Bindings B-1

Index

Preface

Welcome to the Oracle Application Development Framework Developer's Guide!

Audience

This manual is intended for software developers who are creating and deploying applications using the Oracle Application Development Framework with JavaServer Faces, ADF Faces, TopLink Java Objects, and EJB 3.0 session beans.

Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible, with good usability, to the disabled community. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Accessibility standards will continue to evolve over time, and Oracle is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For more information, visit the Oracle Accessibility Program Web site at

http://www.oracle.com/accessibility/

Accessibility of Code Examples in Documentation

Screen readers may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, some screen readers may not always read a line of text that consists solely of a bracket or brace.

Accessibility of Links to External Web Sites in Documentation

This documentation may contain links to Web sites of other companies or organizations that Oracle does not own or control. Oracle neither evaluates nor makes any representations regarding the accessibility of these Web sites.

TTY Access to Oracle Support Services

Oracle provides dedicated Text Telephone (TTY) access to Oracle Support Services within the United States of America 24 hours a day, seven days a week. For TTY support, call 800.446.2398.

Related Documents

For more information, see the following:

- *Oracle Application Development Framework Developer's Guide for Forms/4GL Developers* for enterprise developers who are familiar with 4GL tools like Oracle Forms, PeopleTools, SiebelTools, and Visual Studio, and who need to create and deploy database-centric J2EE applications with a service-oriented architecture using the Oracle Application Development Framework (Oracle ADF). This guide explains how to build these applications using ADF Business Components, JavaServer Faces, and ADF Faces: the same technology stack Oracle employs to build the web-based Oracle EBusiness Suite.
- *Oracle JDeveloper 10g Release Notes*, included with your JDeveloper 10g installation, and on Oracle Technology Network
- *Oracle JDeveloper 10g Online Help*
- *Oracle Application Server 10g Release Notes*
- *Oracle Application Server 10g Documentation Library* available on CD-ROM and on Oracle Technology Network

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

Part I

Getting Started with Oracle ADF Applications

Part I contains the following chapters:

- Chapter 1, "Introduction to Oracle ADF Applications"
- Chapter 2, "Oracle ADF Service Request Demo Overview"
- Chapter 3, "Building and Using Application Services"

1 Introduction to Oracle ADF Applications

This chapter describes the architecture and key functionality of the Oracle Application Development Framework and highlights the typical development process for using JDeveloper 10*g* Release 3 (10.1.3) to build web applications using Oracle ADF, Enterprise JavaBeans, Oracle TopLink, and JSF.

This chapter includes the following sections:

- Section 1.1, "Overview of Oracle Application Development Framework"
- Section 1.2, "Development Process with Oracle ADF and JavaServer Faces"

1.1 Overview of Oracle Application Development Framework

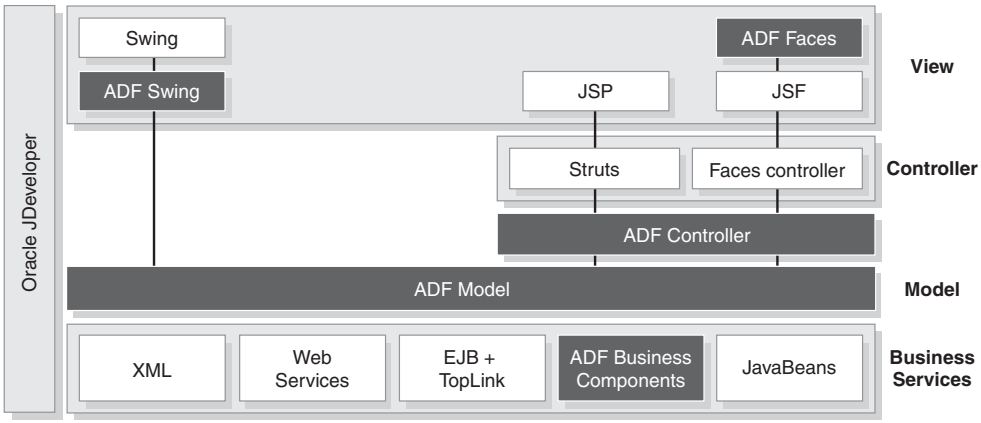
The Oracle Application Development Framework (Oracle ADF) is an end-to-end application framework that builds on J2EE standards and open-source technologies to simplify and accelerate implementing service-oriented applications. If you develop enterprise solutions that search, display, create, modify, and validate data using web, wireless, desktop, or web services interfaces, Oracle ADF can simplify your job. Used in tandem, Oracle JDeveloper 10*g* 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.

1.1.1 Framework Architecture and Supported Technologies

In line with community best practices, applications you build using Oracle ADF achieve a clean separation of concerns by adhering to a model, view, controller architecture. Figure 1–1 illustrates where each ADF module fits in this architecture. The core module in the framework is *Oracle ADF Model*, a declarative data binding facility that implements the JSR-227 specification. The Oracle ADF Model layer enables a unified approach to bind any user interface to any business service with no code. The other modules Oracle ADF comprises are:

- *Oracle ADF Controller* integrates Struts and JSF with Oracle ADF Model
- *Oracle ADF Faces* offers a library of components for web applications built with JSF
- *Oracle ADF Swing* extends Oracle ADF Model to desktop applications built with Swing
- *Oracle ADF Business Components* simplifies building business services for developers familiar with 4GL tools like Oracle Forms.

Figure 1–1 Simple ADF Architecture



1.1.1.1 View Layer Technologies Supported

In the view layer of your application, where you design the web user interface, you can develop using either classic JavaServer Pages (JSP) or the latest JavaServer Faces (JSF) standard. Alternatively, you can choose the polish and interactivity of a desktop UI, and develop using any off-the-shelf Swing components or libraries to ensure just the look and feel you need. Whatever your choice, you work with WYSIWYG visual designers and drag-and-drop data binding. One compelling reason to choose JSF is the comprehensive library of nearly one hundred JSF components that the ADF Faces module provides.

ADF Faces components include sophisticated features like look and feel "skinning" and the ability to incrementally update only the bits of the page that have changed using the latest AJAX programming techniques. The component library supports multiple JSF render kits to allow targeting users with web browsers and roaming users with PDA telnet devices. In short, these components dramatically simplify building highly attractive and functional web and wireless UIs without getting your hands "dirty" with HTML and JavaScript.

1.1.1.2 Controller Layer Technologies Supported

In the controller layer, where handling page flow of your web applications is a key concern, Oracle ADF integrates both with the popular Apache Struts framework and the built-in page navigation functionality included in JSF. In either case, JDeveloper offers visual page flow diagrammers to design your page flow, and the ADF Controller module provides appropriate plug-ins to integrate the ADF Model data binding facility with the controller layer's page processing lifecycle.

1.1.1.3 Business Services Technologies Supported by ADF Model

In the model layer, Oracle ADF Model implements the JSR-227 service abstraction called the *data control* and provides out-of-box data control implementations for the most common business service technologies. Whichever ones you employ, JDeveloper and Oracle ADF work together to provide you a declarative, drag-and-drop data binding experience as you build your user interfaces. Supported technologies include:

- *Enterprise JavaBeans (EJB) Session Beans*

  Since most J2EE applications require transactional services, EJB session beans are a logical choice because they offer declarative transaction control. Behind the EJB session bean facade for your business service, you use plain old Java objects (POJOs) or EJB entity beans to represent your business domain objects. JDeveloper

offers integrated support for creating EJB session beans, generating initial session facade implementations, and creating either Java classes or entity beans. You can also use Oracle TopLink in JDeveloper to configure the object/relational mapping of these classes.

- *JavaBeans*

  You can easily work with any Java-based service classes as well, including the ability to leverage Oracle TopLink mapping if needed.

- *Web Services*

  When the services your application requires expose standard web services interfaces, just supply Oracle ADF with the URL to the relevant Web Services Description Language (WSDL) for the service endpoints and begin building user interfaces that interact with them and present their results.

- *XML*

  If your application needs to interact with XML or comma-separated values (CSV) data that is not exposed as a web service, this is easy to accomplish, too. Just supply the provider URL and optional parameters and you can begin to work with the data.

- *ADF Application Modules*

  These service classes are a feature of the ADF Business Components module, and expose an updateable dataset of SQL query results with automatic business rules enforcement.

1.1.1.4 Recommended Technologies for J2EE Enterprise Developers

The remainder of this guide focuses attention on using Oracle ADF with the technologies Oracle recommends to J2EE developers building new web applications: JavaServer Faces for the view and controller layers, and the combination of an EJB session bean with mapped Java classes for the business service implementation. However, this chapter begins with a very simple Oracle ADF application built with these technologies to acquaint you with typical development process.

Note: If you are a developer coming to J2EE development with experience in 4GL tools like Oracle Forms, Oracle Designer, Visual Basic, PowerBuilder, and so on, Oracle recommends that you take advantage of the additional declarative development features offered by the Oracle ADF Business Components module. *Oracle ADF Developer's Guide for Forms/4GL Developers* covers using Oracle ADF with additional framework functionality in the business services tier using this module. You can access the developer's guide for Forms/4GL developers from `http://www.oracle.com/technology/products/adf/learnadf.html`.

1.1.2 Declarative Development with Oracle ADF and JavaServer Faces

For seasoned Java developers, choosing to develop declaratively instead of coding admittedly takes some getting used to. However, most developers will acknowledge a true time-saver, and they are also likely to have some exposure to declarative techniques through their experience with frameworks like Spring or Apache Struts and tag libraries like the JSP Standard Tag Library (JSTL). JavaServer Faces

incorporates similar declarative functionality and Oracle ADF complements it by adding declarative data binding to the mix.

1.1.2.1 Declarative J2EE Technologies You May Have Already Used

Using the Spring Framework, developers configure the instantiation of JavaBeans through a beans XML file, declaratively specifying dependencies between them. At runtime, generic framework code reads the XML file, instantiates the beans as directed, and resolves the dependencies between beans. This design pattern is commonly known as *dependency injection*, and Spring provides a declaratively configured way to leverage its generic implementation to set up your application's beans.

Using the Struts `struts-config.xml` file, developers configure the mapping of HTTP requests to action handler classes and other page flow information. At runtime, the generic Struts front-controller servlet uses the information contained in the configuration XML file to route the requests as directed. When you use Struts, you leave the request routing to the declaratively configured Struts infrastructure, and concentrate on writing the interesting code that will handle particular requests.

Using the JSTL tag library, developers indicate the model data to iterate over and present on the page using declarative expressions, as shown in this snippet:

```
<c:when test="${not empty UserList}">
  <c:forEach var="user" items="${UserList.selectedUsers}">
    <tr>
      <td><c:out value="${user.name}"/></td>
      <td><c:out value="${user.email}"/>
    </tr>
  </c:forEach>
</c:when>
```

Rather than resorting to an unmaintainable mix of Java scriptlet code and tags in their page, developers embed expressions like `${not empty UserList}`, `${UserList.selectedUsers}`, and `${user.name}` into tag attributes. At runtime a generic expression evaluator returns the `boolean`-, `List`- and `String`-valued results, respectively, performing work to access beans and their properties without writing code. This same declarative *expression language*, nicknamed "EL," that originally debuted as part of the JSTL tag library has been improved and incorporated into the current versions of the JSP and JSF standards.

1.1.2.2 JSF Offers Dependency Injection, Page Handling, EL and More

JavaServer Faces simplifies building web user interfaces by introducing web UI components that have attributes, events, and a consistent runtime API. Instead of wading knee-high through tags and script, you assemble web pages from libraries of off-the-shelf, data-aware components that adhere to the JSF standard. As part of fulfilling their mission to simplify web application building, the industry experts who collaborated to design the JavaServer Faces standard incorporated numerous declarative development techniques. In fact, JSF supports all three of the facilities discussed above: instantiation and dependency injection for beans, page request handling and page navigation, and use of the standard expression language.

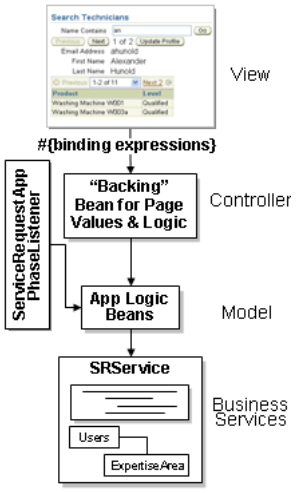
At runtime, the value of a JSF component is determined by its `value` attribute. While a component can have static text as its value, typically the `value` attribute will contain an EL expression that the runtime infrastructure evaluates to determine what data to display. For example, an `outputText` component that displays the name of the currently logged-in user might have its `value` attribute set to the expression `#{UserInfo.name}` to accomplish this. Since any attribute of a component can be assigned a value using an EL expression, it's easy to build dynamic, data-driven user

interfaces. For example, you could hide a component when a collection of beans you need to display is empty by setting the component's `rendered` attribute to a boolean-valued EL expression like `#{not empty UserList.selectedUsers}`. If the list of selected users in the bean named `UserList` is empty, then the `rendered` attribute evaluates to false and the component disappears from the page.

To simplify maintenance of controller-layer application logic, JSF offers a declarative bean creation mechanism similar to the Spring Framework. To use it, you configure your beans in the JSF `faces-config.xml` file. They are known as "managed beans" since the JSF runtime manages instantiating them on demand when any EL expression references them for the first time. JSF also offers a declarative dependency injection feature. Managed beans can have managed properties whose runtime value is assigned by the JSF runtime based on a developer-supplied EL expression. Managed properties can depend on other beans that, in turn, also have managed properties of their own, and the JSF runtime will guarantee that the "tree" of related beans is created in the proper order.

As illustrated in Figure 1–2, JSF managed beans serve two primary roles.

Figure 1–2 Basic Architecture of a JSF Application



Request-scoped managed beans that are tightly related to a given page are known colloquially as "backing" beans, since they support the page at runtime with properties and methods. The relationship between a UI component in the page and the backing bean properties and methods is established by EL expressions in appropriate attributes of the component like:

- `value="#{`*`expr`*`}"`

  References a property with data to display or modify

- `action="#{`*`expr`*`}"`

  References a method to handle events

- `binding="#{`*`expr`*`}"`

  References a property holding a corresponding instance of the UI component that you need to manipulate programmatically — show/hide, change color, and so on.

Think of managed beans that aren't playing the role of a page's backing bean simply as "application logic beans." They contain code and properties that are not specific to a single page. While not restricted to this purpose, they sometimes function as business

service wrappers to cache method results in the controller layer beyond a single request and to centralize pre- or post-processing of business service methods that might be used from multiple pages.

In addition to using managed beans, you can also write application code in a so-called `PhaseListener` class to augment any of the standard processing phases involved in handling a request for a JSF page. These standard steps that the JSF runtime goes through for each page are known as the "lifecycle" of the page. Most real-world JSF applications will end up customizing the lifecycle by implementing a custom phase listener of some kind, typically in order to perform tasks like preparing model data for rendering when a page initially displays, among other things.

1.1.2.3 Oracle ADF Further Raises the Level of Declarative Development for JSF

The Oracle ADF Model layer follows the same declarative patterns as other J2EE technologies, by using XML configuration files to drive generic framework facilities. The only interesting difference is that ADF Model focuses on adding value in the data binding layer. It implements the two concepts in JSR-227 that enable decoupling 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 its public interface. This includes information about the properties, methods, and types involved. At design time, visual tools leverage the service metadata to let you bind your UI components declaratively to any public member of a data control. At runtime, the generic Oracle ADF Model layer reads the information describing your data controls and bindings from appropriate XML files and implements the two-way "wiring" that connects your user interface to your service. This combination enables three key benefits:

- You write less code, so there are fewer lines to test and debug.
- You work the same way with any UI and business service technologies.
- You gain useful runtime features that you don't have to code yourself.

There are three basic kinds of binding objects that automate the key aspects of data binding that all enterprise applications require:

- *Action bindings* invoke business service methods to perform a task or retrieve data.
- *Iterator bindings* keep track of the current row in a data collection.
- *Attribute bindings* connect UI components to attributes in a data collection.

Typically UI components like hyperlinks or buttons use action bindings. This allows the user to click on the component to invoke a business service without code. UI components that display data use attribute bindings. Iterator bindings simplify building user interfaces that allow scrolling and paging through collections of data and drilling-down from summary to detail information.

The group of bindings supporting the UI components on a page are described in a page-specific XML file called the *page definition file*. Generic bean factories provided by ADF Model use this file at runtime to instantiate the page's bindings. These bindings are held in a request-scoped Map called the *binding container* accessible during each page request using the EL expression `#{bindings}`. This expression always evaluates to the binding container for the current page. Figure 1–3 shows how EL value binding expressions relate the UI components in a page to the binding objects in the binding container.

Figure 1–3 Bindings in the Binding Container Are EL Accessible at Runtime

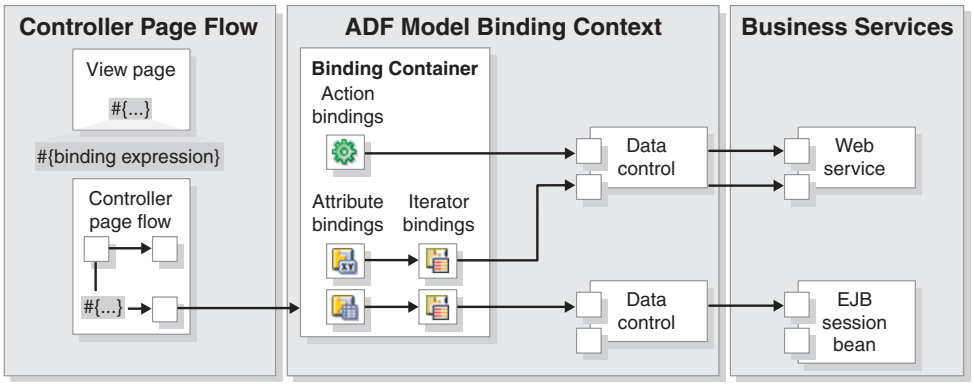
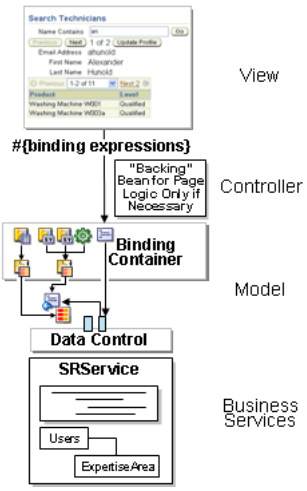


Figure 1–4 illustrates the architecture of a JSF application when leveraging ADF Model for declarative data binding. When you combine Oracle ADF Model with JavaServer Faces, it saves you from having to write a lot of the typical managed bean code that would be required for real-world applications. In fact, many pages you build won't require a "backing bean" at all, unless you perform programmatic controller logic that can't be handled by a built-in action or service method invocation (which ADF Model can do without code for you). You can also do away with any application logic beans that wrap your business service, since the ADF Model's data control implements this functionality for you. And finally, you can often avoid the need to write any custom JSF phase listeners because ADF Model offers a generic JSF phase listener that performs most of the common operations you need in a declarative way based on information in your page definition metadata.

Figure 1–4 Architecture of a JSF Application Using ADF Model Data Binding



1.1.3 Key ADF Binding Features for JSF Development

With the fundamentals of ADF Model data binding for JSF applications described, it's time to describe the full breadth of added-value functionality that ADF Model brings to the table. The following sections give an overview of the key functional areas that further improve your development productivity.

1.1.3.1 Comprehensive JDeveloper Design-Time Support

JDeveloper's comprehensive design- time support for ADF Model declarative data binding includes:

- *Data Control Wizards*

  Quickly create a data control for EJB session beans, Java service classes, web services, XML or CSV data from a URL, and ADF application modules. When additional information is required, helpful wizards guide you step by step.

- *Data Control Palette*

  Visualize all business services that you have exposed as data controls and drag service properties, methods, method parameters, and method results to create appropriate bound user interface elements. Easily create read-only and editable forms, tables, master/detail displays, and individual bound UI components including single and multiselect lists, checkboxes, radio groups, and so on. Creating search forms, data creation pages, and parameter forms for invoking methods is just as easy. If your process involves collaboration with page designers in another team, you can drop attributes onto existing components in the page to bind them after the fact. In addition to the UI components created, appropriate declarative bindings are created and configured for you in the page definition file with robust undo support so you can experiment or evolve your user interface with confidence that your bindings and UI components will stay in sync.

- *Page Definition Editor*

  Visualize page definition metadata in the Structure window and configure declarative binding properties using the appropriate editor or the Property Inspector. Create new bindings by inserting them into the structure where desired.

- *Binding Metadata Code Insight*

  Edit binding metadata with context-sensitive, XML schema-driven assistance on the structure and valid values. Visualize page definition metadata in the Structure window and configure declarative binding properties using the appropriate editor or the Property Inspector.

1.1.3.2 More Sophisticated UI Functionality Without Coding

The JSF reference implementation provides a bare-bones set of basic UI components that includes basic HTML input field types and a simple table display, but these won't take you very far when building real-world applications. The ADF Model layer implements several features that work hand-in-hand with the more sophisticated UI components in the Oracle ADF Faces library to make quick work of the rich functionality your end users crave, including:

- *More Sophisticated Table Model*

  Tables are a critical element of enterprise application UIs. By default, JSF doesn't support paging or sorting in tables. The ADF Faces table and the ADF Model table binding cooperate to display pageable, editable or read-only, tables with sorting on any column.

- *Key-Based Current Selection Tracking*

  One of the most common tasks of web user interfaces is presenting lists of information and allowing the user to scroll through them or to select one or more entries in the list. The ADF Model iterator binding simplifies tracking the selected row in a robust way, using row keys instead of relying on positional indicators that can change when data is refreshed and positions have changed. In concert with the

ADF Faces table and multiselection components, it's easy to work with single or multiple selections, and build screens that navigate master/detail information.

- *Declarative Hierarchical Tree Components and Grids*

  Much of the information in enterprise applications is hierarchical, but JSF doesn't support displaying or manipulating hierarchical data out of the box. The ADF Model layer provides hierarchical bindings that you can configure declaratively and use with the ADF Faces tree or hierarchical grid components to implement interactive user interfaces that present data in the most intuitive way to your users.

- *More Flexible Models for Common UI Components*

  Even simple components like the checkbox can be improved upon. By default, JSF supports binding a checkbox only to boolean properties. ADF Model adds the ability to map the checkbox to any combination of true or valid values your data may present. List components are another area where ADF Model excels. The valid values for the list can come from any data collection in a data control and the list can perform updates or be used for row navigation, depending on your needs. The ADF Model list binding also makes null-handling easy by optionally adding a translatable "<No Selection>" choice to the list.

1.1.3.3 Centralize Common Functionality in Layered Model Metadata

ADF Model can improve the reuse of several aspects of application functionality by allowing you to associate layered metadata with the data control structure definitions that can be reused by any page presenting their information. Examples of functionality that ADF Model allows you to reuse includes:

- *Translatable Prompts, Tooltips, and Format Masks*

  JSF supports a simple mechanism to reference translatable strings in resource bundles, but it has no knowledge about what the strings are used for and no way to associate the strings with specific business domain objects. With ADF Model, you can associate translatable prompts, tooltips, and format masks with any attribute of any data type used in the data control service interface so that the attribute's data is presented in a consistent, locale-sensitive way on every page where it appears.

- *Declarative Validation*

  JSF supports validators that can be associated with a UI component; however, it offers no mechanism to simplify validating the same business domain data in a consistent way on every screen where it's used. With ADF Model, you can associate an extensible set of validator objects with the data control structure definition metadata so that the validations will be enforced consistently, regardless of which page the user employs to enter or modify the object's data.

- *Declarative Security*

  JSF has no mechanism for integrating authorization information with UI components. With ADF Model, you can associate user or role authorization information with each attribute in the data control structure definition metadata so that pages can display the information consistently only to users authorized to see it.

1.1.3.4 Simplified Control Over Page Lifecycle

JSF rigorously defines the page processing lifecycle, but for some very common tasks it requires you to write code in your own phase listener to implement it. What's more,

until a future version of the JSF specification, phase listeners are global in nature, requiring you to write conditional code based on the current page's name when the functionality applies only to a specific page. The ADF Model and ADF Controller layers cooperate to simplify per-page control over the most common things you would typically code in a custom phase listener, including:

- *Declarative Method Invocation*

  Configure business service method invocations with EL expression-based parameter passing, and bind to method results with options to cache results until method parameters change to avoid unnecessary requerying of data. You can have methods invoked by the press of a command component like a link or button, or configure your page definition to automatically invoke the method at an appropriate phase of the JSF lifecycle

- *Declarative Page Lifecycle Control*

  Declaratively configure an iterator binding to refresh its data during a specific JSF lifecycle phase, and optionally provide a conditional EL expression for finer control over when that refresh is desired. You have the same control over when any automatic method invocation should invoke its method as well.

- *Centralized Error Reporting*

  Customize the error reporting approach for your application in a single point instead of on each page.

1.2 Development Process with Oracle ADF and JavaServer Faces

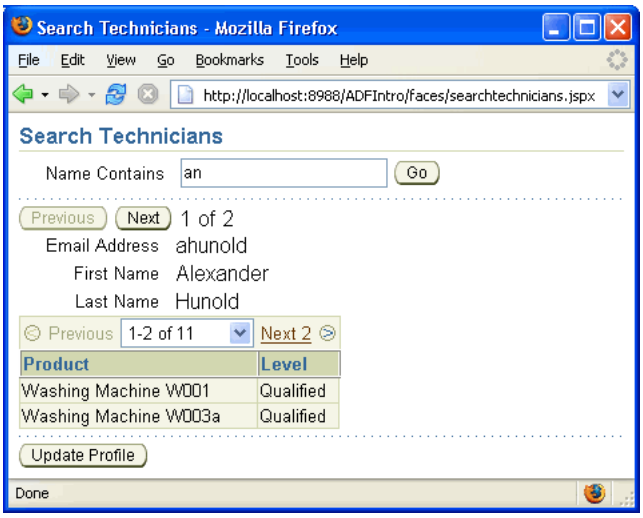
In this section, describes a simple example to acquaint you with the typical development process of building an Oracle ADF application with JavaServer Faces. This information is intended as a high-level overview of the basic workflow of J2EE application building with Oracle ADF.

1.2.1 Overview of the Steps for Building an Application

Our example is based on a highly-simplified version of the Service Request tracking system (the SRDemo sample), the real-world sample application used throughout the remainder of this guide. In the Service Request tracking system, external users log requests for technical assistance with products they've purchased. Internal users try to assist the customers in the area in which they have expertise. This introduction focuses on the basics, and examines a small slice of the system's functionality related to users and their areas of technical expertise.

You'll examine the steps involved in building a simple JSF page like the one you see in Figure 1–5 that allows the end user to browse for users by name, scroll through the results, and for each user found, see their areas of technical expertise.

Figure 1–5 Simple Browse Users Page with Search and Master/Detail Data



1.2.1.1 Starting by Creating a New Application

The first step in building a new application is to assign it a name and to specify the directory where its source files will be saved. Selecting Application from the JDeveloper New Gallery launches the Create Application dialog shown in Figure 1–6. Here you give the application a name, set a working directory, and provide a package prefix for the classes you'll create in the application. Suppose that you enter a package prefix of `oracle.srdemo` so that, by default, all of the classes will be created in packages whose names will begin with `oracle.srdemo.*`. Since you will be building a web application using JSF, EJB, and TopLink, Figure 1–6 shows the corresponding application template selected from the list. This application template is set up to create separate projects named Model and ViewController with appropriate technologies selected to build the respective layers of the application.

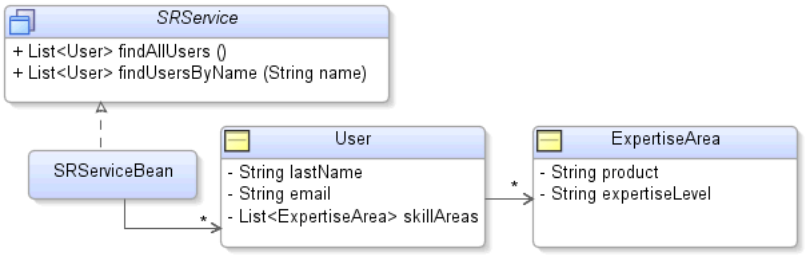
Figure 1–6 Creating a New Application Using an Application Template

1.2.1.2 Building the Business Service in the Model Project

You will typically start by building your business service interface, which by default is done in the project named Model. Your Model project will comprise an EJB 3.0 session bean to function as the service facade, and Java classes that represent the business domain objects you need to work with. The model doesn't need to be functionally complete to proceed on to the subsequent steps of developing the UI, but defining the service interface forces you to think about the data the view layer will need and the information it may need to supply as parameters to your service methods to complete the job. Since you'll want to work with users and areas of expertise, the Java classes named `User` and `ExpertiseArea` are created. Each class will contain properties to reflect the data needed to represent users and areas of expertise.

Based on the requirements, suppose the business service needs to support finding users by name. For this purpose, you can use the EJB Session Bean wizard to create a stateless EJB 3.0 session bean using a container-managed transaction. Next you can add a method called `findUsersByName()` to its local interface that accepts the matching pattern as a parameter called `name`. To clearly communicate the type of the result and obtain the best compile-time type checking possible, it is best practice to declare the return type of the method to be `List<User>`, a strongly typed list of User beans. Finally, you can write the method in the `SRServiceBean` class that implements the service interface. Figure 1–7 shows what the service and its classes look like in the Java class diagram in JDeveloper. You can see that the class also contains the useful `findAllUsers()` method to return all users if needed.

Figure 1–7 SRService Session Bean Facade and Supporting Domain Classes



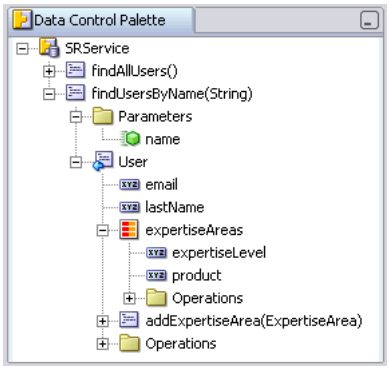
Because of the clean separation that ADF Model affords between the service and the user interface, the remaining steps to build the page depend only on the service interface, not its implementation. You can begin with a service that returns static test data, but eventually you will want to map the `User` and `ExpertiseArea` classes to appropriate tables in your database. This is where Oracle TopLink excels, and JDeveloper's integrated support for configuring your TopLink session and mappings makes quick work of the task. If you already have database tables with similar structure to your classes, Oracle TopLink can "automap" the classes to the tables for you, and then you can adjust the results as needed. If the database tables do not exist yet, you can use JDeveloper's database diagrammer to design them before performing the mapping operation. To implement the `findUsersByName()` method, you will create a named query as part of the `User` mapping descriptor and provide the criteria required to retrieve users matching a name supplied as a parameter. At runtime, the Oracle TopLink runtime handles retrieving the results of the parameterized query from the database based on XML-driven object/relational mapping information.

1.2.1.3 Creating a Data Control for Your Service to Enable Data Binding

With the business service in place, you can begin to think about building the user interface. The first step in enabling drag-and-drop data binding for the business service is to create a data control for it. Creating the data control publishes the service

interface to the rest of the Oracle ADF Model design time using JSR-227 service and structure descriptions. To create a data control, you just drag the `SRServiceBean` class onto JDeveloper's Data Control Palette. Figure 1–8 shows the Data Control Palette following this operation. You can see it reflects all of the service methods, any parameters they expect, and the method return types. For the `findUsersByName()` method, you can see that it expects a `name` parameter and that its return type contains beans of type `User`. The nested `email`, `lastName`, and `expertiseAreas` properties of the user are also displayed. Since `expertiseAreas` is a collection-typed property (of type `List<ExpertiseArea>`), you also see its nested properties. The Operations folder, shown collapsed in the figure, contains the built-in operations that the ADF Model layer supports on collections like Previous, Next, First, Last, and so on.

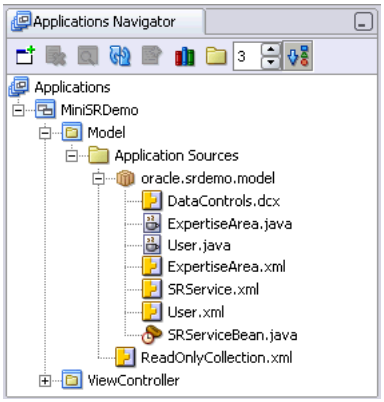
Figure 1–8 Data Control Palette Displays Services Declarative Data Binding



As you build your application, when you add additional methods on your service interface or change existing ones, simply drag and drop the `SRServiceBean` class again on to the Data Control Palette and the palette—as well as its underlying data binding metadata—will be refreshed to reflect your latest changes. The data control configuration information resides in an XML file named `DataControls.dcx` that JDeveloper adds to your `Model` project when you create the first data control. If you create multiple data controls, the information about the kind of data control they are (for example EJB, JavaBean, XML, web service, and so on.) and how to construct them at runtime lives in this file. In addition, JDeveloper creates an XML structure definition file for each data type involved in the service interface in a file whose name matches the name of that data type. For an EJB service interface, this means one structure definition file for the service class itself, and one for each JavaBean that appears as a method return value or method argument in the service interface.

These structure definition files drive the Data Control Palette display and come into play when you leverage the declarative, model-layer features like validators, prompts, tooltips, format masks, and declarative security. Since you defined these features at the model layer in these structure definition files, all your pages that present information related to these types display and validate the information in a consistent way. Figure 1–9 shows all of these files in the `Model` project in the Application Navigator after the data control for `SRServiceBean` has been created.

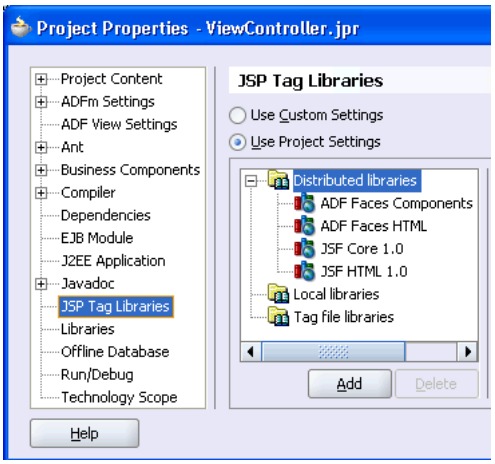
Figure 1–9 Service Classes and Data Control Metadata Files in Model Project



1.2.1.4 Dragging and Dropping Data to Create a New JSF Page

With the data control created, you can begin doing drag-and-drop data binding to create your page. Since you'll be using ADF Faces components in your page, you first ensure that the project's tag libraries are configured to use them. Double-clicking the ViewController project in the Application Navigator brings up the Project Properties dialog where you can see what libraries are configured on the JSP Tag Libraries page. If the ADF Faces Components and ADF Faces HTML libraries are missing, you can add them from here. Figure 1–10 shows the Project Properties dialog with the correction libraries for the ViewController project.

Figure 1–10 Configuring ViewController Project Tag Libraries to Use ADF Faces



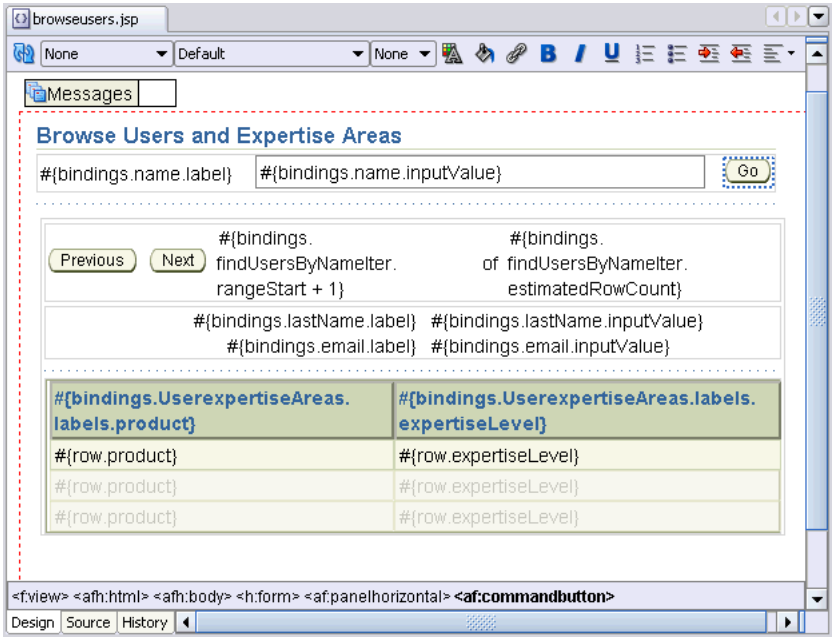
Next, you use the Create JSF JSP wizard to create a page called `browseusers.jspx`. You may be more familiar working with JSP pages that have a `*.jsp` extension, but using a standard XML-based JSP "Document" instead is a best practice for JSF development since it:

- Simplifies treating your page as a well-formed tree of UI component tags
- Discourages you from mixing Java code and component tags
- Allows you to easily parse the page to create documentation or audit reports

When the Create JSF JSP wizard completes, JDeveloper opens the new page in the visual editor. From there, creating the databound page shown in Figure 1–11 is a completely drag-and-drop experience. As you drop elements from the Data Control

Palette onto the page, a popup menu appears to show the sensible options for UI elements you can create for that element.

Figure 1–11 Browse Users JSF Page in the Visual Designer



The basic steps to create this page are:

1. Drop a **panelForm** component from the ADF Faces Core page of the Component Palette onto the page and set its text attribute in the Property Inspector to "Browse Users and Expertise Areas".
2. Drop the **findUsersByName()** method from the Data Control Palette to create an *ADF parameter form*. This operation creates a `panelForm` component containing the label, field, and button to collect the value of the `name` parameter for passing to the method when the button is clicked.
3. Drop the User return value of the **findUsersByName()** node from the Data Control Palette to create an *ADF read-only form*. This operation creates a `panelForm` component containing the label and fields for the properties of the User bean.
4. Expand the **Operations** folder child of the **User** return value in the Data Control Palette and drop the built-in **Previous** operation to the page as a command button. Repeat to drop a **Next** button to the right of it.
5. Drop the **expertiseAreas** property nested inside the **User** return value in the Data Control Palette as an *ADF read-only table*. Select **Enable Sorting** in the Edit Table Columns dialog that appears to enable sorting the data by clicking on the column headers.

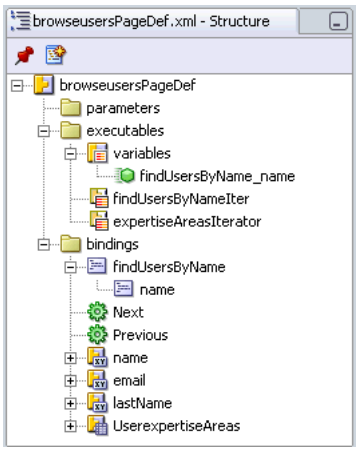
At any time you can run or debug your page to try out the user interface that you've built.

1.2.1.5 Examining the Binding Metadata Files Involved

The first time you drop a databound component from the Data Control Palette on a page, JDeveloper will create the page definition file for it. Figure 1–12 shows the contents of the `browseusersPageDef.xml` file in the Structure window. You can see that an action binding named `findUsersByName` will be created to invoke the service

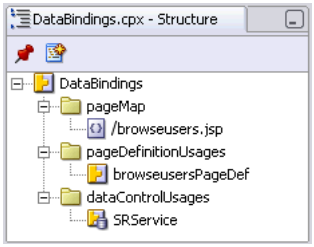
method of the same name. Iterator bindings named `findUsersByNameIter` and `expertiseAreasIterator` will be created to handle the collection of `User` beans returned from the service method and to handle the nested collection of `ExpertiseArea` beans. Action bindings named `Next` and `Previous` will be created to support the buttons that were dropped. And finally, attribute bindings of appropriate names will be created to support the read-only `outputText` fields and the table.

Figure 1–12 Page Definition XML File for browseusers.jsp



The very first time you perform Oracle ADF Model data binding in a project, JDeveloper creates one additional XML file called `DataBindings.cpx` that stores information about the mapping between page names and page definition names and lists the data controls that are in use in the project. Figure 1–13 shows what the `DataBindings.cpx` file looks like in the Structure window. At runtime, this file is used to create the overall Oracle ADF Model binding context. In addition, page map and page definition information from this file are used to instantiate the binding containers for pages as they are needed by the pages the user visits in your application.

Figure 1–13 Structure of DataBindings.cpx



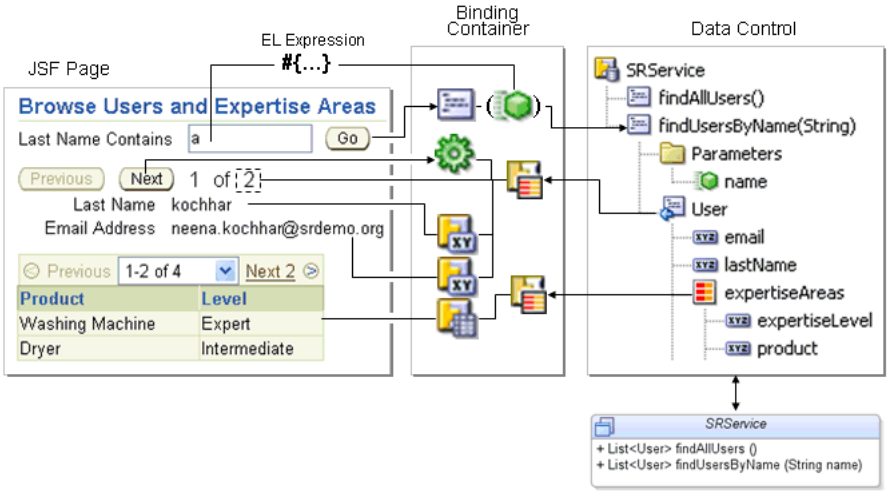
Note: For complete details on the structure and contents of the `DataControls.dcx`, `DataBindings.cpx`, and `PageDef.xml` metadata files, see Appendix A, "Reference ADF XML Files".

1.2.1.6 Understanding How Components Reference Bindings via EL

As you perform drag-and-drop data binding operations, JDeveloper creates the required ADF Model binding metadata in the page definition and creates the JSF components you've requested. Importantly it also ties the two together by configuring

various properties on the components to have EL expression values that reference the bindings. Figure 1–14 summarizes how the components on the page reference the bindings from the page's binding container at runtime.

Figure 1–14 EL Expressions Related Page Components to Bindings



As a simple example, take the Previous button. When you drop this built-in operation as a button, an action binding named `Previous` is created in the page definition file, and two properties of the `commandButton` component are set:

- `actionListener="#{bindings.Previous.execute}"`
- `disabled="#{!bindings.Previous.enabled}"`

These two EL expressions "wire" the button to invoke the built-in `Previous` operation and to automatically disable the button when the `Previous` operation does not make sense, such as when the user has navigated to the first row in the collection.

Studying another example in the page, like the read-only `outputText` field that displays the user's email, you would see that JDeveloper sets up the following properties on the component to refer to its binding:

- `value="#{bindings.email.inputValue}"`
- `label="#{bindings.email.label}"`

The combination of these two binding attribute settings tells the component to pull its value from the `email` binding, and to use the `email` binding's `label` property as a display label. Suppose you had configured custom prompts for the `User` and `ExpertiseArea` beans in the `Model` project, the bindings can then expose this information at runtime allowing the prompts to be referenced in a generic way by components on the page.

The drag-and-drop data binding steps above did not account for how the current record display (for example "*N* of *M*") appeared on the page. Since information about the current range of visible rows, the starting row in the range, and the total number of rows in the collection are useful properties available for reference on the iterator binding, to create this display, just drop three `outputText` components from the Component Palette and set each's `value` attribute to an appropriate expression. The first one needs to show the current row number in the range of results from the `findUsersByName` method, so it is necessary to set its `value` attribute to an EL expression that references the (zero-based!) `rangeStart` property on the `findUsersByNameIter` binding.

```
#{bindings.findUsersByNameIter.rangeStart + 1}
```

The second `outputText` component just needs to show the word "of", so setting its `value` property to the constant string "`of`" will suffice. The third `outputText` component needs to show the total number of rows in the collection. Here, just a reference to an attribute on the iterator binding called `estimatedRowCount` is needed.

1.2.1.7 Configuring Binding Properties If Needed

Any time you want to see or set properties on bindings in the page definition, you can select Go to Page Definition in the context menu on the page. For example, you would do this to change the number of rows displayed per page for each iterator binding by setting its `RangeSize` property. In the example shown in Figure 1–14, after visiting the page definition, the Property Inspector was used to set the `RangeSize` of the `findUsersByNameIter` iterator binding to `1` and the same property of the `expertiseAreasIterator` to `2`. Setting the `RangeSize` property for each iterator causes one user and two expertise areas to display at a time on the page.

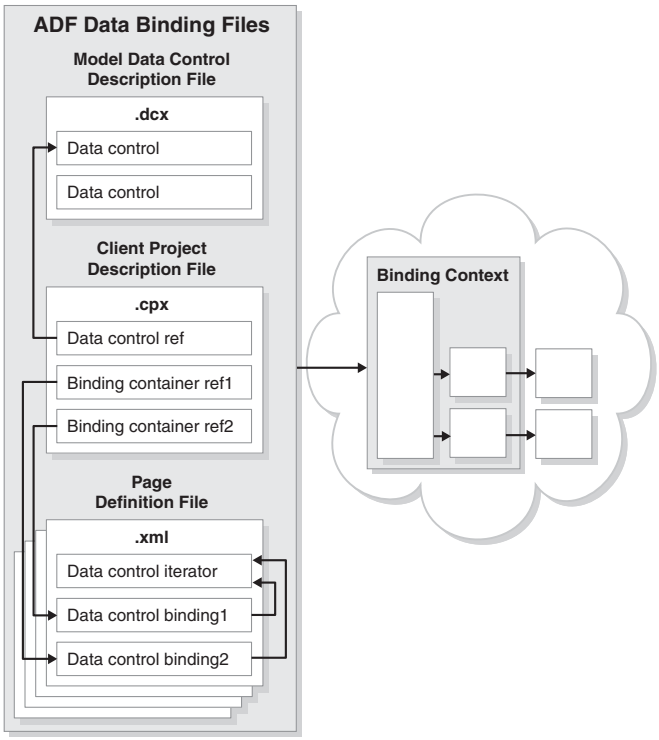
1.2.1.8 Understanding How Bindings Are Created at Runtime

The final piece of the puzzle to complete your basic understanding of Oracle ADF Model involves knowing how your data controls and declarative bindings are created at runtime based on the XML configuration files you've created. As part of configuring your project for working with Oracle ADF data binding, JDeveloper registers a servlet filter called `ADFBindingFilter` in the `web.xml` file of your `ViewController` project and maps this filter by default to URLs matching the pattern `*.jsp` and `*.jspx`.

This `ADFBindingFilter` servlet filter is responsible for finding your `DataBindings.cpx` file, based on the information in the `web.xml` file and creating the ADF binding context. The binding context is a Map that contains all binding containers, data controls, and the mapping of page names to page definition files. You can reference it at any time in your application using the EL expression `#{data}`. It's also the place where the centralized error handler is registered, and APIs are provided to set a custom error handler if needed (together with numerous other useful APIs).

When the page request is received the application invokes both the JSF lifecycle and the ADF lifecycle. Specifically, during execution of the ADF lifecycle execution, another object, the `ADFPhaseListener`, lazily instantiates the bindings in a binding container and data controls the first time they are needed to service a page request. The `ADFPhaseListener` references the information in the page map on the binding context to know which binding container to use for which page; it also references information in the `DataControls.dcx` file to know what data control factory to use. On each request, it ensures that the binding container of the current page being requested is available for reference via EL using the expression `#{bindings}`. Figure 1–15 summarizes the relationships between these metadata files.

Figure 1–15 How ADF Binding Metadata Is Used at Runtime



Once the binding container is set up for a given page, the `ADFPhaseListener` integrates the JSF page handling lifecycle with the bindings. It coordinates the per-page execution of the iterators and service methods based on information in the appropriate page definition file. The iterators and method invocation bindings are known as "executable" bindings for this reason.

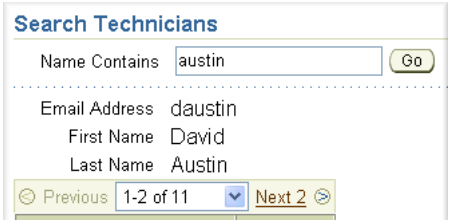
1.2.2 Making the Display More Data-Driven

After you have a basic page working, you will likely notice some aspects that you'd like to make more sophisticated. For example, you can use the properties of ADF bindings to hide or show groups of components or to toggle between alternative sets of components.

1.2.2.1 Hiding and Showing Groups of Components Based on Binding Properties

If the application user enters a last name in the `browseusers.jspx` page that matches a single user, it doesn't look very nice to show disabled Next and Previous navigation buttons and a "1 of 1" record counter. Instead, you might want a result like what you see in Figure 1–16, where these components disappear when only a single row is returned.

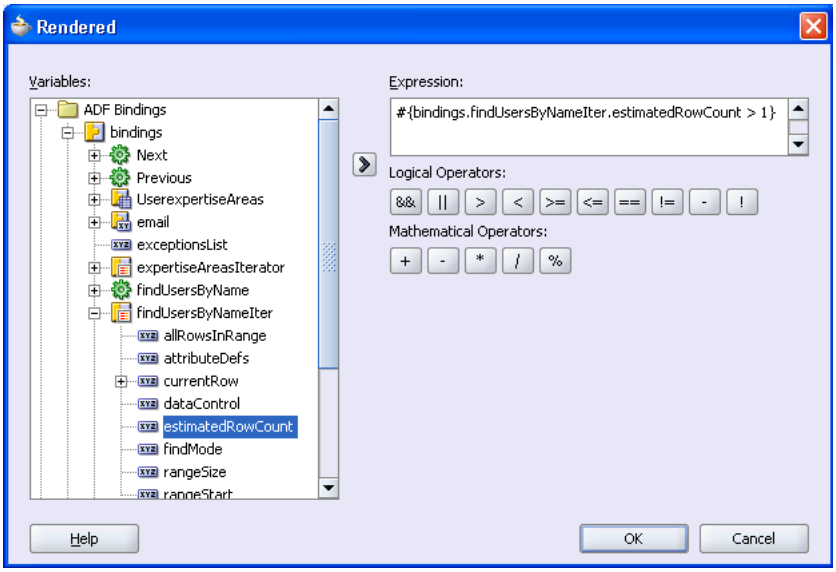
Figure 1–16 Hiding Panel with Navigation Buttons When Not Relevant



Luckily, this is easy to accomplish. You start by organizing the navigation buttons and the record counter display into a containing panel component like `panelHorizontal`. After creating the panel to contain them, you can drag and drop in the visual editor, or drag and drop in the Structure window to place the existing controls inside another container. Then, to hide or show all the components in the panel, you just need to set the value of the panel's `rendered` attribute to a data-driven EL expression.

Recall that the number of rows in an iterator binding's collection can be obtained using its `estimatedRowCount` property. Figure 1–17 shows the EL picker dialog that appears when you select the `panelHorizontal` component, click in the Property Inspector on its `rendered` attribute, and click the ... button. If you expand the bindings for the current page you will see the `findUsersByNameIter` iterator binding. You can then expand it further to see the most common properties that developers reference in EL. By picking `estimatedRowCount` and clicking the > button, you can then change the expression to a boolean expression by introducing a comparison operator to compare the row count to see if it is greater than one. When you set such an expression, the panel will be rendered at runtime only when there are two or more rows in the result.

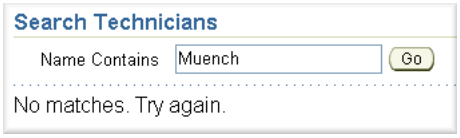
Figure 1–17 Setting a Panel's Rendered Attribute Based on Binding Properties



1.2.2.2 Toggling Between Alternative Sets of Components Based on Binding Properties

Consider another situation in the sample page. When no rows are returned, by default the read-only form would display its prompts next to empty space where the data values would normally be, and the table of experience areas would display the column headings and a blank row containing the words "No rows yet". To add a little more polish to the application, you might decide to display something different when no rows are returned in the iterator binding's result collection. For example, you might simply display a "No matches. Try again" message as shown in Figure 1–18.

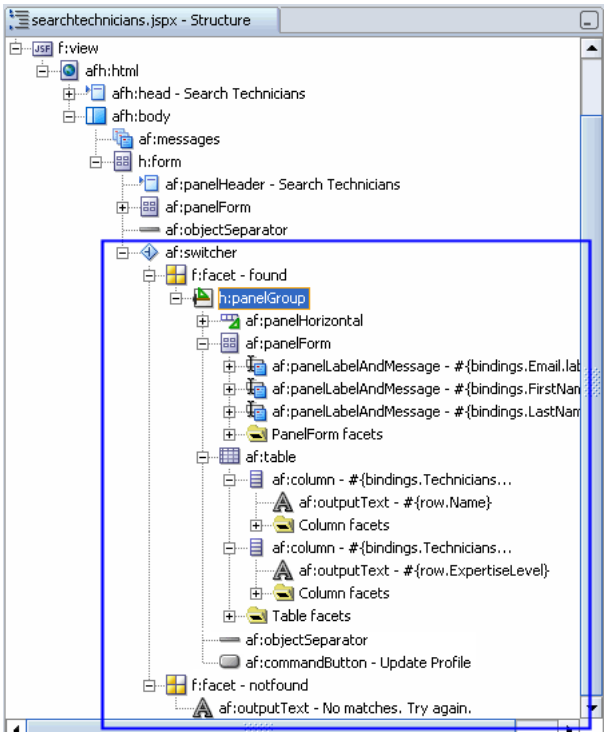
Figure 1–18 Alternative Display If Search Produces Empty Collection



JSF provides a basic feature called a "facet" that allows a UI component to contain one or more named, logical groups of other components that become rendered in a specific way at runtime. ADF Faces supplies a handy `switcher` component that can evaluate an EL expression in its `FacetName` attribute to determine which of its facets becomes rendered at runtime. Using this component effectively lets you switch between any groups of components in a dynamic and declarative way. If you group the components that present the user information and experience area table into a panel, then you can use the `switcher` component to switch between showing that panel and a simple message depending on the number of rows returned.

Figure 1–19 shows the Structure window for the `browseusers.jsp` page reflecting the hierarchical containership of JSF components after the `switcher` component is introduced. First, you would set up two JSF facets and give them meaningful names like `found` and `notfound`. Then you can organize the existing components into the appropriate facet using drag and drop in the Structure window. In the `found` facet, you want a panel containing all of the components that show the user and experience area information. In the `notfound` facet, you want just an `outputText` component that displays the "No matches. Try again" message.

Figure 1–19 Structure Window View of browseusers.jsp



By setting the `facetName` attribute of `switcher` to the EL expression, the `found` facet will be used when the row count is greater than zero, and the `notfound` facet will be used when the row count equals zero:

```
#{bindings.findUsersByNameIter.estimatedRowCount > 0
 ?'found':'notfound'}
```

The combination of Oracle ADF declarative bindings, ADF Faces components, and EL expressions demonstrates another situation that previously required tedious, repetitive coding which now can be handled with ease.

This concludes the introduction to building J2EE applications with Oracle ADF. The rest of this guide describes the details of building a real-world sample application using Oracle ADF, EJB, and JSF.

2

Oracle ADF Service Request Demo Overview

Before examining the individual web pages and their source code in depth, you may find it helpful to become familiar with the functionality of the Oracle ADF Service Request demo (SRDemo) application.

This chapter contains the following sections:

- Section 2.1, "Introduction to the Oracle ADF Service Request Demo"
- Section 2.2, "Setting Up the Oracle ADF Service Request Demo"
- Section 2.3, "Quick Tour of the Oracle ADF Service Request Demo"

2.1 Introduction to the Oracle ADF Service Request Demo

The SRDemo application is a simplified, yet complete customer relationship management sample application that lets customers of a household appliance servicing company attempt to resolve service issues over the web. The application, which consists of sixteen web pages, manages the customer-generated service request through the following flow:

1. A customer enters the service request portal on the web and logs in.
2. A manager logs in and assigns the request to a technician.

   Additionally, while logged in, managers can view and adjust the list of products that technicians are qualified to service.
3. The technician logs in and reviews their assigned requests, then supplies a solution or solicits more information from the customer.
4. The customer returns to the site to check their service request and either provides further information or confirms that the technician's solution resolved their problem.
5. The technician returns to view their open service requests and closes any confirmed by the customer as resolved.
6. While a request is open, managers can review an existing request for a technician and if necessary reassign it to another technician.

After the user logs in, they see only the application functionality that fits their role as a customer, manager, or technician.

Technically, the application design adheres to the Model-View-Controller (MVC) architectural design pattern and is implemented using these existing J2EE application frameworks:

- EJB Session Bean to encapsulate the application services of the entity classes

- JavaBean entity classes created from TopLink mappings and database tables
- JavaServer Faces navigation handler and declarative navigation rules
- Oracle ADF Faces user interface components in standard JSF web pages
- Oracle ADF Model layer components to provide data binding

As with all MVC-style web applications, the SRDemo application has the basic architecture illustrated in Chapter One, "Getting Started with Oracle ADF Applications".

This developer's guide describes in detail the implementation of each of these layers. Each chapter describes features of Oracle JDeveloper 10*g* and how these features can be used to build J2EE web applications using techniques familiar to enterprise J2EE developers.

2.1.1 Requirements for Oracle ADF Service Request Application

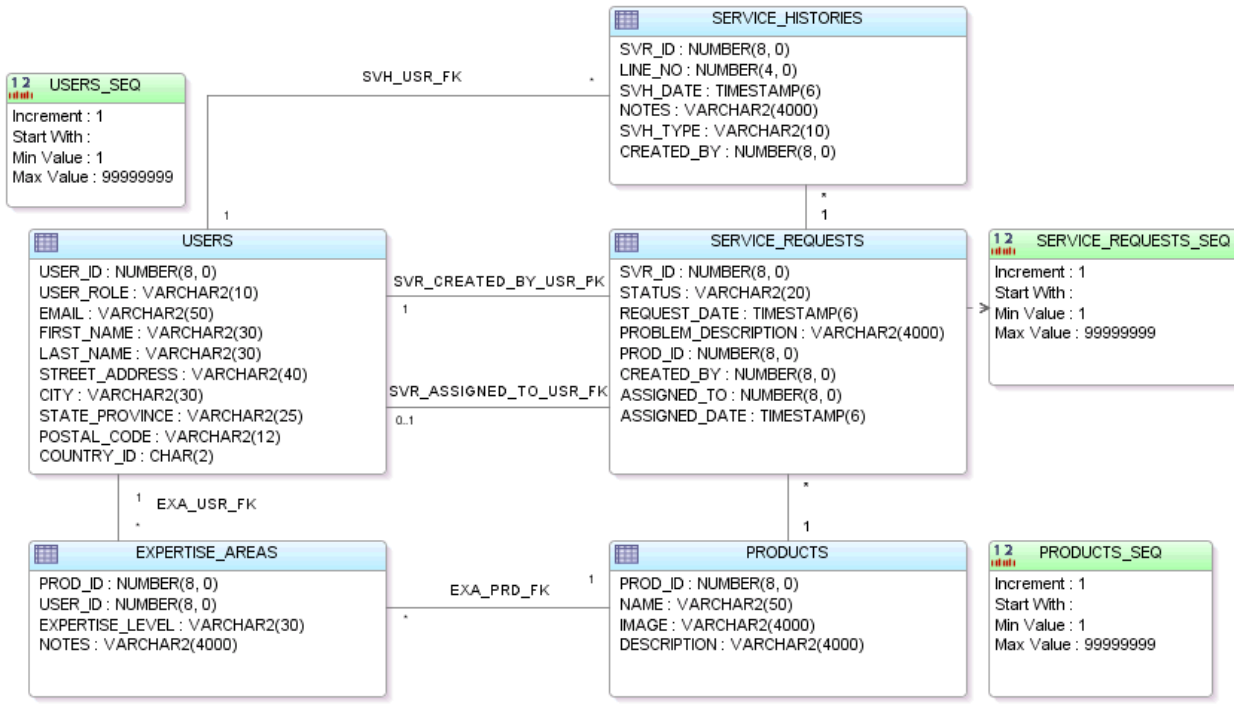
The SRDemo application has the following basic requirements:

- An Oracle database (any edition) is required for the sample schema.
- You must create a database connection named "SRDemo" to connect to the SRDemo application schema. If you install the SRDemo application using the Update Center, this connection will have been created for you (see Section 2.2.3, "Creating the Oracle JDeveloper Database Connection").
- The JUnit extension for JDeveloper must be installed. If you install the SRDemo application using the Update Center, this extension will also be installed for you (see Section 2.2.5, "Running the Oracle ADF Service Request Demo Unit Tests in JDeveloper").

2.1.2 Overview of the Schema

Figure 2–1 shows the schema for the SRDemo application.

Figure 2–1 Schema Diagram for the SRDemo Application



The schema consists of five tables and three database sequences. The tables include:

- USERS: This table stores all the users who interact with the system, including customers, technicians, and managers. The e-mail address, first and last name, street address, city, state, postal code, and country of each user is stored. A user is uniquely identified by an ID.
- SERVICE_REQUESTS: This table represents both internal and external requests for activity on a specific product. In all cases, the requests are for a solution to a problem with a product. When a service request is created, the date of the request, the name of the individual who opened it, and the related product are all recorded. A short description of the problem is also stored. After the request is assigned to a technician, the name of the technician and date of assignment are also recorded. All service requests are uniquely identified by a sequence-assigned ID.
- SERVICE_HISTORIES: For each service request, there may be many events recorded. The date the request was created, the name of the individual who created it, and specific notes about the event are all recorded. Any internal communications related to a service request are also tracked. The service request and its sequence number uniquely identify each service history.
- PRODUCTS: This table stores all of the products handled by the company. For each product, the name and description are recorded. If an image of the product is available, that too is stored. All products are uniquely identified by a sequence-assigned ID.
- EXPERTISE_AREAS: This table defines specific areas of expertise of each technician. The areas of expertise allow service requests to be assigned based on the technician's expertise.

The sequences include:

- USERS_SEQ: Populates the ID for new users.

- PRODUCTS_SEQ: Populates the ID for each product.
- SERVICE_REQUESTS_SEQ: Populates the ID for each new service request.

2.2 Setting Up the Oracle ADF Service Request Demo

These instructions assume that you are running Oracle JDeveloper 10*g*, Studio Edition, version 10.1.3.x. The application will *not* work with earlier versions of JDeveloper. To obtain JDeveloper, you may download it from the Oracle Technology Network (OTN):

`http://www.oracle.com/technology/software/products/jdev/index.html`

To complete the following instructions, you must have access to an Oracle database, and privileges to create new user accounts to set up the sample data.

2.2.1 Downloading and Installing the Oracle ADF Service Request Application

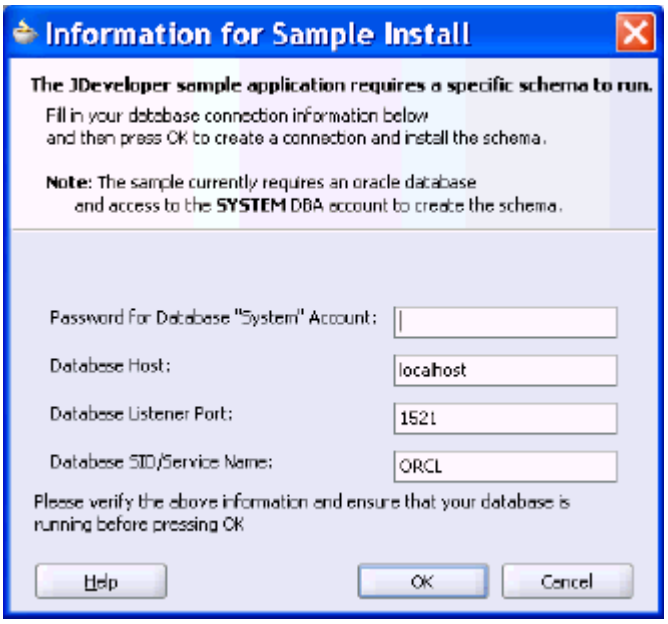
The SRDemo application is available for you to install as a JDeveloper extension. In JDeveloper, you use the Check for Updates wizard to begin the process of installing the extension.

To install the SRDemo application from the Update Center:

1. If you are using JDeveloper, save your work and close. You will be asked to restart JDeveloper to complete the update.
2. Open JDeveloper and choose **Help > Check for Updates**.
3. In the wizard, click **Next** and make sure that **Search Update Centers** and **Internal Automatic Updates** are both selected. Click **Next**.
4. Among the available updates, locate **Oracle ADF SRDemo Application** and select it. Click **Next** to initiate the download.

   Note that the Update Center may display two versions of the SRDemo Sample application. For this guide, you want to choose the one that shows "EJB 3.0 session beans and TopLink persistence" in the description.
5. When prompted, restart JDeveloper.
6. When JDeveloper restarts, select **Yes** to open the SRDemo application workspace in the Application Navigator.
7. JDeveloper displays the SRDemo Application Schema Install dialog to identify the database to use for the sample data.

Figure 2–2 SRDemo Application Schema Dialog



8. If you want to install the sample data and have access to a SYSTEM DBA account, enter the connection information in the Sample Install dialog.

   Note: The connection information you provide may be for either a local or a remote database, and that database may be installed with or without an existing SRDemo schema.

The SRDemo application will appear in the directory *<JDEV_INSTALL>*`/jdev/samples/SRDemoSample`. The Update Center also installs the extension JAR file *<JDEV_INSTALL>*`/jdev/extensions/oracle.jdeveloper.srdemo.10.1.3.jar` which allows JDeveloper to create the SRDemo application workspace.

2.2.2 Installing the Oracle ADF Service Request Schema

The SRDemo schema is defined in a series of SQL scripts stored in the `<JDEV_INSTALL>/jdev/samples/SRDemoSample/DatabaseSchema/scripts` directory. The schema will automatically be created when you install the application using the Update Center; however, for manual purposes, you can install or reinstall the schema in several ways.

> **Note:** You may skip the following procedure if you installed the SRDemo application using the Update Center in JDeveloper and proceeded with the schema installer. For details, see Section 2.2.1, "Downloading and Installing the Oracle ADF Service Request Application".

Follow these instructions to manually create the SRDemo schema.

To manually install the SRDemo schema:

- From the command line, run the `SRDemoInstall.bat/.sh` command line script from the `SRDemoSample` root directory.

or

- In JDeveloper, open the ANT build file `build.xml` in the BuildAndDeploy project of the SRDemoSample workspace and run the `setupDBOracle` task by choosing **Run Ant Target** from the task's context menu.

  or

- From SQL*Plus, run the `build.sql` script when connected as a DBA such as SYSTEM.

When you install the schema manually, using the setupDBOracle task, the following questions and answers will appear:

```
-------------------------------------------------------------
SRDemo Database Schema Install 10.1.3
(c) Copyright 2006 Oracle Corporation. All rights reserved.
-------------------------------------------------------------
This script installs the SRDemo database schema into an
Oracle database.
The script uses the following defaults:

Schema name: SRDEMO
Schema password: ORACLE
Default tablespace for SRDEMO: USERS
Temp tablespace for SRDEMO: TEMP
DBA account used to create the Schema: SYSTEM
If you wish to change these defaults update the file
BuildAndDeploy/build.properties with your values
-------------------------------------------------------------
```

What happens next depends on how the demo was installed and what kind of JDeveloper installation yours is (either FULL or BASE).

- If the SRDemo application was installed manually and is not in the expected *<JDEV_HOME>*`/jdev/samples/SRDemoSample` directory, you will be prompted for the JDeveloper home directory.
- If JDeveloper is a BASE install (one without a JDK), then you will be prompted for the location of the JDK (1.5).
- If the SRDemo application was installed using the Update Center into a FULL JDeveloper install. The task proceeds.

You will next be prompted to enter database information. Two default choices are given, or you can supply the information explicitly:

```
Information about your database:
-------------------------------------------
Select one of the following database options:
1. Default local install of Oracle Personal, Standard or Enterprise edition
   Host=localhost, Port=1521, SID=ORCL
2. Default local install of Oracle Express Edition
   Host=localhost, Port=1521, SID=XE
3. Any other non-default or remote database install
Choice [1]:
```

If you choose 1 or 2, the install proceeds to conclusion. If you choose 3, then you will need to supply the following information: (defaults shown in brackets)

```
Host Name or IP Address for your database machine [localhost]:
Database Port [1521]:
Database SID [orcl]:
```

The final question is for the DBA Account password:

```
Enter password for the SYSTEM DBA account [manager]:
```

The install continues.

2.2.3 Creating the Oracle JDeveloper Database Connection

You must create a database connection called "SRDemo" to connect to the sample data schema. If you installed the SRDemo application using the Update Center, this connection will have been created for you.

> **Note:** You may skip the following procedure if you installed the SRDemo application using the Update Center in JDeveloper. In that case, the database connection will automatically be created when you download the application.

Follow these instructions to manually create a new database connection to the Service Request schema.

To manually create a database connection for the SRDemo application:

1. In JDeveloper, choose **View > Connections Navigator**.
2. Right-click the **Database** node and choose **New Database Connection** from the context menu.
3. Click **Next** on the Welcome page.
4. In the **Connection Name** field, type the connection name `SRDemo`. Then click **Next**.

   Note: The name of the connection (`SRDemo`) is case sensitive and must be typed exactly to match the SRDemo application's expected connection name.
5. On the Authentication page, enter the following values. Then click **Next**.

   Username: SRDEMO

   Password: Oracle

   Deploy Password: Select the checkbox.
6. On the Connection page, enter the following values. Then click **Next**.

   Host Name: localhost

   JDBC Port: 1521

   SID: ORCL (or XE)

   Note: If you are using Oracle 10*g* Express Edition, then the default SID is "XE" instead of "ORCL".
7. Click **Test Connection**. If the database is available and the connection details are correct, then continue. If not, click the **Back** button and check the values.
8. Click **Finish**. The connection now appears below the **Database Connection** node in the Connections Navigator.

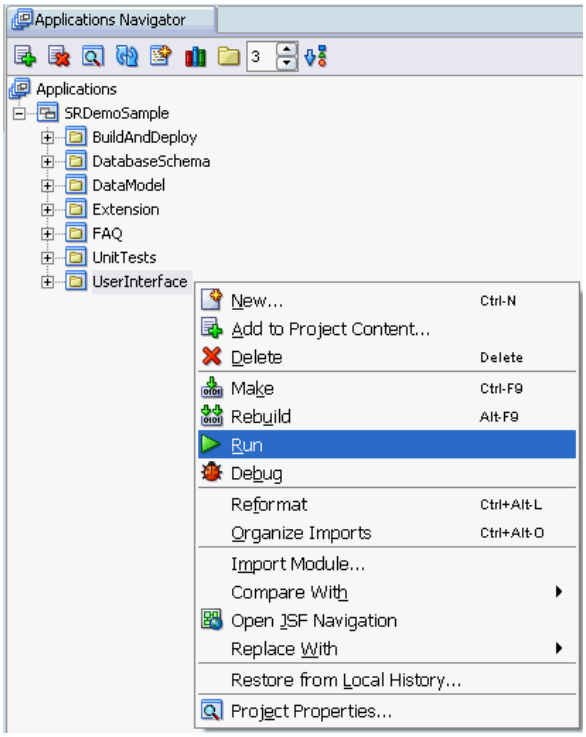
You can now examine the schema from JDeveloper. In the Connections Navigator, expand **Database > SRDemo**. Browse the database elements for the schema and confirm that they match the schema definition described in Section 2.1.2, "Overview of the Schema".

2.2.4 Running the Oracle ADF Service Request Demo in JDeveloper

If you installed the SRDemo application using the Update Center, choose **Help > Open SRDemo Application Workspace** to open the application workspace.

- Run the application in JDeveloper by selecting the **UserInterface** project in the Application Navigator and choosing **Run** from the context menu, as shown in Figure 2–3.

Figure 2–3 Running the SRDemo Application in JDeveloper



Tip: The **UserInterface** project defines `index.jspx` to be the default run target. This information appears in the Runner page of the Project Properties dialog for the **UserInterface** project. This allows you to simply click the **Run** icon in the JDeveloper toolbar when this project is active, or right-click the project and choose **Run**. To see the project's properties, select the project in the navigator, right-click, and choose **Property Properties**.

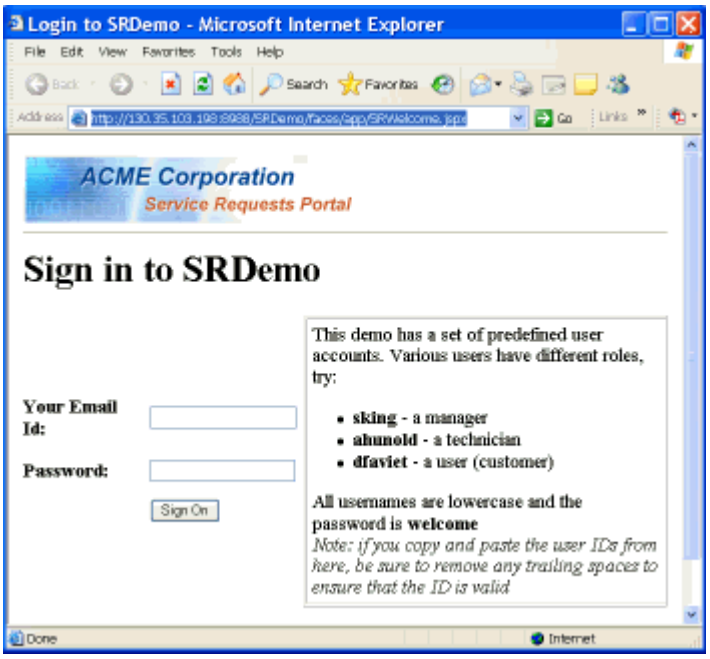
Running the `index.jspx` page from inside JDeveloper will start the embedded Oracle Application Server 10*g* Oracle Containers for J2EE (OC4J) server, launch your default browser, and cause it to request the following URL:

```
http://130.35.103.198:8988/SRDemo/faces/app/SRWelcome.jspx
```

If everything is working correctly, the `index.jspx` page's simple scriptlet `response.sendRedirect("faces/app/SRWelcome.jspx")`, will redirect to display the login page of the SRDemo application, as shown in Figure 2–4.

Tip: If your machine uses DHCP to get an automatically-assigned IP address, then after JDeveloper launches your default browser and starts embedded OC4J you may see an HTTP error stating that the web page does not exist. To correct this issue, you can specify the host name, localhost. Choose **Embedded OC4J Preferences** from the **Tools** menu and on the **Startup** tab set the **Host Name or IP Address Used to Refer to the Embedded OC4J** preference to use the **Specify Host Name** option, and enter the value `localhost`. Then, edit the URL above to use `localhost` instead of `130.35.103.198`.

Figure 2–4 SRWelcome.jspx: SRDemo Login Page



See Section 2.3, "Quick Tour of the Oracle ADF Service Request Demo" to become familiar with the web pages that are the subject of this developer's guide. Additionally, read the tour to learn about ADF functionality used in the SRDemo application and to find links to the implementation details documented in this guide.

2.2.5 Running the Oracle ADF Service Request Demo Unit Tests in JDeveloper

JUnit is a popular framework for building regression tests for Java applications. Oracle JDeveloper 10*g* features native support for creating and running JUnit tests, but this feature is installed as a separately downloadable JDeveloper extension. You can tell if you already have the JUnit extension installed by choosing **File > New** from the JDeveloper main menu and verifying that you have a **Unit Tests (JUnit)** category under the **General** top-level category in the New Gallery.

If you do *not* already have the JUnit extension installed, then use the Update Center in JDeveloper to install it.

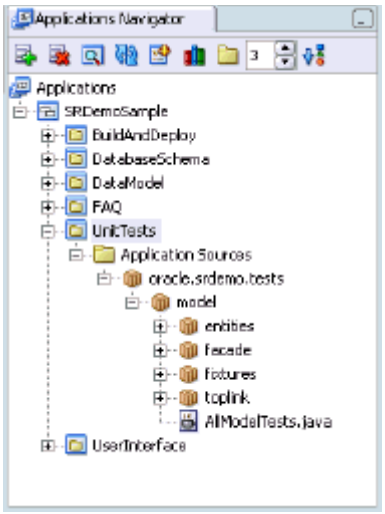
Note: You may skip the following procedure if you installed the SRDemo application using the Update Center in JDeveloper. In that case, the JUnit extension will automatically be installed when you download the application.

To install the JUnit extension from the Update Center:

1. If you are using JDeveloper, save your work and close. You will be asked to restart JDeveloper to complete the update.
2. Open JDeveloper and choose **Help > Check** for Updates.
3. In the wizard, click **Next** and make sure that **Search Update Centers** and **Internal Automatic Updates** are both selected. Click **Next**.
4. Among the available updates, locate **JUnit Integration 10.1.3.xx** and select it. Click **Next** to initiate the download.
5. When prompted, restart JDeveloper.
6. When JDeveloper restarts, the new extension will be visible in the **Unit Tests** category in the New Gallery.

The `UnitTests` project in the SRDemo application workspace contains a suite of JUnit tests that are configured in the `AllModelTests.java` class shown in Figure 2–5. To run the regression test suite, select the `AllModelTests.java` class in the Application Navigator and choose **Run** from the context menu. Since this class is configured as the default run target of the `UnitTests` project, alternatively you can select the project itself in the Application Navigator and choose **Run** from its context menu.

Figure 2–5 Running the SRDemo Unit Tests in JDeveloper



JDeveloper opens the JUnit Test Runner window to show you the results of running all the unit tests in the suite. Each test appears in a tree display at the left, grouped into test cases. Green checkmark icons appear next to each test in the suite that has executed successfully, and a progress indicator gives you visual feedback on the percentage of your tests that are passing.

> **Tip:** You can find more details on JUnit on the web at `http://www.junit.org/index.htm`.

2.3 Quick Tour of the Oracle ADF Service Request Demo

The SRDemo application is a realistic web portal application that allows customers to obtain appliance servicing information from qualified technicians. After the customer opens a new service request, a service manager assigns the request to a technician with suitable expertise. The technician sees the open request and updates the customer's service request with information that may help the customer solve their problem.

The application recognizes three user roles (customer, manager, and technician). As the following sections show, the application features available to the user depend on the user's role.

> **Note:** The remainder of this chapter provides an overview of the web pages you will see when you run the SRDemo application. You can quickly find implementation details in this guide from the list at the end of each section. For an overview of the underlying business logic, read Chapter Three.

2.3.1 Customer Logs In and Reviews Existing Service Requests

Enter the log in information for a customer:

- **User name**: `dfaviet`
- **Password**: `welcome`

Click the **Sign On** button to proceed to the web portal home page.

To enter the web portal click the **Start** button.

This action displays the customer's list of open service requests, as shown in Figure 2–6.

Figure 2–6 SRList.jspx: List Page for a Customer

When you log in as the customer, the list page displays a menu with only two tabs, with the subtabs for **My Service Requests** selected.

Note that additional tabs will be visible when you log in as the manager or technician.

Select the menu subtab **All Requests** to display both closed and open requests.

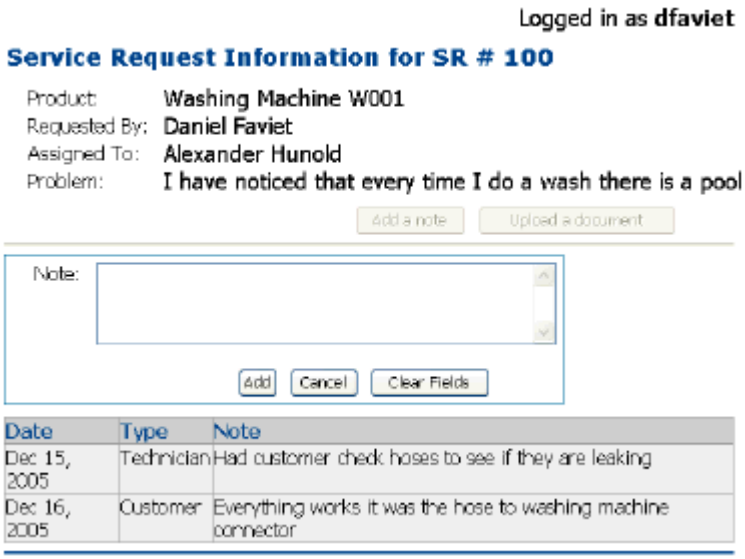
To browse the description of any request, select the radio button corresponding to the row of the desired request and click **View**.

The same operation can also be performed by clicking on the service request link in **Request Id** column.

The customer uses the resulting page to update the service request with their response. To append a note to the current service request, click **Add a note**.

Figure 2–7 shows an open service request selected by a customer and the note they are about to append. Notice that the buttons above the text input field appear disabled to prevent the user from selecting those operations until the task is completed. Below the note field, is the list of previous notes for this master service request.

Figure 2–7 SRMain.jspx: Main Page for a Customer



Where to Find Implementation Details

The Oracle ADF Developers Guide describes the following major features of this section.

- Using dynamic navigation menus: The menu tabs and subtabs which let the user access the desired pages of the application, are created declaratively by binding each menu component to a menu model object and using the menu model to

display the appropriate menu items. See Section 11.2, "Using Dynamic Menus for Navigation".

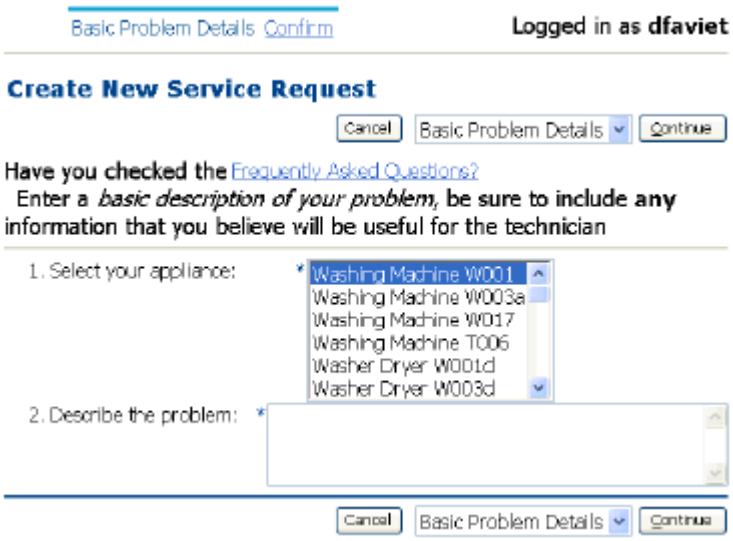
- Displaying data items in tables: The list of service requests is formatted by a UI table component bound to a collection. The Data Control Palette lets you easily drop databound components into your page. See Section 7.2, "Creating a Basic Table".
- Displaying a page by passing parameter values: The user may select a service request from the list of open requests and edit the details in the edit page. The `commandLink` is used to both navigate to the detail page and to send the parameters that the form creation method uses to display the detail page data. See Section 10.4, "Setting Parameter Values Using a Command Component".
- Using a method with parameters to create a form: The user drills down to a browse form that gets created using a finder method from the service. Instead of the method returning all service requests, it displays only the specific service request passed by the previous page. See Section 10.6, "Creating a Form or Table Using a Method that Takes Parameters".
- Displaying master-detail information: The user can browse the service history for a single service request in one form. The enter form can be created using the Data Control Palette. See Section 8.3, "Using Tables and Forms to Display Master-Detail Objects".

2.3.2 Customer Creates a Service Request

To create a new service request, select the **New Service Request** tab.

This action displays the first page of a two-step process train for creating the service request. Figure 2–8 shows the first page.

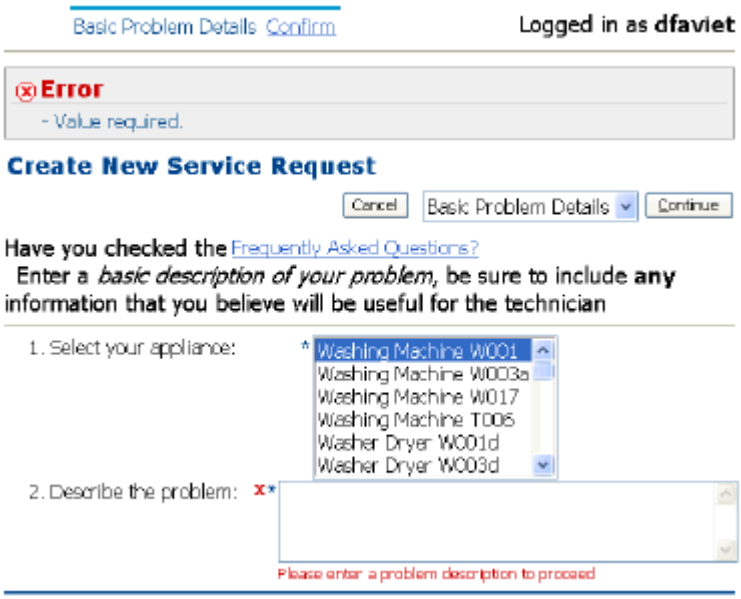
Figure 2–8 SRCreate.jspx: Step One, Create-Service-Request Page



The input fields that your page displays can raise validation errors when the user fails to supply required information. To see how the application handles a validation error, by clicking the **Logout** menu item before entering a problem description for the new service request.

Figure 2–9 shows the validation error that occurs in the create-service-request page when the problem description is not entered. The error message that displays below the problem description field directs the user to enter a description.

Figure 2–9 SRCreate.jspx: Step One Validation Error in Page

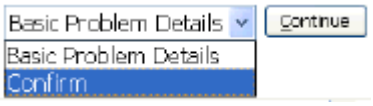


You can see another way of handling validation errors by clicking the **Continue** button before entering a problem description.

Figure 2–10 shows the validation error that displays within a dialog when the problem description is not entered.

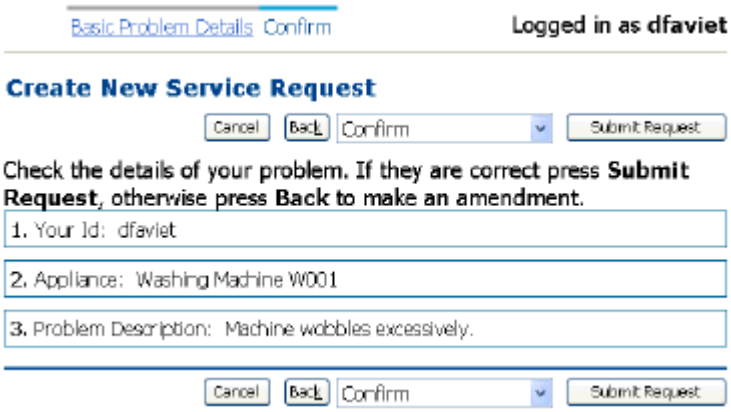
Figure 2–10 SRCreate.jspx: Step One Validation Error in Separate Dialog

To proceed to the next page of the process train, first type some text into the problem description field, then either choose **Confirm** from the dropdown menu or click the **Continue** button.

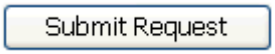


In the next step, the customer confirms that the information is correct before submitting the request. Figure 2–11 shows the final page. Notice that the progress bar at the top of the page identifies **Confirm** is the last step in this two-page create-service-request process chain.

Figure 2–11 SRCreateConfirm.jspx: Step Two, Create-Service-Request Page



Click the **Submit Request** button to enter the new service request into the database. A confirmation page displays after the new entry is created, showing the service request ID assigned to the newly created request.



To continue the application as the manager role, click the **Logout** menu item to return to the login page.

Where to Find Implementation Details

The Oracle ADF Developers Guide describes the following major features of this section.

- Creating a new record: The user creates a new service request using a form that commits the data to the data source. JDeveloper lets you create default constructor methods on the service as an easy way to drop record creation forms. Alternatively, custom methods on the service may be used. See Section 10.7, "Creating an Input Form for a New Record".
- Multipage process: The ADF Faces components `processTrain` and `processChoiceBar` guide the user through the process of creating a new service request. See Section 11.5, "Creating a Multipage Process".
- Showing validation errors in the page: There are several ways to handle data-entry validation in an ADF application. You can take advantage of validation rules provided by the ADF Model layer. See Section 12.3, "Adding Validation".

- Handling page navigation using a command button: The application displays the appropriate page when the user chooses the **Cancel** or **Submit** button. Navigation rules, with defined outcomes, determine which pages is displayed after the button click. See Section 9.1, "Introduction to Page Navigation".

2.3.3 Manager Logs In and Assigns a Service Request

Enter the log in information for a manager:

- **User name**: `sking`
- **Password**: `welcome`

Click the **Sign On** button to proceed to the web portal home page.

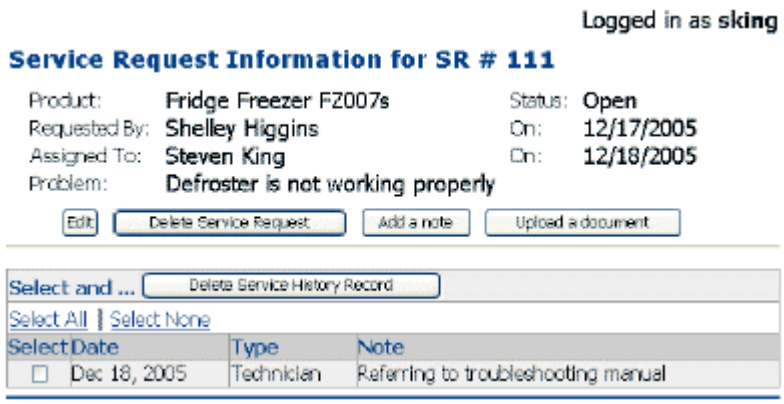
Click the **Start** button.

This action displays the manager's list of open service requests. The list page displays four menu tabs, with the subtabs for the **My Service Requests** tab selected.

To see a description of any request, select a radio button corresponding to the row of the desired request and click **View**.

Figure 2–12 shows an open service request. Notice that when logged in as the manager, the page displays an **Edit** button and a **Delete Service History Record** button. These two operations are role-based and only available to the manager.

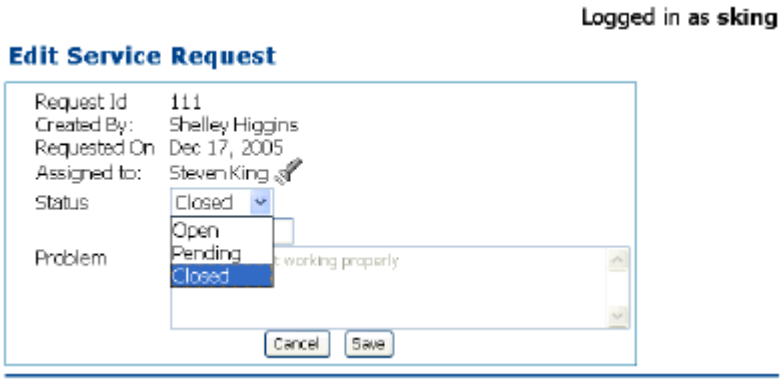
Figure 2–12 SRMain.jspx: Main Page for a Manager



To edit the current service request, click **Edit**.

Figure 2–13 shows the detail edit page for a service request. Unlike the page displayed for the technician, the manager can change the status and the assigned technician.

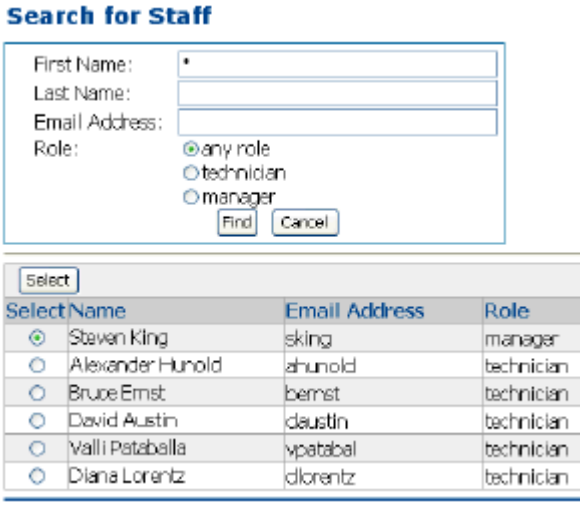
Figure 2–13 SREdit.jspx: Edit Page for a Manager



To find another technician to assign, click the symbol next to the assigned person's name.

Figure 2–14 shows the query by criteria search page that allows the manager to search for staff members (managers and technicians). This type of search allows wild card characters, such as the % and * symbols.

Figure 2–14 SRStaffSearch.jspx: Staff Search Page for a Manager



To assign another staff member to this service request, click the selection button next to the desired staff's name.

To update the open service request with the selected staff member, click the **Select** button.

Where to Find Implementation Details

The Oracle ADF Developers Guide describes the following major features of this section.

- Databound dropdown lists: The ADF Faces component `selectOneChoice` allows the user to change the status of the service request or to pick the type of service request to perform a search on. See Section 11.7, "Creating Databound Dropdown Lists".
- Searching for a record: The user can search existing service requests using a query-by-example search form. In this type of query, the user enters criteria info a form based on known attributes of an object. Wild card search is supported. See Section 10.8, "Creating Search Pages".
- Using a popup dialog: At times you may prefer to display information in a separate dialog that lets the user postback information to the page. The search window uses a popup dialog rather than display the search function in the page. See Section 12.7, "Displaying Error Messages" and Section 11.3, "Using Popup Dialogs".
- Using Partial Page Rendering: When the user clicks the flashlight icon (which is a `commandLink` component with an `objectImage` component), a popup dialog displays to allow the user to search and select a name. After selecting a name, the popup dialog closes and the **Assigned to** display-only fields are refreshed with the selected name; other parts of the edit page are not refreshed. See Section 11.4, "Enabling Partial Page Rendering".
- Using managed bean to store information: Pages often require information from other pages in order to display correct information. Instead of setting this information directly on a page, which essentially hardcodes the information, you can store this information on a managed bean. For example, the managed bean allows the application to save the page which displays the SREdit page and to use the information in order to determine where to navigate for the Cancel action. See Section 10.2, "Using a Managed Bean to Store Information".
- Passing parameters between pages: The `commandLink` component is used to both navigate to the SREdit page and to set the needed parameter for the `findServiceRequestById(Integer)` method used to create the form that displays the data on the SREdit page. You can use the ADF Faces `setActionListener` component to set parameters. See Section 10.4, "Setting Parameter Values Using a Command Component".

2.3.4 Manager Views Reports and Updates Technician Skills

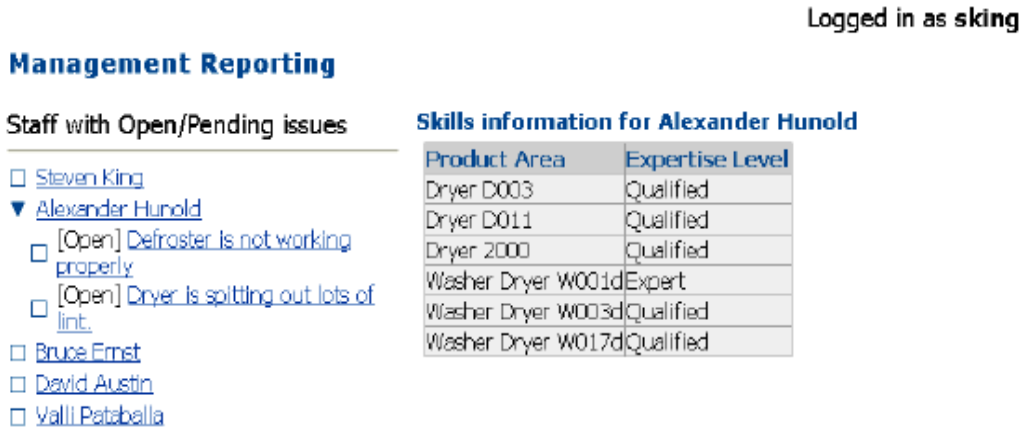
To access the manager-only page, select the **Management** tab.

This action displays the staff members and their service requests in a master-detail ADF Faces tree table component. Figure 2–15 shows the tree table with an expanded technician node.

Figure 2–15 SRManage.jspx: Management Reporting Page

Each child node in the tree table is linked to a detail service request report. Click the child node link `Defroster is not working properly` to display the detail:

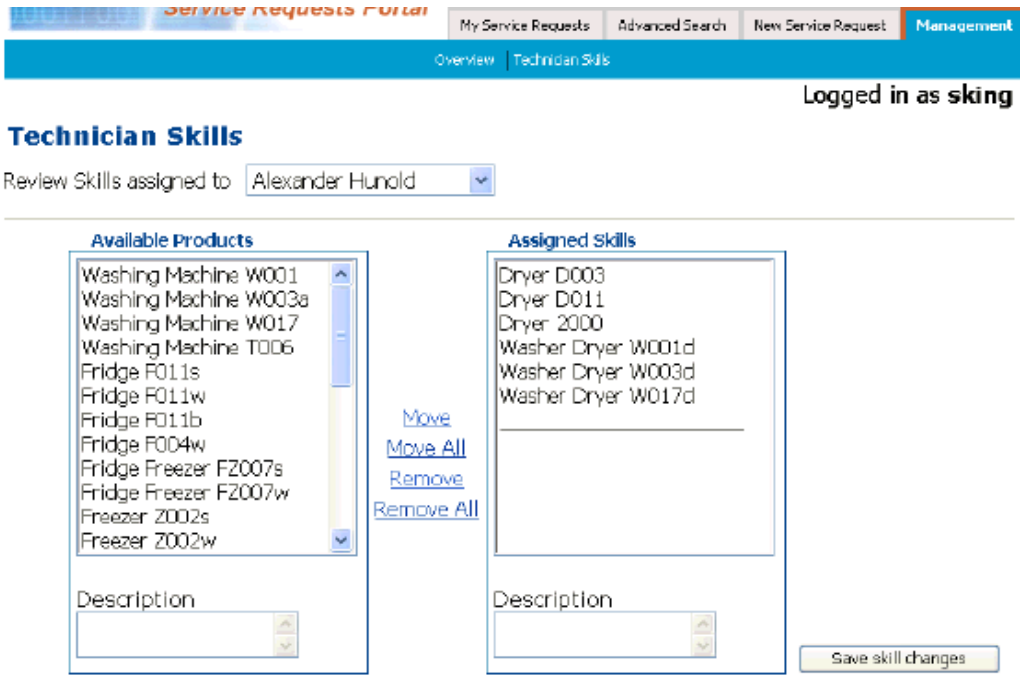
Each staff name is linked to a detail of the staff member's assigned skills. Click the staff name link `Alexander Hunold` to display the list of assigned skills:



To access the skills assignment page, select the **Technician Skills** subtab.

This action displays a staff selection dropdown list and an ADF Faces shuttle component. Figure 2–16 shows the shuttle component populated with the skills of the selected staff member.

Figure 2–16 SRSkills.jspx: Technician Skills Assignment Page



Use the supplied `Move`, `Move All`, `Remove`, or `Remove All` links to shuttle items between the two lists. The manager can make multiple changes to the **Assigned Skills** list before committing the changes. No changes to the list are committed until the **Save skill changes** button is clicked.

To continue the application as the technician role, click the **Logout** menu item to return to the login page.

Where to Find Implementation Details

The Oracle ADF Developers Guide describes the following major features of this section.

- Creating a shuttle control: The ADF Faces component `selectManyShuttle` lets managers assign product skills to a technician. The component renders two list boxes, and buttons that allow the user to select multiple items from the leading (or "available") list box and move or shuttle the items over to the trailing (or "selected") list box, and vice versa. See Section 11.8, "Creating a Databound Shuttle".
- Role-based authorization: You can set authorization policies against resources and users. For example, you can allow only certain groups of users the ability to view, create or change certain data or invoke certain methods. Or you can prevent components from rendering based on the group a user belongs to. See Section 18.7, "Implementing Authorization Programmatically".

2.3.5 Technician Logs In and Updates a Service Request

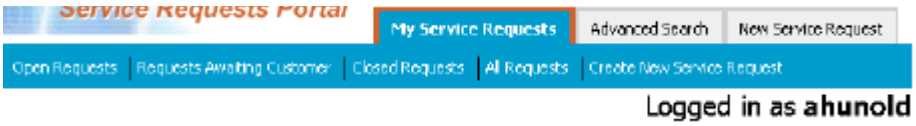
Enter the log in information for a technician:

- **User name**: `ahunold`
- **Password**: `welcome`

Click the **Sign On** button to proceed to the web portal home page.

Click the **Start** button.

This action displays the technician's list of open service requests. The list page displays two tabs, with the subtabs for the **My Service Requests** tab selected.



To open a request, select a radio button corresponding to the row with the desired request and click **View**.

The technician uses the displayed page to update the service request with their response. To attach a document to the current service request, click **Upload a document**.

Figure 2–17 shows the file upload window. (Please note the SRDemo application currently provides no means to view the contents of the uploaded document.)

Figure 2–17 SRFileUpload.jspx: File Upload Page Displayed for a Technician

Where to Find Implementation Details

File uploading: Standard J2EE technologies such as Servlets and JSP, and JSF 1.1.x, do not directly support file uploading. The ADF Faces framework, however, has integrated file uploading support at the component level via the `inputFile` component. See Section 11.6, "Providing File Upload Capability".

Changing application look and feel: Skins allow you to globally change the appearance of ADF Faces components within an application. A skin is a global style sheet that only needs to be set in one place for the entire application. Instead of having to style each component, or having to insert a style sheet on each page, you can create one skin for the entire application. See Section 14.3, "Using Skins to Change the Look and Feel".

Automatic locale-specific UI translation: ADF Faces components provide automatic translation. The resource bundle used for the components' skin (which determines look and feel, as well as the text within the component) is translated into 28 languages. For example, if a user sets the browser to use the German language, any text contained within the components will automatically display in German. See Section 14.4, "Internationalizing Your Application".

3

Building and Using Application Services

This chapter describes how to build and use application services in JDeveloper

This chapter includes the following sections:

- Section 3.1, "Introduction to Business Services"
- Section 3.2, "Implementing Services with EJB Session Beans"
- Section 3.3, "Creating Classes to Map to Database Tables"
- Section 3.4, "Mapping Classes to Tables"
- Section 3.5, "Mapping Related Classes with Relationships"
- Section 3.6, "Finding Objects by Primary Key"
- Section 3.7, "Querying Objects"
- Section 3.8, "Creating and Modifying Objects with a Unit of Work"
- Section 3.9, "Interacting with Stored Procedures"
- Section 3.10, "Exposing Services with ADF Data Controls"

3.1 Introduction to Business Services

Oracle recommends developing the model portion of an application using TopLink to persist POJO (plain old Java objects) for your business services, EJB session beans to implement a session facade, and how to expose the functionality through a data control. Oracle JDeveloper includes several wizards to quickly and easily create your model project.

Refer to Chapter 19, "Advanced TopLink Topics" for additional information on using TopLink ADF.

For detailed information on Oracle TopLink, refer to the complete *Oracle TopLink Developer's Guide* and Oracle TopLink Javadoc.

> **Tip:** Most teams have their own respective source control management (SCM) procedures, policies, and common philosophies towards what constitutes a transaction or unit of work for the SCM system. In the absence of a policy, you should group logical changes into a transaction, and also commit your changes when you need to share your modifications with another member of your team. In general, it is not advisable to commit changes when they do not compile cleanly or pass the unit test created for them.

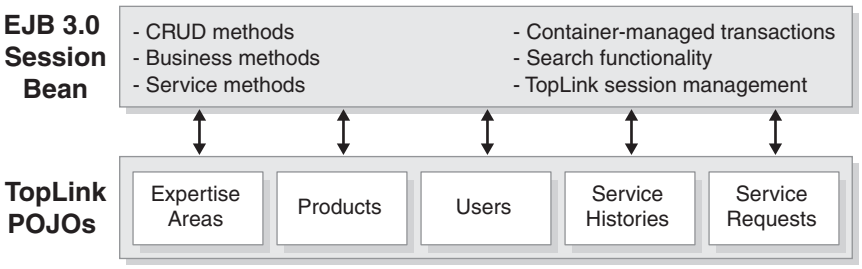
3.2 Implementing Services with EJB Session Beans

A session bean exposes the functionality of the business layer to the client.

> **Note:** While you can expose methods on a TopLink entity directly as a business service, this is not the best practice for a Web application. This model will work for basic CRUD functionality, but even simple operations that include interactions between business layer objects require custom code that becomes difficult and unwieldy.

The most common use of a session bean is to implement the session facade J2EE design pattern. A session facade is a session bean that aggregates data and presents it to the application through the model layer. Session facades have methods that access entities as well as methods that expose services to clients. Session beans have a transactional context via the container, so they automatically support basic CRUD functionality.

Figure 3–1 Session Facade Functionality



3.2.1 How to Create a Session Bean

To create a session bean, use the Create Session Bean wizard. This wizard is available from the New Gallery, in the Business Tier category.

The Create Session Bean wizard offers several options, such as EJB version, stateful and stateless sessions, remote and/or local interfaces, container- or bean-managed transactions (CMT or BMT), and choosing how to implement session facade methods. When you create a session bean for a TopLink project, you must choose an EJB 3.0 version session bean and a stateless session. You should also choose container-managed transactions (CMT), as bean-managed transactions (BMT) are beyond the scope of this book. The other options in the Create Session Bean wizard are discussed below.

3.2.1.1 Remote and Local Interfaces

The type of interface required depends on the client. If the client is running in the same virtual machine (VM), a local interface is usually the best choice. If the client runs on a separate VM, a remote interface is required. Most Web applications (JSF/JSP/Servlet) have the client and service running in the same VM, so a local interface is the best practice. Java clients (ADF Swing) run in a separate VM and require a remote interface.

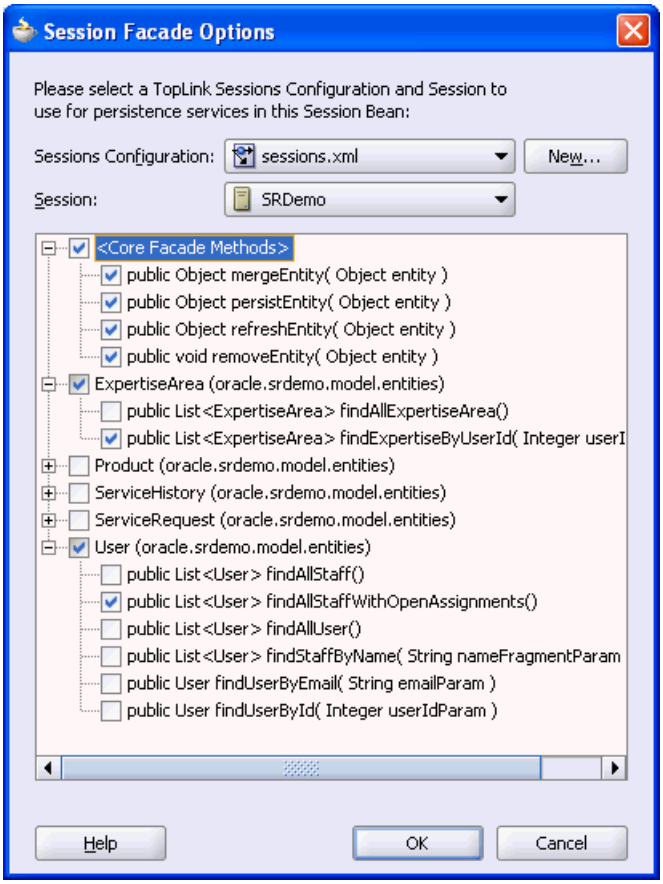
3.2.1.2 Generating Session Facade Methods

A session facade contains core CRUD methods for transactions as well as methods to access entities. To generate session facade methods, select the checkbox for Generate Session Facade Methods in the Create Session Bean wizard, and use the following

page to specify which methods to generate. JDeveloper automatically detects all the entities in the project and allows you to choose which entities and methods you want to create session facade methods for.

You can generate session facade methods for every entity in the same project, which can be useful for testing purposes, but is often too much clutter in a single session bean. Session beans are often tailored to a specific task, and contain no more information than is required for that task. Use the tree control to explicitly choose which methods to generate.

Figure 3–2 Selecting Session Facade Methods



3.2.2 What Happens When You Create a Session Bean

The session bean class contains session-wide fields and service methods. When you create a session bean, JDeveloper generates the bean class and a separate file for the local and/or remote interfaces. The remote interface is the name of the session bean, for example, SRAdminFacade.java, while the bean class is appended with Bean.java and the local interface is appended with Local.java. You should not need to modify the interface files directly, so they are not visible in the Application Navigator. To view the interface files, use the System Navigator or the Structure Pane.

Example 3–1 SRAdminFacade.java Interface

```
package oracle.srdemo.model;
import java.util.List;
import javax.ejb.Local;
import oracle.srdemo.model.entities.ExpertiseArea;
```

```
import oracle.srdemo.model.entities.Product;
import oracle.srdemo.model.entities.User;
import oracle.toplink.sessions.Session;

@Local
public interface SRAdminFacade {
  Object mergeEntity(Object entity);
  Object persistEntity(Object entity);
  Object refreshEntity(Object entity);

  void removeEntity(Object entity);

  List<ExpertiseArea> findExpertiseByUserId(Integer userIdParam);

  ExpertiseArea createExpertiseArea(Product product, User user, Integer prodId,
                                    Integer userId, String expertiseLevel,
                                    String notes);

  Product createProduct(Integer prodId, String name, String image,
                        String description);

  List<User> findAllStaffWithOpenAssignments();

  User createUser(Integer userId, String userRole, String email,
                  String firstName, String lastName, String streetAddress,
                  String city, String stateProvince, String postalCode,
                  String countryId);

  void updateStaffSkills(Integer userId, List<Integer> prodIds);
}
```

Example 3–2 SRAdminFacadeBean.java Bean Class

```
package oracle.srdemo.model;

import java.util.ArrayList;
import java.util.List;
import java.util.Vector;
import javax.ejb.Stateless;
import oracle.srdemo.model.entities.ExpertiseArea;
import oracle.srdemo.model.entities.Product;
import oracle.srdemo.model.entities.User;
import oracle.toplink.sessions.Session;
import oracle.toplink.sessions.UnitOfWork;
import oracle.toplink.util.SessionFactory;

@Stateless(name="SRAdminFacade")
public class SRAdminFacadeBean implements SRAdminFacade {
  private SessionFactory sessionFactory;

  public SRAdminFacadeBean() {
    this.sessionFactory =
        new SessionFactory("META-INF/sessions.xml", "SRDemo");
  }

  /**
   * Constructor used during testing to use a local connection
   * @param sessionName
   */
  public SRAdminFacadeBean(String sessionName) {
```

```
    this.sessionFactory =
        new SessionFactory("META-INF/sessions.xml", sessionName);

  }

  public Object mergeEntity(Object entity) {
    UnitOfWork uow = getSessionFactory().acquireUnitOfWork();
    Object workingCopy = uow.readObject(entity);
    if (workingCopy == null)
      throw new RuntimeException("Could not find entity to update");
    uow.deepMergeClone(entity);
    uow.commit();

    return workingCopy;
  }

  public Object persistEntity(Object entity) {
    UnitOfWork uow = getSessionFactory().acquireUnitOfWork();
    Object newInstance = uow.registerNewObject(entity);
    uow.commit();

    return newInstance;
  }

  public Object refreshEntity(Object entity) {
    Session session = getSessionFactory().acquireUnitOfWork();
    Object refreshedEntity = session.refreshObject(entity);
    session.release();

    return refreshedEntity;
  }

  public void removeEntity(Object entity) {
    UnitOfWork uow = getSessionFactory().acquireUnitOfWork();
    Object workingCopy = uow.readObject(entity);
    if (workingCopy == null)
      throw new RuntimeException("Could not find entity to update");
    uow.deleteObject(workingCopy);
    uow.commit();
  }

  private SessionFactory getSessionFactory() {
    return this.sessionFactory;
  }

  public List<ExpertiseArea> findExpertiseByUserId(Integer userIdParam) {
    List<ExpertiseArea> result = null;

    if (userIdParam != null){
      Session session = getSessionFactory().acquireSession();
      Vector params = new Vector(1);
      params.add(userIdParam);
      result = (List<ExpertiseArea>)session.executeQuery("findExpertiseByUserId",
ExpertiseArea.class, params);
      session.release();
    }

    return result;
  }
```

```
  public ExpertiseArea createExpertiseArea(Product product, User user,
                                           Integer prodId, Integer userId,
                                           String expertiseLevel,
                                           String notes) {
    UnitOfWork uow = getSessionFactory().acquireUnitOfWork();
    ExpertiseArea newInstance =
(ExpertiseArea)uow.newInstance(ExpertiseArea.class);

    if (product == null) {
        product = (Product)uow.executeQuery("findProductById", Product.class,
prodId);
    }

    if (user == null){
        user = (User)uow.executeQuery("findUserById", User.class, userId);
    }

    newInstance.setProduct(product);
    newInstance.setUser(user);
    newInstance.setProdId(prodId);
    newInstance.setUserId(userId);
    newInstance.setExpertiseLevel(expertiseLevel);
    newInstance.setNotes(notes);
    uow.commit();

    return newInstance;
  }

  public Product createProduct(Integer prodId, String name, String image,
                               String description) {
    UnitOfWork uow = getSessionFactory().acquireUnitOfWork();
    Product newInstance = (Product)uow.newInstance(Product.class);
    newInstance.setProdId(prodId);
    newInstance.setName(name);
    newInstance.setImage(image);
    newInstance.setDescription(description);
    uow.commit();

    return newInstance;
  }

  public List<User> findAllStaffWithOpenAssignments() {
    Session session = getSessionFactory().acquireSession();
    List<User> result =
      (List<User>)session.executeQuery("findAllStaffWithOpenAssignments",
User.class);
      session.release();
    return result;
  }

  public User createUser(Integer userId, String userRole, String email,
                         String firstName, String lastName,
                         String streetAddress, String city,
                         String stateProvince, String postalCode,
                         String countryId) {
    UnitOfWork uow = getSessionFactory().acquireUnitOfWork();
    User newInstance = (User)uow.newInstance(User.class);
    newInstance.setUserId(userId);
    newInstance.setUserRole(userRole);
    newInstance.setEmail(email);
```

```
    newInstance.setFirstName(firstName);
    newInstance.setLastName(lastName);
    newInstance.setStreetAddress(streetAddress);
    newInstance.setCity(city);
    newInstance.setStateProvince(stateProvince);
    newInstance.setPostalCode(postalCode);
    newInstance.setCountryId(countryId);
    uow.commit();

    return newInstance;
  }

  public void updateStaffSkills(Integer userId, List<Integer> prodIds) {
    List<Integer> currentSkills;

    if (userId != null) {
        List<ExpertiseArea> currentExpertiseList =  findExpertiseByUserId(userId);
        currentSkills = new ArrayList(currentExpertiseList.size());

        //Look for deletions
        for(ExpertiseArea expertise:currentExpertiseList){
            Integer prodId = expertise.getProdId();
            currentSkills.add(prodId);

            if (!prodIds.contains(prodId)){
                removeEntity(expertise);
            }
        }

        //Look for additions
        for (Integer newSkillProdId: prodIds){
            if(!currentSkills.contains(newSkillProdId)){
                //need to add
                this.createExpertiseArea(null,
null,newSkillProdId,userId,"Qualified",null);
            }
        }
    }
  }
}
```

3.2.3 What You May Need to Know When Creating a Session Bean

Typically you create one session facade for every logical unit in your application. A task could be defined in a large scope, by a role for instance, such as creating a session facade for administrative client operations and another session facade for customer client operations.How you create and name your session facades can facilitate UI development, so tailoring your session facades toward a particular task and using names that describe the task is a good practice.

When you generate session facade methods, a findAll() method is created by default for each entity. If you do not want to generate this method, deselect it in the tree control on the Session Facade Options page.

When creating or editing session facade methods, you cannot select both TopLink and EJB entities. If the project is enabled for TopLink entities, only those entities will be available as session facade methods. Support for combining TopLink and EJB entities in a single session facade is planned for a future release.

3.2.4 How to Update an Existing Session Bean With New Entities

New session beans can be created at any time using the wizard. However, you may have an existing session bean that already contains custom implementation code that you want to update with new persistent data objects or methods.

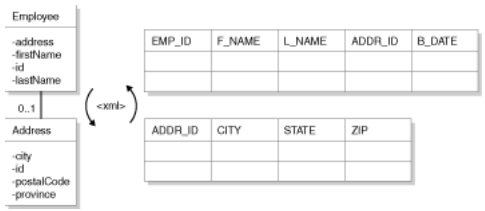
To update an existing session bean, right click on the session bean and choose Edit Session Facade. Use the Session Facade Options dialog to select the entities and methods to expose. Note that if you have created new entities, the Session Facade Options dialog will display new entities in the same project, but cannot detect entities in different projects.

3.3 Creating Classes to Map to Database Tables

The TopLink map (`.mwp` file) contains the information required to represent database tables as Java classes. You can use the Create TopLink Map wizard or the Mapping editor to create this data, or manually code the file using Java and the TopLink API.

Use this information, or metadata, to pass configuration information into the run-time environment. The run-time environment uses the information in conjunction with the persistent entities (Java objects or EJB entity beans) and the code written with the TopLink API, to complete the application.

Figure 3–3 TopLink Metadata



Descriptors

Descriptors describe how a Java class relates to a data source representation. They relate object classes to the data source at the data model level. For example, persistent class attributes may map to database columns.

TopLink uses descriptors to store the information that describes how an instance of a particular class can be represented in a data source (see Section 3.4, "Mapping Classes to Tables"). Most descriptor information can be defined by TopLink, then read from the project XML file at run time.

Persistent Classes

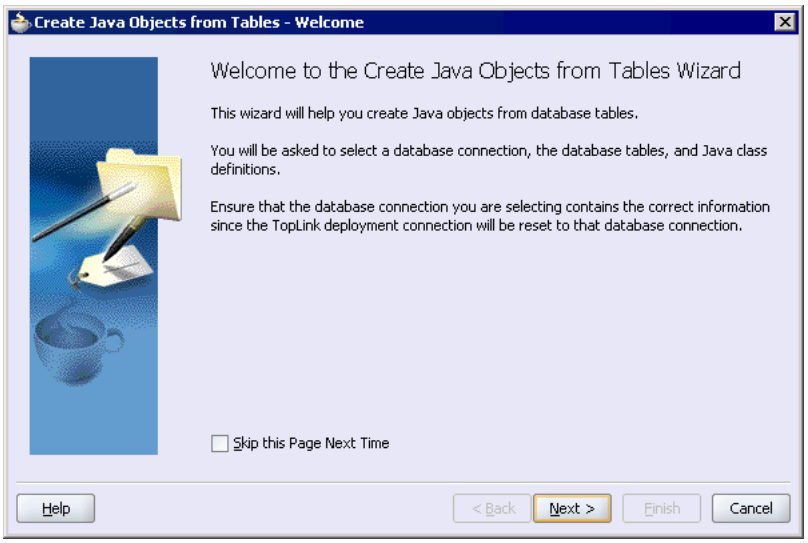
Any class that registers a descriptor with a TopLink database session is called a persistent class. TopLink does not require that persistent classes provide public accessor methods for any private or protected attributes stored in the database.

3.3.1 How to Create Classes

To automatically create Java classes from your database table, use the Create Java Objects from Tables wizard. With this wizard you can create the following:

- Java class for each table
- TopLink map
- Mapped attributes for each tables' columns

Figure 3–4 Create Java Objects from Tables Wizard

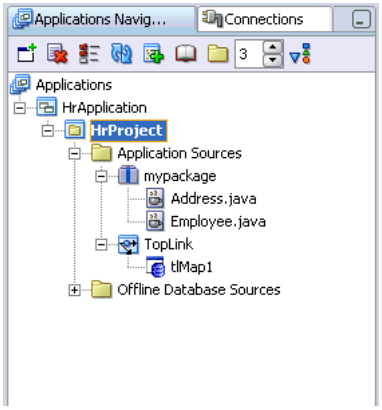


After creating the initial Java classes and TopLink mappings, use the Mapping editor to customize the information. Refer to the Oracle JDeveloper online help for additional information.

3.3.2 What Happens when you Create a Class

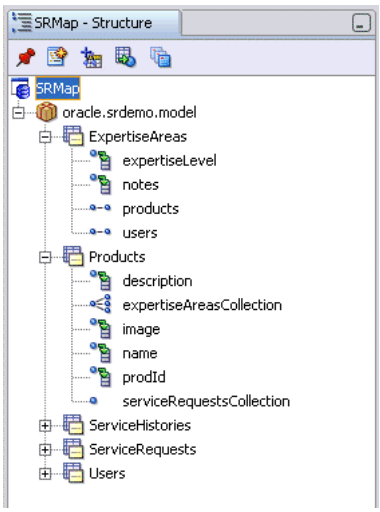
After completing the Create Java Objects from Tables wizard JDeveloper creates a TopLink map and adds it to the project.

Figure 3–5 Navigation Window



The wizard will also create TopLink descriptor and mappings for each Java attribute (as defined by the structure and relationships in the database).

Figure 3–6 Structure Window



3.3.3 What You May Need to Know

After creating a Java class from a database table, you can modify the generated TopLink descriptor and mappings. This section includes information on the following:

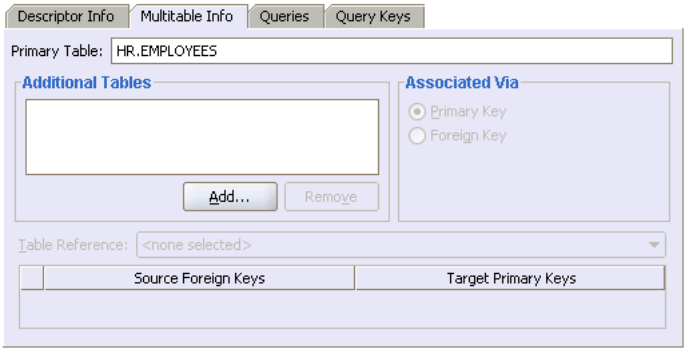
- Associating Descriptors with Different Database Tables
- Using Amendment Methods
- Modifying the Generated Code

3.3.3.1 Associating Descriptors with Different Database Tables

The Create Java Objects from Tables wizard will associate the TopLink descriptor with a specific database table.

Use the Multitable Info tab in the Mapping editor (as shown in Figure 3–7) to associate an amendment method with a descriptor.

Figure 3–7 Sample Multitable Info Tab



3.3.3.2 Using Amendment Methods

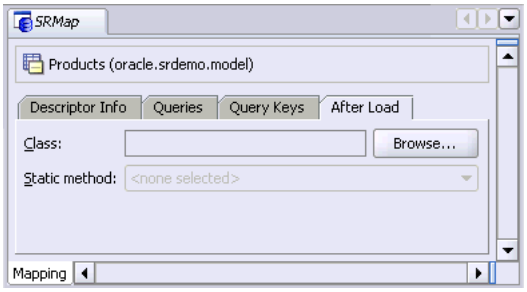
You can associate a static Java method to be called when a descriptor is loaded at run time. This method can amend the run-time descriptor instance through the descriptor Java code API. Use this method to make some advanced configuration options that may not be currently supported by the TopLink.

The Java method must have the following characteristics:

- Be public static.
- Take a single parameter of type `oracle.toplink.descriptors.ClassDescriptor`.

Use the After Load tab in the Mapping editor (as shown in Figure 3–8) to associate an amendment method with a descriptor.

Figure 3–8 Sample After Load Tab



3.3.3.3 Modifying the Generated Code

When using the Create Java Objects from Tables wizard, Oracle JDeveloper automatically generates the basic code for your Java classes.

Example 3–3 Sample Generated Java Class

```
package mypackage;
  import java.util.ArrayList;
  import java.util.List;

public class Address {
  /**Map employeeCollection <-> mypackage.Employee
   * @associates <{mypackage.Employee}>
   */

  private List employeeCollection;
  private Long addressId;
  private String pCode;
...
```

3.4 Mapping Classes to Tables

One of the greatest strengths of TopLink is its ability to transform data between an object representation and a representation specific to a data source. This transformation is called **mapping** and it is the core of a TopLink project.

A mapping corresponds to a single data member of a domain object. It associates the object data member with its data source representation and defines the means of performing the two-way conversion between object and data source.

TopLink uses the metadata produced by Mapping editor to describe how objects and beans map to the data source. This approach isolates persistence information from the object model—developers are free to design their ideal object model and DBAs are free to design their ideal schema.

3.4.1 Types of Mappings

Within ADF, TopLink supports relational and object-relational mappings.

- Relational Mappings – Mappings that transform any object data member type to a corresponding relational database (SQL) data source representation in any supported relational database. Relational mappings allow you to map an object model into a relational data model.
- Object-Relational Mappings – Mappings that transform certain object data member types to structured data source representations optimized for storage in specialized object-relational databases such as Oracle Database. Object-relational mappings allow you to map an object model into an object-relational data model.

3.4.2 Direct Mappings

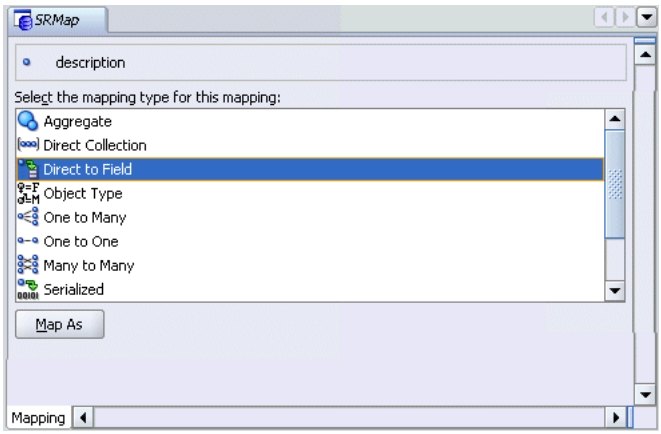
You can create the following direct mappings in TopLink:

- Direct-to-field mappings – Map a Java attribute directly to a database field.
- Type conversion mappings – Map Java values with simple type conversions, such as character to string.
- Object type mappings – Use an association to map values to the database.
- Serialized object mappings – Map serializable objects, such as multimedia objects, to database BLOB fields.
- Transformation mappings – Allow you to create custom mappings where one or more fields can be used to create the object be stored in the attribute.

3.4.3 How to Create Direct Mappings

To map create Java classes directly to database tables, select the Java attribute in the TopLink Map – Structure window. Oracle JDeveloper displays a list of the available mappings for the selected attribute (as shown in Figure 3–9).

Figure 3–9 Mapping Editor



You can also use TopLink Automap feature to automatically map the attributes in a specific Java class or package. Refer to the Oracle JDeveloper online help for more information.

3.4.4 What Happens when you Create a Direct Mapping

Example 3–4 illustrates the Java code that Oracle JDeveloper generates when you create a direct-to-field direct mapping. In this example, the `description` attribute of the `Products` class maps directly to a field on the database table.

Example 3–4 Java Code for a Direct Mapping

```
...
package oracle.srdemo.model;
public class Products {

  private String description;

    public String getDescription() {
      return this.description;
    }

    public void setDescription(String description) {
      this.description = description;
    }
}
```

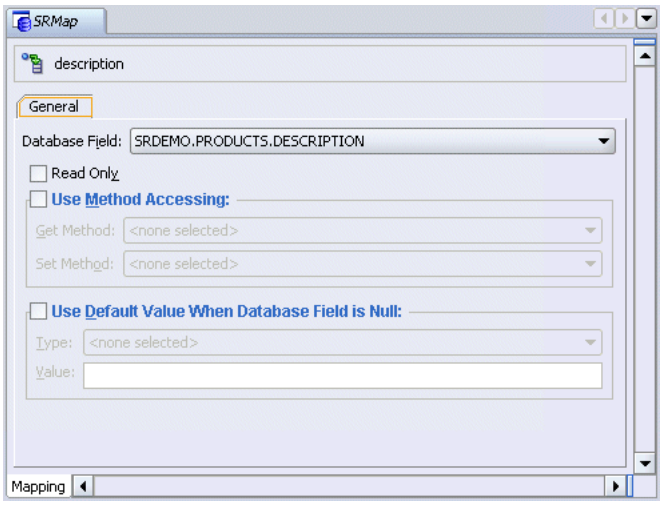
3.4.5 What You May Need to Know

Use the Mapping editor to customize the TopLink mappings. Some common customizations for direct mappings include:

- Specifying the mapping as "read only." These mappings will not be included during update or delete operations.
- Using custom `get` and `set` methods.
- Defining a default value. This value will be used if the actual field in the database is null.

Figure 3–10 shows the General tab of a direct-to-field mapping in the Mapping editor. Each direct mapping (see Section 3.4.2, "Direct Mappings") may have additional, specific options as well. Refer to the Oracle JDeveloper online help for more information.

Figure 3–10 Sample DIrect-to-Field Mapping



3.5 Mapping Related Classes with Relationships

Relational mappings define how persistent objects reference other persistent objects. TopLink provides the following relationship mappings:

- Direct collection mappings - Map Java collections of objects that do not have descriptors.
- Aggregate object mappings - Strict one-to-one mappings that require both objects to exist in the same database row.
- One-to-one mappings - map a reference to another persistent Java object to the database.
- Variable one-to-one mappings - Map a reference to an interface to the database.
- One-to-many mappings - Map Java collections of persistent objects to the database.
- Aggregate collection mappings also map Java collections of persistent objects to the database.
- Many-to-many mappings use an association table to map Java collections of persistent objects to the database

Do not confuse relational mappings with object-relational mappings. Object-relational mappings let you map an object model into an object-relational data model, such as Oracle Database. TopLink can create the following mappings:

- Object-Relational Structure Mapping – Map to object-relational aggregate structures (the `Struct` type in JDBC and the `OBJECT` type in Oracle Database)
- Object-Relational Reference Mapping – Map to object-relational references (the `Ref` type in JDBC and the `REF` type in Oracle Database)
- Object-Relational Array Mapping – Map a collection of primitive data to object-relational array data types (the Array type in JDBC and the VARRAY type in Oracle Database).
- Object-Relational Object Array Mapping – Map to object-relational array data types (the Array type in JDBC and the VARRAY type in Oracle Database).

- Object-Relational Nested Table Mapping – Map to object-relational nested tables (the Array type in JDBC and the NESTED TABLE type in Oracle Database)

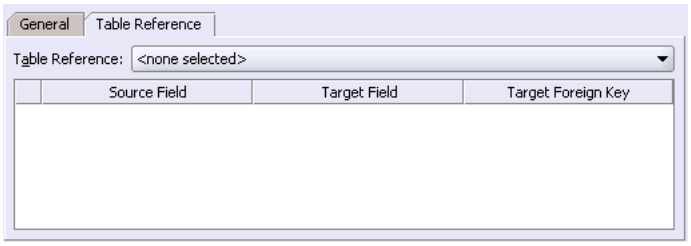
Although the Oracle TopLink runtime supports these mappings, they must be created in Java code – you cannot use the Mapping editor.

3.5.1 How to Create Relationship Mappings

Similarly to direct mappings (see Section 3.4.3, "How to Create Direct Mappings"), to map create Java classes directly to database tables, select the Java attribute in the TopLink Map – Structure window.

Relationship mappings contain a Table Reference tab in the Mapping editor to define (or create) relationships on the database tables.

Figure 3–11 Sample Table Reference Tab



Refer to the Oracle JDeveloper online help for more information.

3.5.2 What Happens when you Create a Relationship

Example 3–5 illustrates the Java code that Oracle JDeveloper generates when you create a direct-to-field direct mapping. In this example, the `address` attribute of the `ServiceRequest` class has a one-to-one relationship to another class, `User` (that is, each `ServiceRequest` was created by one `User`)

Example 3–5 Java Code for a Relationship Mapping

```
package oracle.srdemo.model;


    /**Map createdBy <-> oracle.srdemo.model.Users
     * @associates <{oracle.srdemo.model.Users}>
     */
  private ValueHolderInterface createdBy;

    public Users getCreatedBy() {
      return (Users)this.createdBy.getValue();
    }

    public void setCreatedBy(Users createdBy) {
      this.createdBy.setValue(createdBy);
    }
```

3.5.3 What You May Need to Know

Use the Mapping editor to customize the TopLink mappings. Some common customizations for relationship mappings include:

- Specifying the mapping as "read only." These mappings will not be included during update or delete operations.
- Using custom `get` and `set` methods.
- Defining a default value. This value will be used if the actual field in the database is null.
- Using indirection. When using indirection, TopLink uses an indirection object as a placeholder for the referenced object: TopLink defers reading the dependent object until you access that specific attribute.
- Configuring private or independent relationships. In a private relationship, the target object is a private component of the source object; destroying the source object will also destroy the target object. In an independent relationship, the source and target objects exist independently; destroying one object does not necessarily imply the destruction of the other.
- Specifying bidirectional relationship in which the two classes in the relationship reference each other with one-to-one mappings

Figure 3–12 shows the **General** tab of a one-to-one mapping in the Mapping editor. Use the **Table Reference** tab (see Figure 3–13) to define the foreign key reference for the mapping. Each direct mapping (see Section 3.5, "Mapping Related Classes with Relationships") may have additional, specific options as well. Refer to the Oracle JDeveloper online help for more information.

Figure 3–12 Sample One-to-One Mapping, General Tab

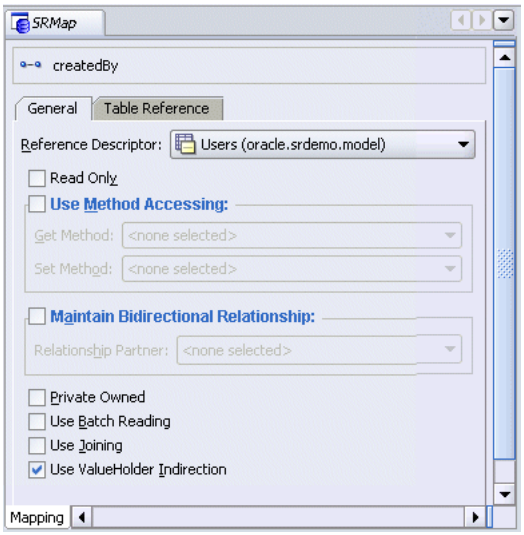
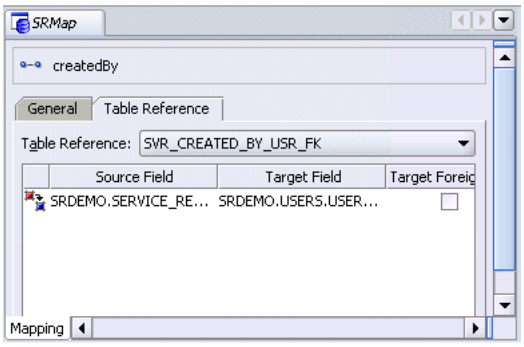


Figure 3–13 Sample One-to-One Mapping, Table Reference Tab



3.6 Finding Objects by Primary Key

TopLink provides a predefined finder (`findByPrimaryKey`) that takes a primary key as an `Object`. This finder is defined at runtime – not in the Mapping editor

Example 3–6 Executing a Primary Key Finder

```
{
    Employee employee = getEmployeeHome().findByPrimaryKey(primraryKey);
}
```

3.7 Querying Objects

To query objects, you can create a TopLink Named query then create a data control for the class specified in the query. This will expose the TopLink query to the data control.

A named query is a TopLink query that you create and store for later retrieval and execution. Named queries improve application performance because they are prepared once and they (and all their associated supporting objects) can be efficiently reused thereafter making them well suited for frequently executed operations.

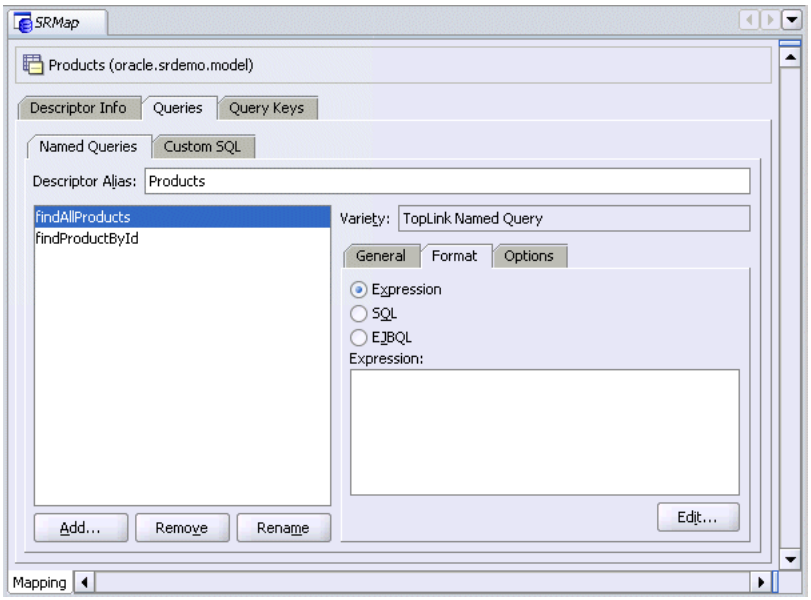
You can create the following queries:

- ReadAllQuery
- ReadObjectQuery

3.7.1 How to Create a Query

You can create TopLink Named Queries by using the TopLink expression builder, SQL expressions, or EJB QL expressions. Using the Mapping editor (see Figure 3–14), you can configure queries at the descriptor- or session-level.

Figure 3–14 Named Queries Tab



3.7.2 What You May Need to Know

3.7.2.1 Using a Query By Example

A **query by example** enables you to specify query selection criteria in the form of a sample object instance that you populate with only the attributes you want to use for the query. To define a query by example, provide a `ReadObjectQuery` or a `ReadAllQuery` with a sample persistent object instance and an optional query by example policy.

With ADF, a TopLink query by example performs *only* in-memory querying.

3.7.2.2 Sorting Query Results

You cannot configure the sort criteria of a TopLink query from Oracle JDeveloper. You must write a Java method, using descriptor amendment method. See Section 3.3.3.2, "Using Amendment Methods" for more information.

3.8 Creating and Modifying Objects with a Unit of Work

A database **transaction** is a set of operations (create, read, update, or delete) that either succeed or fail as a single operation. The database discards, or *rolls back*, unsuccessful transactions, leaving the database in its original state.

In TopLink, transactions are contained in the **unit of work** object. You acquire a unit of work from a session and using its API, you can control transactions directly or through a Java 2 Enterprise Edition (J2EE) application server transaction controller such as the Java Transaction API (JTA).

The unit of work isolates changes in a transaction from other threads until it successfully commits the changes to the database. Unlike other transaction mechanisms, the unit of work automatically manages changes to the objects in the transaction, the order of the changes, and changes that might invalidate other TopLink caches.

The unit of work manages these issues by calculating a minimal change set, ordering the database calls to comply with referential integrity rules and deadlock avoidance, and merging changed objects into the shared cache. In a clustered environment, the unit of work also synchronizes changes with the other servers in the coordinated cache.

3.8.1 How to Create a Unit of Work

Example 3–7 shows how to acquire a unit of work from a client session object.

Example 3–7 Acquiring a Unit of Work

```
public UnitOfWork acquireUnitOfWork() {

  Server server = getServer();

  if (server.hasExternalTransactionController()) {
    return server.getActiveUnitOfWork();
  server.acquireUnitOfWork();

}
```

3.8.1.1 Creating Objects with Unit of Work

When you create new objects in the unit of work, use the `registerObject` method to ensure that the unit of work writes the objects to the database at commit time.

The unit of work calculates commit order using foreign key information from one-to-one and one-to-many mappings. If you encounter constraint problems during a commit transaction, verify your mapping definitions. The order in which you register objects with the `registerObject` method does not affect the commit order.

Example 3–8 and Example 3–9 show how to create and persist a simple object (without relationships) using the clone returned by the unit of work `registerObject` method.

Example 3–8 Creating an Object: Preferred Method

```
UnitOfWork uow = session.acquireUnitOfWork();
    Pet pet = new Pet();
    Pet petClone = (Pet)uow.registerObject(pet);
    petClone.setId(100);
    petClone.setName("Fluffy");
    petClone.setType("Cat");
uow.commit();
```

Example 3–9 shows a common alternative.

Example 3–9 Creating an Object: Alternative Method

```
UnitOfWork uow = session.acquireUnitOfWork();
    Pet pet = new Pet();
    pet.setId(100);
    pet.setName("Fluffy");
    pet.setType("Cat");
    uow.registerObject(pet);
uow.commit();
```

Both approaches produce the following SQL:

```
INSERT INTO PET (ID, NAME, TYPE, PET_OWN_ID) VALUES (100, 'Fluffy', 'Cat', NULL)
```

Example 3–8 is preferred: it gets you into the pattern of working with clones and provides the most flexibility for future code changes. Working with combinations of new objects and clones can lead to confusion and unwanted results.

3.8.1.2 Typical Unit of Work Usage

TopLink uses the unit of work as follows:

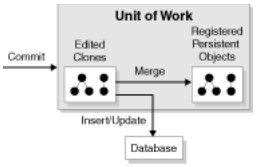
1. The client application acquires a unit of work from a session object.
2. The client application queries TopLink to obtain a cache object it wants to modify, and then registers the cache object with the unit of work.
3. The unit of work registers the object according to the object's change policy.

   By default, as each object is registered, the unit of work accesses the object from the session cache or database and creates a backup clone and working clone. The unit of work returns the working clone to the client application.
4. The client application modifies the working object returned by the unit of work.
5. The client application (or external transaction controller) commits the transaction.
6. The unit of work calculates the change set for each registered object according to the object's change policy.

   By default, at commit time, the unit of work compares the working clones to the backup clones and calculates the change set (that is, determines the minimum changes required). The comparison is done with a backup clone so that concurrent changes to the same objects will not result in incorrect changes being identified. The unit of work then attempts to commit any new or changed objects to the database.

   If the commit transaction succeeds, the unit of work merges changes into the shared session cache. Otherwise, no changes are made to the objects in the shared cache. If there are no changes, the unit of work does not start a new transaction.

Figure 3–15 The Life Cycle of a Unit of Work



Example 3–10 shows the default life cycle in code.

Example 3–10 Unit of Work Life Cycle

```
// The application reads a set of objects from the database
Vector employees = session.readAllObjects(Employee.class);

// The application specifies an employee to edit
. . .
Employee employee = (Employee) employees.elementAt(index);

try {
    // Acquire a unit of work from the session
    UnitOfWork uow = session.acquireUnitOfWork();

    // Register the object that is to be changed. Unit of work returns a clone
    // of the object and makes a backup copy of the original employee
```

```
    Employee employeeClone = (Employee)uow.registerObject(employee);

    // Make changes to the employee clone by adding a new phoneNumber.
    // If a new object is referred to by a clone, it does not have to be
    // registered. Unit of work determines it is a new object at commit time
    PhoneNumber newPhoneNumber = new PhoneNumber("cell","212","765-9002");
    employeeClone.addPhoneNumber(newPhoneNumber);

    // Commit the transaction: unit of work compares the employeeClone with
    // the backup copy of the employee, begins a transaction, and updates the
    // database with the changes. If successful, the transaction is committed
    // and the changes in employeeClone are merged into employee. If there is an
    // error updating the database, the transaction is rolled back and the
    // changes are not merged into the original employee object
    uow.commit();
} catch (DatabaseException ex) {

    // If the commit fails, the database is not changed. The unit of work should
    // be thrown away and application-specific action taken
}
// After the commit, the unit of work is no longer valid. Do not use further
```

3.8.2 What Happens when you Modify a Unit of Work

In Example 3–11, a `Pet` is read prior to a unit of work: the variable `pet` is the cache copy clone for that `Pet`. Inside the unit of work, register the cache copy to get a working copy clone. We then modify the working copy clone and commit the unit of work.

Example 3–11 Modifying an Object

```
// Read in any pet
Pet pet = (Pet)session.readObject(Pet.class);
UnitOfWork uow = session.acquireUnitOfWork();
    Pet petClone = (Pet) uow.registerObject(pet);
    petClone.setName("Furry");
uow.commit();
```

In Example 3–12, we take advantage of the fact that you can query through a unit of work and get back clones, saving the registration step. However, the drawback is that we do not have a handle to the cache copy clone.

If we wanted to do something with the updated `Pet` after the commit transaction, we would have to query the session to get it (remember that after a unit of work is committed, its clones are invalid and must not be used).

Example 3–12 Modifying an Object: Skipping the Registration Step

```
UnitOfWork uow = session.acquireUnitOfWork();
    Pet petClone = (Pet) uow.readObject(Pet.class);
    petClone.setName("Furry");
uow.commit();
```

Both approaches produce the following SQL:

```
UPDATE PET SET NAME = 'Furry' WHERE (ID = 100)
```

Take care when querying through a unit of work. All objects read in the query are registered in the unit of work and therefore will be checked for changes at commit time. Rather than do a `ReadAllQuery` through a unit of work, it is better for

performance to design your application to do the `ReadAllQuery` through a session, and then register in a unit of work only the objects that need to be changed.

3.8.2.1 Deleting Objects

To delete objects in a unit of work, use the `deleteObject` or `deleteAllObjects` method. When you delete an object that is not already registered in the unit of work, the unit of work registers the object automatically.

When you delete an object, TopLink deletes the object's privately owned child parts, because those parts cannot exist without the owning (parent) object. At commit time, the unit of work generates SQL to delete the objects, taking database constraints into account.

3.8.2.1.1 Explicitly Deleting Objects from the Database If there are cases where you have objects that will not be garbage collected through privately owned relationships (especially root objects in your object model), then you can explicitly tell TopLink to delete the row representing the object using the `deleteObject` API, as shown in Example 3–13.

Example 3–13 Explicitly Deleting

```
UnitOfWork uow = session.acquireUnitOfWork();
    pet petClone = (Pet)uow.readObject(Pet.class);
    uow.deleteObject(petClone);
uow.commit();
```

The preceding code generates the following SQL:

```
DELETE FROM PET WHERE (ID = 100)
```

3.8.3 What You May Need to Know

The TopLink unit of work is a powerful transaction model. In addition to the items listed in this section, you should review the "Understanding TopLink Transactions" chapter in the *Oracle TopLink Developer's Guide*.

3.8.3.1 Unit of Work and Change Policy

The unit of work tracks changes for a registered object based on the change policy you configure for the object's descriptor. If there are no changes, the unit of work will not start a new transaction.

Table 3–1 lists the change policies that TopLink provides.

Table 3–1 TopLink Change Policies

Change Policy	Applicable to...
Deferred Change Detection Policy	Wide range of object change characteristics. The default change policy.
Object-Level Change Tracking Policy	Objects with few attributes or with many attributes and many changed attributes.
Attribute Change Tracking Policy	Objects with many attributes and few changed attributes. The most efficient change policy. The default change policy for EJB 3.0 or 2.x CMP on OC4J.

3.8.3.2 Nested and Parallel Units of Work

You can use TopLink to create the following:

- Nested Unit of Work
- Parallel Unit of Work

3.8.3.2.1 Nested Unit of Work You can nest a unit of work (the *child*) within another unit of work (the *parent*). A nested unit of work does not commit changes to the database. Instead, it passes its changes to the parent unit of work, and the parent attempts to commit the changes at commit time. Nesting units of work lets you break a large transaction into smaller isolated transactions, and ensures that:

- Changes from each nested unit of work commit or fail as a group.
- Failure of a nested unit of work does not affect the commit or rollback transaction of other operations in the parent unit of work.
- Changes are presented to the database as a single transaction.

3.8.3.2.2 Parallel Unit of Work You can modify the same objects in multiple unit of work instances in parallel because the unit of work manipulates copies of objects. TopLink resolves any concurrency issues when the units of work commits the changes.

3.9 Interacting with Stored Procedures

You can provide a `StoredProcedureCall` object to any query instead of an expression or a SQL string, but the procedure must return all data required to build an instance of the class you query.

Example 3–14 A Read-All Query with a Stored Procedure

```
ReadAllQuery readAllQuery = new ReadAllQuery();
call = new StoredProcedureCall();
call.setProcedureName("Read_All_Employees");
readAllQuery.setCall(call);
Vector employees = (Vector) session.executeQuery(readAllQuery);
```

Using a `StoredProcedureCall`, you can access the following:

- Specifying an Input Parameter
- Specifying an Output Parameter
- Specifying an Input / Output Parameter
- Using an Output Parameter Event

Note: You no longer need to use `DatabaseQuery` method `bindAllParameters` when using a `StoredProcedureCall` with `OUT` or `INOUT` parameters. However, you should always specify the Java type for all `OUT` and `INOUT` parameters. If you do not, be aware of the fact that they default to type `String`.

3.9.1 Specifying an Input Parameter

In Example 3–15, you specify the parameter `POSTAL_CODE` as an input parameter using the `StoredProcedureCall` method `addNamedArgument`, and you can specify the value of the argument using method `addNamedArgumentValue`.

Example 3–15 Stored Procedure Call with an Input Parameter

```
StoredProcedureCall call = new StoredProcedureCall();
call.setProcedureName("CHECK_VALID_POSTAL_CODE");
call.addNamedArgument("POSTAL_CODE");
call.addNamedArgumentValue("L5J1H5");
call.addNamedOutputArgument(
    "IS_VALID",    // procedure parameter name
    "IS_VALID",    // out argument field name
    Integer.class  // Java type corresponding to type returned by procedure
);
ValueReadQuery query = new ValueReadQuery();
query.setCall(call);
Number isValid = (Number) session.executeQuery(query);
```

The order in which you add arguments must correspond to the order in which you add argument values. In Example 3–16, the argument `NAME` is bound to the value `Juliet` and the argument `SALARY` is bound to the value `80000`.

Example 3–16 Matching Arguments and Values in a Stored Procedure Call

```
StoredProcedureCall call = new StoredProcedureCall();
call.setProcedureName("CHECK_VALID_POSTAL_CODE");
call.addNamedArgument("NAME");
call.addNamedArgument("SALARY");
call.addNamedArgumentValue("Juliet");
call.addNamedArgumentValue(80000);
```

3.9.2 Specifying an Output Parameter

Output parameters enable the stored procedure to return additional information. You can use output parameters to define a `readObjectQuery` if they return all the fields required to build the object.

In Example 3–17, you specify the parameter IS_VALID as an output parameter using the `StoredProcedureCall` method `addNamedOutputArgument`.

Example 3–17 Stored Procedure Call with an Output Parameter

```
StoredProcedureCall call = new StoredProcedureCall();
call.setProcedureName("CHECK_VALID_POSTAL_CODE");
call.addNamedArgument("POSTAL_CODE");
call.addNamedOutputArgument(
    "IS_VALID",    // procedure parameter name
    "IS_VALID",    // out argument field name
    Integer.class  // Java type corresponding to type returned by procedure
);
ValueReadQuery query = new ValueReadQuery();
query.setCall(call);
query.addArgument("POSTAL_CODE");
Vector parameters = new Vector();
parameters.addElement("L5J1H5");
Number isValid = (Number) session.executeQuery(query,parameters);
```

Note: Not all databases support the use of output parameters to return data. However, because these databases generally support returning result sets from stored procedures, they do not require output parameters.

If you are using an Oracle database, you can make use of TopLink cursor and stream query results.

3.9.3 Specifying an Input / Output Parameter

In Example 3–18, you specify the parameter `LENGTH` as an input/output parameter and specify the value of the argument when it is passed to the stored procedure using the `StoredProcedureCall` method `addNamedInOutputArgumentValue`. If you do not want to specify a value for the argument, use method `addNamedInOutputArgument`.

Example 3–18 Stored Procedure Call with an Input/Output Parameter

```
StoredProcedureCall call = new StoredProcedureCall();
call.setProcedureName("CONVERT_FEET_TO_METERs");
call.addNamedInOutputArgumentValue(
    "LENGTH",          // procedure parameter name
    new Integer(100),  // in argument value
    "LENGTH",          // out argument field name
    Integer.class      // Java type corresponding to type returned by procedure
)
ValueReadQuery query = new ValueReadQuery();
query.setCall(call);
Integer metricLength = (Integer) session.executeQuery(query);
```

3.9.4 Using an Output Parameter Event

TopLink manages output parameter events for databases that support them. For example, if a stored procedure returns an error code that indicates that the application wants to check for an error condition, TopLink raises the session event `OutputParametersDetected` to allow the application to process the output parameters.

Example 3–19 Stored Procedure with Reset Set and Output Parameter Error Code

```
StoredProcedureCall call = new StoredProcedureCall();
call.setProcedureName("READ_EMPLOYEE");
call.addNamedArgument("EMP_ID");
call.addNamedOutputArgument(
    "ERROR_CODE",    // procedure parameter name
    "ERROR_CODE",    // out argument field name
    Integer.class  // Java type corresponding to type returned by procedure
);
ReadObjectQuery query = new ReadObjectQuery();
query.setCall(call);
query.addArgument("EMP_ID");
ErrorCodeListener listener = new ErrorCodeListener();
session.getEventManager().addListener(listener);
Vector args = new Vector();
args.addElement(new Integer(44));
Employee employee = (Employee) session.executeQuery(query, args);
```

3.9.5 Using a StoredFunctionCall

You use a `StoredProcedureCall` to invoke stored procedures defined on databases that support them. You can also use a `StoredFunctionCall` to invoke stored functions defined on databases that support them, that is, on databases for which the `DatabasePlatform` method `supportsStoredFunctions` returns `true`.

In general, both stored procedures and stored functions let you specify input parameters, output parameters, and input and output parameters. However, stored procedures need not return values, while stored functions always return a single value.

The `StoredFunctionCall` class extends `StoredProcedureCall` to add one new method: `setResult`. Use this method to specify the name (and alternatively both the name and type) under which TopLink stores the return value of the stored function.

When TopLink prepares a `StoredFunctionCall`, it validates its SQL and throws a `ValidationException` under the following circumstances:

- If your current platform does not support stored functions
- If you fail to specify the return type

In Example 3–20, note that the name of the stored function is set using `StoredFunctionCall` method `setProcedureName`.

Example 3–20 Creating a StoredFunctionCall

```
StoredFunctionCall functionCall = new StoredFunctionCall();
functionCall.setProcedureName("READ_EMPLOYEE");
functionCall.addNamedArgument("EMP_ID");
functionCall.setResult("FUNCTION_RESULT", String);
ReadObjectQuery query = new ReadObjectQuery();
query.setCall(functionCall);
query.addArgument("EMP_ID");
Vector args = new Vector();
args.addElement(new Integer(44));
Employee employee = (Employee) session.executeQuery(query, args);
```

3.9.6 Query Sequencing

With query sequencing, you can access a sequence resource using custom read (`ValueReadQuery`) and update (`DataModifyQuery`) queries and a preallocation size that you specify. This allows you to perform sequencing using stored procedures and allows you to access sequence resources that are not supported by the other sequencing types that TopLink provides.

3.10 Exposing Services with ADF Data Controls

The easiest way to bind services to a user interface is by using the ADF Data Control.

This section includes information on the following:

- How to Create ADF Data Controls
- Understanding the Data Control Files
- Understanding the Data Control Palette

3.10.1 How to Create ADF Data Controls

To create an ADF data control from an EJB session bean, right-click a session bean in the Navigator and choose Create Data Control or drag a session bean onto the Data Control Palette.

Note: J2EE developers who do not want to rely on Oracle-specific libraries may use managed beans instead of the ADF data control. This is more complex and beyond the scope of this book.

When you create a data control from an EJB 3.0 session bean, several XML files are generated and displayed in the Navigator. The generated files and the Data Control Palette are covered in the following sections.

3.10.2 Understanding the Data Control Files

When you create a data control, the following XML files are generated in the model

- DataControls.dcx - data control definition file
- <session_bean>.xml - structure definition file
- ReadOnlyCollection.xml - design-time XML file
- ReadOnlySingleValue.xml - design-time XML file
- UpdateableCollection.xml - design-time XML file
- UpdateableSingleValue.xml - design-time XML file
- <entity_name>.xml - entity definition file, one per entity

How these files are related and used are covered in greater detail in Appendix A, "Reference ADF XML Files".

3.10.2.1 About the DataControls.dcx File

The DataControls.dcx file is created when you register data controls on the business services. The .dcx file identifies the Oracle ADF model layer adapter classes that facilitate the interaction between the client and the available business service. In the case of EJB, web services, and bean-based data controls, you can edit this file in the Property Inspector to add or remove parameters and to alter data control settings. For example, you can use the .dcx file to set global properties for various items, such as whether to turn on/off sorting.

3.10.2.2 About the Structure Definition Files

When you register a session bean as an Oracle ADF data control, an XML definition file is created in the Model project for every session bean. This file is commonly referred to as the *structure definition* file. The structure definition file has the same name as the session bean, but has a .xml extension.

A structure definition is made up of three types of objects:

- Attributes
- Accessors
- Operations

3.10.2.3 About the Entity XML Files

When you create a data control, an XML file is generated for each entity (TopLink, EJB, or Java bean). These files are used for both ADF design-time and runtime. These files describe the structure of the class as well as UI hints, validators and labels for each attribute.

3.10.2.4 About the Design-time XML Files

Four files are generated solely for the design-time:

- ReadOnlyCollection.xml
- ReadOnlySingleValue.xml
- UpdateableCollection.xml
- UpdateableSingleValue.xml

These files are referenced by MethodAccessor definitions as the CollectionBeanClass which describes the available operations. Typically you do not edit this file by hand, but you could customize items on the Data Control Palette.

3.10.3 Understanding the Data Control Palette

Client developers use the Data Control Palette to create databound HTML elements (for JSP pages), databound Faces elements (for JSF JSP pages), and databound Swing UI components (for ADF Swing panels). The Data Control Palette comprises two selection lists:

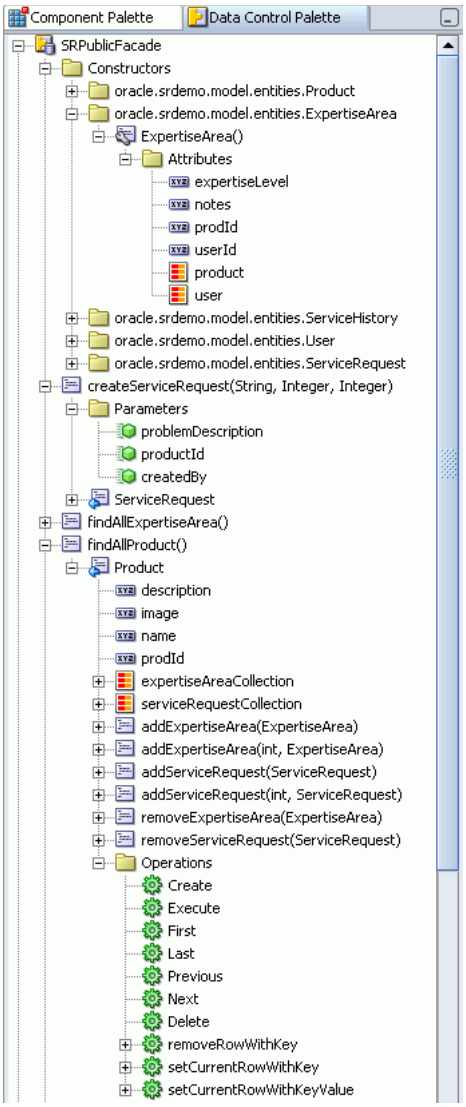
- Hierarchical display of available business objects, methods, and data control operations
- Dropdown list of appropriate visual elements that you can select for a given business object and drop into your open client document

Additionally, web application developers use the Data Control Palette to select methods provided by the business services that can be dropped onto the data pages and data actions of a page flow.

The Palette is a direct representation of the XML files examined in the previous sections, so by editing the files, you can change the elements contained in the Palette.

The hierarchical structure of the business services displayed in the Data Control Palette is determined by which business services you have registered with the data controls in your model project. The palette displays a separate root node for each business service that you register.

Figure 3–16 Data Control Palette



3.10.3.1 Overview of the Data Control Business Objects

The root node of the Data Control Palette represents the data control registered for the business service. Proceeding down the hierarchy from the root data control node, the palette represents bean-based business services as constructors, attributes, accessors or operations:

- Constructors - Createable types are contained within the Constructors node. These types call the default constructor for the object.
- Attributes - such as bean properties, which can define simple scalar value objects, structured objects (beans), or collections.
- Accessors - get() and set() methods.
- Operations - such as bean methods, which may or may not return a value or take method parameters. For Web Services, the Data Control Palette displays only operations.

For more information on using the Data Control Palette, see Chapter 5, "Displaying Data on a Page". For more information on the Data Control files and how they related to each other, see Appendix A, "Reference ADF XML Files".

3.10.3.2 Refreshing ADF Data Controls After Modifying Business Services

After you have already created the data control definition for your Model project, you may decide to update the data control after modifying your business services. Refreshing the data control definition makes the latest business service changes available to the ADF application.

The action you take to refresh the data control definition depends upon the type of change to the model project.

3.10.3.2.1 Viewing modified data controls in the Data Control Palette: If the palette is not yet displayed, select the View menu and choose Data Control Palette. If the palette is already displayed, right-click in the palette and choose Refresh.

3.10.3.2.2 Refreshing a data control definition for business services you have modified In the model project, define the new properties of the bean or other business service you want to create. Compile the .java file to regenerate the business service's metadata in its corresponding .xml file. If the modified business service is bean-based (such as an EJB session bean), right-click the bean's .xml file and choose Refresh.

Note: In the case of ADF Business Components, the data control definition is automatically updated whenever you make changes to your ADF BC project files.

3.10.3.2.3 Removing a data control definition for business services that have been removed: To To remove a data control definition, in the view project, select the DataBindings.dcx file and in the Structure window, select the data control node that represents the business service that no longer appears in your Model project. Right-click the data control node and choose Delete.

JDeveloper updates the data control definition file (DataBindings.dcx) in the Model project. The DataBindings.dcx file identifies the Oracle ADF model layer adapter classes that facilitate the interaction between the client and the available business services.

3.10.3.2.4 Updating a data control after renaming or moving a business services In the model project, if you rename your business service or move it to a new package, you must update the reference to the model project in the client's data control definition.

In the view project, select the DataBindings.dcx file. In the Structure window, select the data control node that represents the moved business service. In the Property Inspector, edit the Package attribute to supply the new package name.

Part II

Building the Web Interface

Part II contains the following chapters:

- Chapter 4, "Getting Started with ADF Faces"
- Chapter 5, "Displaying Data on a Page"
- Chapter 6, "Creating a Basic Page"
- Chapter 7, "Adding Tables"
- Chapter 8, "Displaying Master-Detail Data"
- Chapter 9, "Adding Page Navigation"
- Chapter 10, "Creating More Complex Pages"
- Chapter 11, "Using Complex UI Components"
- Chapter 12, "Using Validation and Conversion"
- Chapter 13, "Adding ADF Bindings to Existing Pages"
- Chapter 14, "Changing the Appearance of Your Application"
- Chapter 15, "Optimizing Application Performance with Caching"
- Chapter 16, "Testing and Debugging Web Applications"

4

Getting Started with ADF Faces

This chapter describes the process of setting up your user interface project to use ADF Faces. It also supplies basic information about creating and laying out a web page that will rely on ADF Faces components for the user interface.

The chapter includes the following sections:

- Section 4.1, "Introduction to ADF Faces"
- Section 4.2, "Setting Up a Workspace and Project"
- Section 4.3, "Creating a Web Page"
- Section 4.4, "Laying Out a Web Page"
- Section 4.5, "Creating and Using a Backing Bean for a Web Page"
- Section 4.6, "Best Practices for ADF Faces"

4.1 Introduction to ADF Faces

Oracle ADF Faces is a 100% JavaServer Faces (JSF) compliant component library that offers a broad set of enhanced UI components for JSF application development. Based on the JSF JSR 127 specification, ADF Faces components can be used in any IDE that supports JSF. More specifically, ADF Faces works with Sun's JSF Reference Implementation 1.1_01 (or later) and Apache MyFaces 1.0.8 (or later).

ADF Faces ensures a consistent look and feel for your application, allowing you to focus more on user interface interaction than look and feel compliance. The component library supports multi–language and translation implementations, and accessibility features. ADF Faces also supports multiple render kits for HTML, mobile, and telnet users—this means you can build web pages with the same components, regardless of the device that will be used to display the pages.

Using the partial-page rendering features of ADF Faces components, you can build interactive web pages that update the display without requiring a complete page refresh. In the future, Oracle plans to provide render kits that make even more sophisticated use of AJAX technologies—JavaScript, XML, and the Document Object Model (DOM)—to deliver more Rich Internet Applications with interactivity nearing that of desktop-style applications.

ADF Faces has many of the framework and component features most needed by JSF developers today, including:

- Partial-page rendering
- Client-side conversion and validation
- A process scope that makes it easier to pass values from one page to another

- A hybrid state-saving strategy that provides more efficient client-side state saving
- Built-in support for label and message display in all input components
- Built-in accessibility support in components
- Support for custom skins
- Support for mobile applications

ADF Faces UI components include advanced tables with column sorting and row selection capability, tree components for displaying data hierarchically, color and date pickers, and a host of other components such as menus, command buttons, shuttle choosers, and progress meters.

ADF Faces out-of-the-box components simplify user interaction, such as the input file component for uploading files, and the select input components with built-in dialog support for navigating to secondary windows and returning to the originating page with the selected values.

For more information about ADF Faces, refer to the following resources:

- ADF Faces Core tags at

  `http://www.oracle.com/technology/products/jdev/htdocs/partners/addins/exchange/jsf/doc/tagdoc/core/index.html`

- ADF Faces HTML tags at

  `http://www.oracle.com/technology/products/jdev/htdocs/partners/addins/exchange/jsf/doc/tagdoc/html/index.html`

- ADF Faces Javadocs at

  `http://www.oracle.com/technology/products/jdev/htdocs/partners/addins/exchange/jsf/doc/apidocs/index.html`

- ADF Faces developer's guide at

  `http://www.oracle.com/technology/products/jdev/htdocs/partners/addins/exchange/jsf/doc/devguide/index.html`

When you create JSF JSP pages that use ADF Faces components for the UI and use JSF technology for page navigation, you can leverage the advantages of the Oracle Application Development Framework (Oracle ADF) by using the ADF Model binding capabilities for the components in the pages. For information about data controls and the ADF Model, see Section 1.1.2, "Declarative Development with Oracle ADF and JavaServer Faces".

Table 4–1 shows the platforms currently supported for ADF Faces.

Table 4–1 Supported Platforms for ADF Faces

User Agent	Windows	Solaris	Mac OS X	Red Hat Linux	Windows Mobile	Palm OS
Internet Explorer	6.0 *				2003+	
Mozilla	1.7.x			1.7.x		
Firefox	1.0.x			1.0.x		
Safari			1.3, 2.0 **			
WebPro (Mobile)						3.0

* Accessibility and BiDi is only supported on IE on Windows.

** Apple bug fixes provided in Safari 1.3 patch 312.2 and Safari 2.0 patch 412.5 required.

> **Tip:** On a UNIX server box, button images may not render as expected. Assuming you're using JDK 1.4 or later, Oracle strongly recommends using `-Djava.awt.headless=true` as a command-line option with UNIX boxes.

Read this chapter to understand:

- How to create a workspace using an application template in JDeveloper
- What files are created for you in the view project when you add a JSF page and insert UI components
- How to use panel and layout components to create page layouts
- What JDeveloper does for you when you work with backing beans

4.2 Setting Up a Workspace and Project

JDeveloper provides application templates that enable you to quickly create the workspace and project structure with the appropriate combination of technologies already specified. The SRDemo application uses the **Web Application [JSF, EJB, TopLink]** application template, which creates one project for the data model, and one project for the controller and view (user interface) components in a workspace.

To create a new application workspace in JDeveloper and choose an application template:

1. Right-click the **Applications** node in the Application Navigator and choose **New Application**.
2. In the Create Application dialog, select the Web Application [JSF, EJB, TopLink] application template from the list.

You don't have to use JDeveloper application templates to create an application workspace—they are provided merely for your convenience.

At times you might already have an existing WAR file and you want to import it into JDeveloper.

To import a WAR file into a new project in JDeveloper:

1. Right-click your application workspace in the Application Navigator and choose **New Project**.
2. In the New Gallery, expand **General** in the **Categories** tree, and select **Projects**.
3. In the **Items** list, double-click **Project from WAR File**.
4. Follow the wizard instructions to complete creating the project.

4.2.1 What Happens When You Use an Application Template to Create a Workspace

By default, JDeveloper names the project for the data model **Model**, and the project for the user interface and controller **ViewController**. You can rename the projects using **File > Rename** after you've created them, or you can use **Tools > Manage Templates** to change the default names that JDeveloper uses.

> **Note:** The illustrations and project names used in this chapter are the JDeveloper default names. The SRDemo application, however, uses the project name **UserInterface** for the JSF view and controller components, and **DataModel** for the project that contains the EJB session beans and TopLink using plain old Java objects. The SRDemo application also has additional projects in the Application Navigator (for example, BuildAndDeploy), which you create manually to organize your application components into logical folders.

Figure 4–1 shows the Application Navigator view of the ViewController project after you create the workspace.

Figure 4–1 ViewController Project in the Navigator After You Create a Workspace

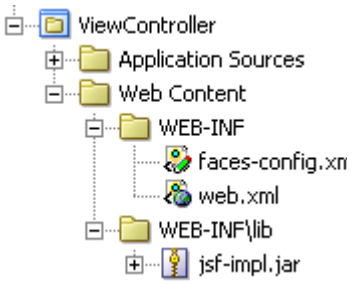


Figure 4–2 shows the actual folders JDeveloper creates in the `<JDEV_HOME>/jdev/mywork` folder in the file system.

Figure 4–2 ViewController Folders in the File System After You Create a Workspace

For example, if you created a workspace named Application1, the ViewController folder and its subfolders would be located in `<JDEV_HOME>/jdev/mywork/Application1` in the file system.

When you use the **Web Application [JSF, EJB, TopLink]** template to create a workspace, JDeveloper does the following for you:

- Creates a ViewController project that uses JSF technology. The project properties include:
  - **JSP Tag Libraries**: JSF Core, JSF HTML. See Table 4–2.
  - **Libraries**: JSF, Commons Beanutils, Commons Digester, Commons Logging, Commons Collections, JSTL.
  - **Technology Scope**: JSF, JSP and Servlets, Java, HTML.

  When you work in the ViewController project, the New Gallery will be filtered to show standard web technologies (including JSF) in the Web Tier category.

  By default, JDeveloper uses JSTL 1.1 and a J2EE 1.4 web container that supports Servlet 2.4 and JSP 2.0.

- Creates a starter `web.xml` file with default settings in `/WEB-INF` of the ViewController project. See Section 4.2.1.1, "Starter web.xml File" if you want to know what JDeveloper adds to `web.xml`.
- Creates an empty `faces-config.xml` file in `/WEB-INF` of the ViewController project. See Section 4.2.1.2, "Starter faces-config.xml File" if you want to learn more about `faces-config.xml`.

  Note that if you double-click **faces-config.xml** in the Application Navigator to open the file, JDeveloper creates a model folder in the ViewController folder in the file system, and adds the file `faces-config.oxd_faces` in the model folder. For information about the `faces-config.oxd_faces` file, see Section 4.3.2, "What Happens When You Create a JSF Page".
- Adds `jsf-impl.jar` in `/WEB-INF/lib` of the ViewController project.
- Creates a Model project that uses TopLink and EJB technology. For more information about the Model project, see Section 1.2.1.2, "Building the Business Service in the Model Project".

4.2.1.1 Starter web.xml File

Part of a JSF application's configuration is also determined by the contents of its J2EE application deployment descriptor, `web.xml`. The `web.xml` file defines everything about your application that a server needs to know (except the root context path, which is assigned by JDeveloper or the system administrator when the application is deployed). Typical runtime settings include initialization parameters, custom tag library location, and security settings.

Example 4–1 shows the starter `web.xml` file JDeveloper first creates for you.

Example 4–1 Starter web.xml File Created by JDeveloper

```
<?xml version = '1.0' encoding = 'windows-1252'?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
          http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
         version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee">
  <description>Empty web.xml file for Web Application</description>

  <servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>/faces/*</url-pattern>
  </servlet-mapping>
...
</web-app>
```

The JSF servlet and servlet mapping configuration settings are automatically added to the starter `web.xml` file when you first create a JSF project.

- JSF servlet: The JSF servlet is `javax.faces.webapp.FacesServlet`, which manages the request processing lifecycle for web applications utilizing JSF to construct the user interface. The configuration setting maps the JSF servlet to a symbolic name.

- JSF servlet mapping: The servlet mapping maps the URL pattern to the JSF servlet's symbolic name. You can use either a path prefix or an extension suffix pattern.

  By default, JDeveloper uses the path prefix `/faces/*`. This means that when the URL `http://localhost:8080/SRDemo/faces/index.jsp` is issued, the URL activates the JSF servlet, which strips off the `faces` prefix and loads the file `/SRDemo/index.jsp`.

To edit `web.xml` in JDeveloper, right-click **web.xml** in the Application Navigator and choose **Properties** from the context menu to open the Web Application Deployment Descriptor editor. If you're familiar with the configuration element names, you can also use the XML editor to modify `web.xml`.

For reference information about the configuration elements you can use in `web.xml` when you work with JSF, see Section A.8, "web.xml".

> **Note:** If you use ADF data controls to build databound web pages, JDeveloper adds the ADF binding filter and a servlet context parameter for the application binding container in `web.xml`. For more information, see Section 5.4, "Configuring the ADF Binding Filter".

4.2.1.2 Starter faces-config.xml File

The JSF configuration file is where you register a JSF application's resources such as custom validators and managed beans, and define all the page-to-page navigation rules. While an application can have any JSF configuration filename, typically the filename is `faces-config.xml`. Example 4–2 shows the starter `faces-config.xml` file JDeveloper first creates for you when you create a project that uses JSF technology.

Small applications usually have one `faces-config.xml` file. For information about using multiple configuration files, see Section 4.2.3, "What You May Need to Know About Multiple JSF Configuration Files".

Example 4–2 Starter faces-config.xml File Created by JDeveloper

```
<?xml version="1.0" encoding="windows-1252"?>
<!DOCTYPE faces-config PUBLIC
  "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.1//EN"
  "http://java.sun.com/dtd/web-facesconfig_1_1.dtd">
<faces-config xmlns="http://java.sun.com/JSF/Configuration">

</faces-config>
```

In JDeveloper you can use either editor to edit `faces-config.xml`:

- JSF Configuration Editor: Oracle recommends you use the JSF Configuration Editor because it provides visual editing.
- XML Source Editor: Use the source editor to edit the file directly, if you're familiar with the JSF configuration elements.

To launch the JSF Configuration Editor:

1. In the Application Navigator, double-click **faces-config.xml** to open the file.

   By default JDeveloper opens `faces-config.xml` in Diagram mode, as indicated by the active **Diagram** tab at the bottom of the editor window. When creating or

modifying JSF navigation rules, Oracle suggests you use the Diagram mode of the JSF Configuration Editor.

In JDeveloper a diagram file, which lets you create and manage page flows visually, is associated with `faces-config.xml`. For information about creating JSF navigation rules, see Chapter 9, "Adding Page Navigation".

2. To create or modify configuration elements other than navigation rules, use the Overview mode of the JSF Configuration Editor. At the bottom of the editor window, select **Overview**.

Both Overview and Diagram modes update the `faces-config.xml` file.

> **Tip:** JSF allows more than one `<application>` element in a single `faces-config.xml` file. The JSF Configuration Editor only allows you to edit the first `<application>` instance in the file. For any other `<application>` elements, you'll need to edit the file directly using the XML editor.

For reference information about the configuration elements you can use in `faces-config.xml`, see Section A.10, "faces-config.xml".

Note: If you use ADF data controls to build databound web pages, JDeveloper adds the ADF phase listener in `faces-config.xml`, as described in Section 5.2.3, "What Happens When You Use the Data Control Palette".

4.2.2 What You May Need to Know About the ViewController Project

The ViewController project contains the web content that includes the web pages and other resources of the web application. By default, the JDeveloper web application template you select adds the word "controller" to the project name to indicate that the web application will include certain files that define the application's flow or page navigation (controller), in addition to the web pages themselves (view).

Note: The concept of separating page navigation from page display is often referred to as *Model 2* to distinguish from earlier style (Model 1) applications that managed page navigation entirely within the pages themselves. In a Model 2 style application, the technology introduces a specialized servlet known as a *page controller* to handle page navigation events at runtime.

The technology that you use to create web pages in JDeveloper will determine the components of the ViewController project and the type of page controller your application will use. The SRDemo application uses JSF combined with JSP to build the web pages:

- JSF provides a component-based framework for displaying dynamic web content. It also provides its own page controller to manage the page navigation.
- JSP provides the presentation layer technology for JSF user interfaces. The JSF components are represented by special JSP custom tags in the JSP pages.

JDeveloper tools will help you to easily bind the JSF components with the Java objects of the Model project, thus creating databound UI components. As described earlier, the ViewController project contains the web pages for the user interface. To declaratively

bind UI components in web pages to a data model, the ViewController project must be able to access data controls in the Model project. To enable the ViewController project to access the data controls, a dependency on the Model project must be specified. The first time you drag an item from the Data Control Palette and drop it onto a JSF page, JDeveloper configures the dependency for you. If you wish to set the dependency on the Model project manually, use the following procedure.

To set dependency on a Model project for a ViewController project in JDeveloper:

1. Double-click **ViewController** in the Application Navigator to open the Project Properties dialog.
2. Select **Dependencies** and then select the checkbox next to **Model.jpr**.

4.2.3 What You May Need to Know About Multiple JSF Configuration Files

A JSF application can have more than one JSF configuration file. For example, if you need individual JSF configuration files for separate areas of your application, or if you choose to package libraries containing custom components or renderers, you can create a separate JSF configuration file for each area or library.

To create another JSF configuration file, simply use a text editor or use the JSF Page Flow & Configuration wizard provided by JDeveloper.

To launch the JSF Page Flow & Configuration wizard:

1. In the Application Navigator, right-click **ViewController** and choose **New**.
2. In the New Gallery window, expand **Web Tier**. Select **JSF** and then double-click **JSF Page Flow & Configuration (faces-config.xml)**.

When creating a JSF configuration file for custom components or other JSF classes delivered in a library JAR:

- Name the file `faces-config.xml` if you desire.
- Store the new file in `/META-INF`.
- Include this file in the JAR that you use to distribute your custom components or classes.

This is helpful for applications that have packaged libraries containing custom components and renderers.

When creating a JSF configuration file for a separate application area:

- Give the file a name other than `faces-config.xml`.
- Store the file in `/WEB-INF`.
- For JSF to read the new JSF configuration file as part of the application's configuration, specify the path to the file using the context parameter `javax.faces.CONFIG_FILES` in `web.xml`. The parameter value is a comma-separated list of the new configuration file names, if there is more than one file.

  If using the JSF Page Flow & Configuration wizard, select the **Add Reference to web.xml** checkbox to let JDeveloper register the new JSF configuration file for you in `web.xml`. Example 4–3 shows how multiple JSF configuration files are set in `web.xml` by JDeveloper if you select the checkbox.

This is helpful for large-scale applications that require separate configuration files for different areas of the application.

Example 4–3 Configuring for Multiple JSF Configuration Files in the web.xml File

```
<context-param>
  <param-name>javax.faces.CONFIG_FILES</param-name>
  <param-value>/WEB-INF/faces-config1.xml,/WEB-INF/faces-config2.xml</param-value>
</context-param>
```

Any JSF configuration file, whether it is named `faces-config.xml` or not, must conform to Sun's DTD located at `http://java.sun.com/dtd/web-facesconfig_1_x.dtd`. If you use the wizard to create a JSF configuration file, JDeveloper takes care of this for you.

If an application uses several JSF configuration files, at runtime JSF finds and loads the application's configuration settings in the following order:

1. Searches for files named `META-INF/faces-config.xml` in any JAR files for the application, and loads each as a configuration resource (in reverse order of the order in which they are found).
2. Searches for the `javax.faces.CONFIG_FILES` context parameter set in the application's `web.xml` file. JSF then loads each named file as a configuration resource.
3. Searches for a file named `faces-config.xml` in the `WEB-INF` directory and loads it as a configuration resource.

JSF then instantiates an `Application` class and populates it with the settings found in the various configuration files.

4.3 Creating a Web Page

While JSF supports a number of presentation layer technologies, JDeveloper uses JSP as the presentation technology for creating JSF web pages. When you use JSF with JSP, the JSF pages can be JSP pages (`.jsp`) or JSP documents (`.jspx`). JSP documents are well-formed XML documents, and the XML standard offers many benefits such as validation against a document type definition. Hence, Oracle recommends that you use JSP documents when you build your web pages using ADF Faces components. Unless otherwise noted, the term *JSF page* in this guide refers to both JSF JSP pages and JSF JSP documents.

JDeveloper gives you two ways to create JSF pages that will appear in your ViewController project:

- Launch the Create JSF JSP wizard from the **JSF** category in the New Gallery.

  OR

- Drag a **JSF Page** from the Component Palette onto the `faces-config.xml` file when the file is open in the Diagram mode of the JSF Configuration Editor.

Section 4.3.1, "How to Add a JSF Page" uses the latter technique. It also introduces the JSF Navigation Modeler, which allows you to plan out your application pages in the form of a diagram, to define the navigation flow between the pages, and to create the pages.

4.3.1 How to Add a JSF Page

Oracle recommends using the *JSF navigation diagram* to plan out and build your application page flow. Because the JSF navigation diagram visually represents the pages of the application, it is also an especially useful way to drill down into individual web pages when you want to edit them in the JSP/HTML Visual Editor.

To add a JSF page to your ViewController project using the JSF navigation diagram:

1. Expand the **ViewController - Web Content - WEB-INF** folder in the Application Navigator and double-click **faces-config.xml** or choose **Open JSF Navigation** from the **ViewController** context menu to open the `faces-config.xml` file.

   By default, JDeveloper opens the file in the **Diagram** tab, which is the JSF navigation diagram. If you've just started the ViewController project, the navigation diagram would be an empty drawing surface. If you don't see a blank drawing surface when you open `faces-config.xml`, select **Diagram** at the bottom of the editor.

2. In the Component Palette, select **JSF Navigation Diagram** from the dropdown list, and then select **JSF Page**.

3. Click on the diagram in the place where you want the page to appear. A page icon with a label for the page name appears on the diagram. The page icon has a yellow warning overlaid–this means you haven't created the actual page yet, just a representation of the page.

4. To create the new page, double-click the page icon and use the Create JSF JSP wizard.

   When creating a page in JDeveloper for the first time, be sure to complete all the steps of the wizard.

5. In Step 1 of the Create JSF JSP wizard, select **JSP Document (*.jspx)** for the JSP file **Type**.

6. Enter a filename and accept the default directory name or choose a new location. By default, JDeveloper saves files in `/ViewController/public_html` in the file system.

7. In Step 2 of the wizard, keep the default selection for not using component binding automatically.

8. In Step 3 of the wizard, make sure that these libraries are added to the **Selected Libraries** list:

   - **ADF Faces Components**
   - **ADF Faces HTML**
   - **JSF Core**
   - **JSF HTML**

9. Accept the default selection for the remaining page and click **Finish**.

Your new JSF page will open in the JSP/HTML Visual Editor where you can begin to lay out the page using ADF Faces components from the Component Palette or databound components dropped from the Data Control Palette.

If you switch back to the JSF navigation diagram (by clicking the **faces-config.xml** editor tab at the top), you will notice that the page icon no longer has the yellow warning overlaid.

> **Tip:** If you create new JSF pages using the wizard from the New Gallery, you can drag them from the Application Navigator to the JSF navigation diagram when designing the application page flow.

4.3.2 What Happens When You Create a JSF Page

Figure 4–3 shows the Application Navigator view of the ViewController project after you complete the wizard steps to add a JSF page.

Figure 4–3 ViewController Project in the Navigator After You Add a JSF Page

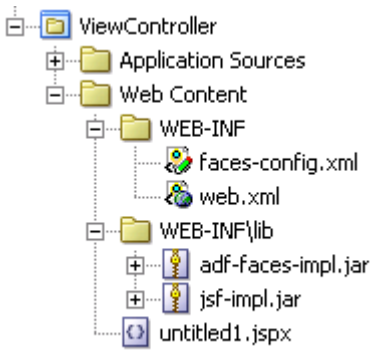
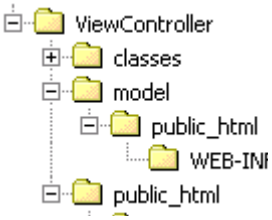


Figure 4–4 shows the actual folders JDeveloper creates in the `<JDEV_HOME>/jdev/mywork` folder in the file system.

Figure 4–4 ViewController Folders in the File System After You Add a JSF Page



JDeveloper does the following when you create your first JSF page in a ViewController project via the JSF navigation diagram:

- Adds `adf-faces-impl.jar` to `/WEB-INF/lib`.
- Adds these libraries to the ViewController project properties:
  - **JSP Tag Libraries**: ADF Faces Components, ADF Faces HTML. See Table 4–2.
  - **Libraries**: JSP Runtime, ADF Faces Runtime, ADF Common Runtime
- Creates the `faces-config.oxd_faces` file in the file system only, for example, in `<JDEV_HOME>/jdev/mywork/Application1/ViewController/model/public_html/WEB-INF`. When you plan out and build your page flow in the JSF navigation diagram, this is the file that holds all the diagram details such as layout and annotations. JDeveloper always maintains this file alongside its associated XML file, `faces-config.xml`. The `faces-config.oxd_faces` file is not visible in the Application or System Navigator.

Whether you create JSF pages by launching the Create JSF JSP wizard from the JSF navigation diagram or the New Gallery, by default JDeveloper creates starter pages that are JSF JSP 2.0 files, and automatically imports the JSF tag libraries into the starter pages. If you select to add the ADF Faces tag libraries in step 3 of the wizard, JDeveloper also imports the ADF Faces tag libraries into the starter pages. Example 4–4 shows a starter page for a JSF JSP document.

Example 4–4 Starter JSF JSP Document Created by JDeveloper

```
<?xml version='1.0' encoding='windows-1252'?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.0"
          xmlns:f="http://java.sun.com/jsf/core"
          xmlns:h="http://java.sun.com/jsf/html"
          xmlns:af="http://xmlns.oracle.com/adf/faces"
          xmlns:afh="http://xmlns.oracle.com/adf/faces/html"
  <jsp:output omit-xml-declaration="true" doctype-root-element="HTML"
              doctype-system="http://www.w3.org/TR/html4/loose.dtd"
              doctype-public="-//W3C//DTD HTML 4.01 Transitional//EN"/>
  <jsp:directive.page contentType="text/html;charset=windows-1252"/>
  <f:view>
    <html>
      <head>
        <meta http-equiv="Content-Type"
              content="text/html; charset=windows-1252"/>
        <title>untitled1</title>
      </head>
      <body>
        <h:form></h:form>
      </body>
    </html>
  </f:view>
</jsp:root>
```

4.3.3 What You May Need to Know About Using the JSF Navigation Diagram

In the JSF navigation diagram, you will notice that the label of the page icon has an initial slash (/), followed by the name of the page. The initial slash is required so that the page can be run from the diagram. If you remove the slash, JDeveloper will automatically reinstate it for you.

Be careful when renaming and deleting pages from the JSF navigation diagram:

- Renaming pages: If you rename a JSF page on a JSF navigation diagram, this is equivalent to removing a page with the original name from the diagram and adding a new one with the new name; the page icon changes to a page icon overlaid with the yellow warning, indicating that the page does not yet exist. If you have already created the underlying page, that page remains with its original name in the Application Navigator.

  Similarly, if you have a JSF page in the Application Navigator and the page icon is displayed on the diagram, if you now rename the page in the Application Navigator, this is equivalent to removing the original file and creating a new file. The diagram, however, retains the original name, and now displays the page icon overlaid with the yellow warning, indicating that the page does not exist.

- Deleting pages: When you delete a page icon in the JSF navigation diagram, the associated web page is no longer visible in the diagram. If you have created the actual file, it is still available from the **Web Content** folder in the ViewController project in the Application Navigator.

For information about the JSF navigation diagram and creating navigation rules, see Chapter 9, "Adding Page Navigation".

4.3.4 What You May Need to Know About ADF Faces Dependencies and Libraries

ADF Faces is compatible with JDK 1.4 (and higher), and cannot run on a server that supports only Sun's JSF Reference Implementation 1.0. The implementation must be JSF 1.1_01 (or later) or Apache MyFaces 1.0.8 (or later).

The ADF Faces deliverables are:

- `adf-faces-api.jar`: All public APIs of ADF Faces are in the `oracle.adf.view.faces` package.
- `adf-faces-impl.jar`: All private APIs of ADF Faces are in the `oracle.adfinternal.view.faces` package.

ADF Faces provides two tag libraries that you can use in your JSF pages:

- ADF Faces Core library
- ADF Faces HTML library

Table 4–2 shows the URIs and default prefixes for the ADF Faces and JSF tag libraries used in JDeveloper.

Table 4–2 ADF Faces and JSF Tag Libraries

Library	URI	Prefix
ADF Faces Core	`http://xmlns.oracle.com/adf/faces`	`af`
ADF Faces HTML	`http://xmlns.oracle.com/adf/faces/html`	`afh`
JSF Core	`http://java.sun.com/jsf/core`	`f`
JSF HTML	`http://java.sun.com/jsf/html`	`h`

JDeveloper also provides the ADF Faces Cache and ADF Faces Industrial tag libraries, which use the prefix `afc` and `afi`, respectively. For information about ADF Faces Cache, see Chapter 15, "Optimizing Application Performance with Caching". For information about ADF Faces Industrial, see the JDeveloper online help topic "Developing ADF Mobile Applications".

All JSF applications must be compliant with the Servlet specification, version 2.3 (or later) and the JSP specification, version 1.2 (or later). The J2EE web container that you deploy to must provide the necessary JAR files for the JavaServer Pages Standard Tag Library (JSTL), namely `jstl.jar` and `standard.jar`. The JSTL version to use depends on the J2EE web container:

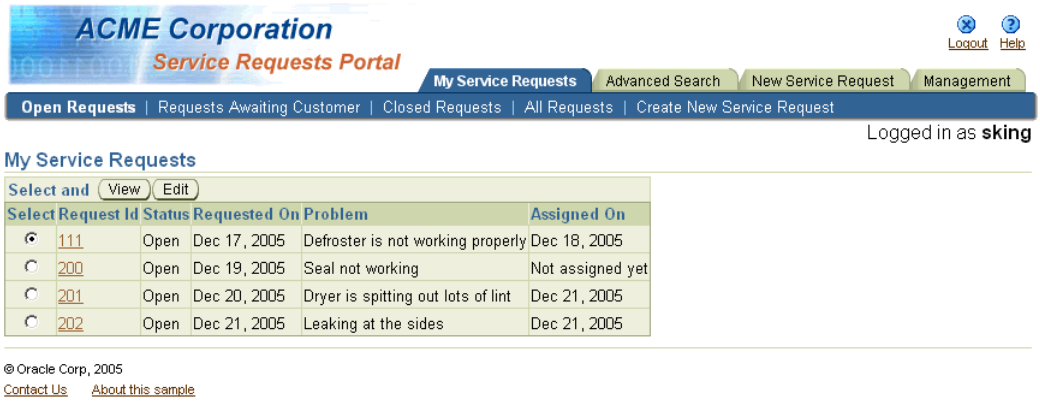
- JSTL 1.0—Requires a J2EE 1.3 web container that supports Servlet 2.3 and JSP 1.2
- JSTL 1.1—Requires a J2EE 1.4 web container that supports Servlet 2.4 and JSP 2.0

For complete information about ADF Faces and JSF deployment requirements, see Chapter 22, "Deploying ADF Applications".

4.4 Laying Out a Web Page

Most of the SRDemo pages use the ADF Faces `panelPage` component to lay out the entire page. The `panelPage` component lets you define specific areas on the page for branding images, navigation menus and buttons, and page-level or application-level text, ensuring that all web pages in the application will have a consistent look and feel. Figure 4–5 shows an example of a page created by using a `panelPage` component.

Figure 4–5 Page Layout Created with a PanelPage Component



After you create a new JSF page using the wizard, JDeveloper automatically opens the blank page in the JSP/HTML Visual Editor. To edit a page, you can use any combination of JDeveloper's page design tools you're comfortable with, namely:

- Structure window
- JSP/HTML Visual Editor
- XML Source Editor
- Property Inspector
- Component Palette

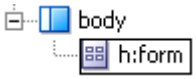
When you make changes to a page in one of the design tools, the other tools are automatically updated with the changes you made.

4.4.1 How to Add UI Components to a JSF Page

You can use both standard JSF components and ADF Faces components within the same JSF page. For example, to insert and use the `panelPage` component in a starter JSF page created by JDeveloper, you could use the following procedure.

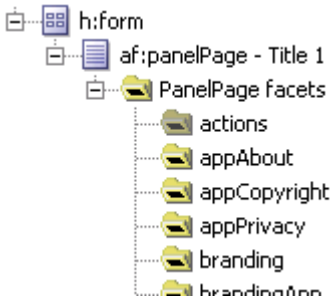
To insert UI components into a JSF page:

1. If not already open, double-click the starter JSF page in the Application Navigator to open it in the visual editor.
2. In the Component Palette, select **ADF Faces Core** from the dropdown list.
3. Drag and drop **PanelPage** from the palette to the page in the visual editor.



As you drag a component on the page in the visual editor, notice that the Structure window highlights the `h:form` component with a box outline, indicating that the `h:form` component is the target component. The target component is the component into which the source component will be inserted when it is dropped.

4. In the Structure window, right-click the newly inserted **af:panelPage** or any of the **PanelPage facets**, and choose from the **Insert before**, **Insert inside**, or **Insert after** menu to add the UI components you desire.



You create your input or search forms, tables, and other page body contents inside the `panelPage` component. For more information about `panelPage` and its facets, see Section 4.4.4, "Using the PanelPage Component".

> **Tip:** Using the context menu in the Structure window to add components ensures that you are inserting components into the correct target locations. You can also drag components from the Component Palette to the Structure window. As you drag a component on the Structure window, JDeveloper highlights the target location with a box outline or a line with an embedded arrow to indicate that the source component will be inserted in that target location when it is dropped. See Section 4.4.3.1, "Editing in the Structure Window" for additional information about inserting components using the Structure window.

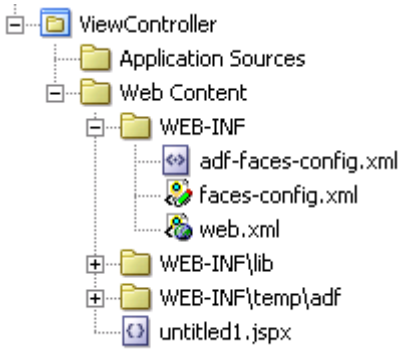
5. To edit the attributes for an inserted component, double-click the component in the Structure window to open a property editor, or select the component and then use the Property Inspector.

As you build your page layout by inserting components, you can also use the Data Control Palette to insert databound UI components. Simply drag the item from the Data Control Palette and drop it into the desired location on the page. For further information about using the Data Control Palette, see Chapter 5, "Displaying Data on a Page".

4.4.2 What Happens When You First Insert an ADF Faces Component

Figure 4–6 shows the Application Navigator view of the ViewController project after adding your first ADF Faces component in a page.

Figure 4–6 ViewController Project in the Navigator After You Insert the First ADF Faces Component



When you first add an ADF Faces component to a JSF page, JDeveloper automatically does the following:

- Imports the ADF Faces Core and HTML tag libraries (if not already inserted) into the page. See Example 4–4.
- Replaces the `html`, `head`, and `body` tags with `afh:html`, `afh:head`, and `afh:body`, respectively. See Example 4–5.
- Adds the ADF Faces filter and mapping configuration settings to `web.xml`. See Section 4.4.2.1, "More About the web.xml File".
- Adds the ADF Faces default render kit configuration setting to `faces-config.xml`. See Section 4.4.2.2, "More About the faces-config.xml File".
- Creates a starter `adf-faces-config.xml` in `/WEB-INF` of the ViewController project. See Section 4.4.2.3, "Starter adf-faces-config.xml File".
- Creates the `/ViewController/public_html/WEB-INF/temp/adf` folder in the file system. This folder contains images and styles that JDeveloper uses for ADF Faces components. You might not see the folder in the Application Navigator until you close and reopen the workspace.

> **Tip:** The `WEB-INF/lib` and `WEB-INF/temp/adf` folders are used by JDeveloper at runtime only. To reduce clutter in the Application Navigator, you may exclude them from the ViewController project. Double-click **ViewController** to open the Project Properties dialog. Under **Project Content**, select **Web Application** and then use the **Excluded** tab to add the folders you wish to exclude.

Example 4–5 JSF JSP Document After You Add the First ADF Faces Component

```
<?xml version='1.0' encoding='windows-1252'?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.0"
          xmlns:f="http://java.sun.com/jsf/core"
          xmlns:h="http://java.sun.com/jsf/html"
          xmlns:afh="http://xmlns.oracle.com/adf/faces/html"
          xmlns:af="http://xmlns.oracle.com/adf/faces">
  <jsp:output omit-xml-declaration="true" doctype-root-element="HTML"
              doctype-system="http://www.w3.org/TR/html4/loose.dtd"
              doctype-public="-//W3C//DTD HTML 4.01 Transitional//EN"/>
  <jsp:directive.page contentType="text/html;charset=windows-1252"/>
  <f:view>
    <afh:html>
      <afh:head title="untitled1">
        <meta http-equiv="Content-Type"
              content="text/html; charset=windows-1252"/>
      </afh:head>
      <afh:body>
        <h:form>
          <af:panelPage title="Title 1">
            <f:facet name="menu1"/>
            <f:facet name="menuGlobal"/>
            <f:facet name="branding"/>
            <f:facet name="brandingApp"/>
            <f:facet name="appCopyright"/>
            <f:facet name="appPrivacy"/>
            <f:facet name="appAbout"/>
          </af:panelPage>
        </h:form>
      </afh:body>
```

```
    </afh:html>
  </f:view>
</jsp:root>
```

4.4.2.1 More About the web.xml File

When you insert an ADF Faces component into a JSF page for the first time, JDeveloper automatically inserts the following ADF Faces configuration settings into `web.xml`:

- ADF Faces filter: Installs `oracle.adf.view.faces.webapp.AdfFacesFilter`, which is a servlet filter to ensure that ADF Faces is properly initialized by establishing a `AdfFacesContext` object. `AdfFacesFilter` is also required for processing file uploads. The configuration setting maps `AdfFacesFilter` to a symbolic name.
- ADF Faces filter mapping: Maps the JSF servlet's symbolic name to the ADF Faces filter.
- ADF Faces resource servlet: Installs `oracle.adf.view.faces.webapp.ResourceServlet`, which serves up web application resources (such as images, style sheets, and JavaScript libraries) by delegating to a `ResourceLoader`. The configuration setting maps `ResourceServlet` to a symbolic name.
- ADF Faces resource mapping: Maps the URL pattern to the ADF Faces resource servlet's symbolic name.

Example 4–6 shows the `web.xml` file after you add the first ADF Faces component.

Example 4–6 Configuring for ADF Faces in the web.xml File

```
<?xml version = '1.0' encoding = 'windows-1252'?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
          http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
         version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee">
  <description>Empty web.xml file for Web Application</description>

  <!-- Installs the ADF Faces filter -- >
  <filter>
    <filter-name>adfFaces</filter-name>
    <filter-class>oracle.adf.view.faces.webapp.AdfFacesFilter</filter-class>
  </filter>

  <!-- Adds the mapping to ADF Faces filter -- >
  <filter-mapping>
    <filter-name>adfFaces</filter-name>
    <servlet-name>Faces Servlet</servlet-name>
  </filter-mapping>

  <servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <!-- Installs the ADF Faces ResourceServlet -- >
  <servlet>
    <servlet-name>resources</servlet-name>
    <servlet-class>oracle.adf.view.faces.webapp.ResourceServlet</servlet-class>
```

```
  </servlet>

  <servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>/faces/*</url-pattern>
  </servlet-mapping>

  <!-- Maps URL pattern to the ResourceServlet's symbolic name -->
  <servlet-mapping>
    <servlet-name>resources</servlet-name>
    <url-pattern>/adf/*</url-pattern>
  </servlet-mapping>
...
</web-app>
```

For reference information about the configuration elements you can use in `web.xml` when you work ADF Faces, see Section A.8.1, "Tasks Supported by the web.xml File".

> **Tip:** If you use multiple filters in your application, make sure that they are listed in `web.xml` in the order in which you want to run them. At runtime, the filters are called in the sequence listed in that file.

4.4.2.2 More About the faces-config.xml File

As mentioned earlier, JDeveloper creates one empty `faces-config.xml` file for you when you create a new project that uses JSF technology. When you insert an ADF Faces component into a JSF page for the first time, JDeveloper automatically inserts the default render kit for ADF components into `faces-config.xml`, as shown in Example 4–7.

Example 4–7 Configuring for ADF Faces Components in the faces-config.xml File

```
<?xml version="1.0" encoding="windows-1252"?>
<!DOCTYPE faces-config PUBLIC
  "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.1//EN"
  "http://java.sun.com/dtd/web-facesconfig_1_1.dtd">
<faces-config xmlns="http://java.sun.com/JSF/Configuration">
  ...
  <!-- Default render kit for ADF components -->
  <application>
    <default-render-kit-id>oracle.adf.core</default-render-kit-id>
  </application>
  ...
</faces-config>
```

4.4.2.3 Starter adf-faces-config.xml File

When you create a JSF application using ADF Faces components, you configure ADF Faces–specific features (such as skin family and level of page accessibility support) in the `adf-faces-config.xml` file. The `adf-faces-config.xml` file has a simple XML structure that enables you to define element properties using the JSF expression language (EL) or static values.

In JDeveloper, when you insert an ADF Faces component into a JSF page for the first time, a starter `adf-faces-config.xml` file is automatically created for you in the `/WEB-INF` directory of your ViewController project. Example 4–8 shows the starter `adf-faces-config.xml` file.

Example 4–8 Starter adf-faces-config.xml File Created by JDeveloper

```
<?xml version="1.0" encoding="windows-1252"?>
<adf-faces-config xmlns="http://xmlns.oracle.com/adf/view/faces/config">

  <skin-family>oracle</skin-family>

</adf-faces-config>
```

By default JDeveloper uses the Oracle skin family for a JSF application. You can change this to `minimal` or use a custom skin. The SRDemo application uses the `srdemo` skin. If you wish to use a custom skin, you need to create the `adf-faces-skins.xml` configuration file, and modify `adf-faces-config.xml` to use the custom skin. For more information, see Section 14.3.1, "How to Use Skins".

To edit the `adf-faces-config.xml` file in JDeveloper, use the following procedure.

To edit the adf-faces-config.xml file:

1. In the Application Navigator, double-click **adf-faces-config.xml** to open the file in the XML editor.
2. If you're familiar with the element names, enter them in the editor. Otherwise use the Structure window to help you insert them.
3. To use the Structure window, follow these steps:
   a. Right-click any element to choose from the **Insert before** or **Insert after** menu, and click the element you wish to insert.
   b. Double-click the newly inserted element in the Structure window to open it in the properties editor.
   c. Enter a value or select one from a dropdown list (if available).

      In most cases you can enter either a JSF EL expression (such as `#{view.locale.language=='en' ? 'minimal' : 'oracle'}`) or a static value (e.g., `<debug-output>true</debug-output>`). EL expressions are dynamically reevaluated on each request, and must return an appropriate object (for example, a Boolean object).

      Note: All elements can appear in any order within the root element `<adf-faces-config>`. You can include multiple instances of any element. For reference information about the configuration elements you can use in `adf-faces-config.xml`, see Section A.11, "adf-faces-config.xml".

Typically, you would want to configure the following in `adf-faces-config.xml`:

- Level of page accessibility support (See Section 4.6, "Best Practices for ADF Faces")
- Skin family (See Section 14.3, "Using Skins to Change the Look and Feel")
- Time zone (See Section 14.4.2, "How to Configure Optional Localization Properties for ADF Faces")
- Enhanced debugging output (See Section A.11.1.3, "Configuring For Enhanced Debugging Output")
- Oracle Help for the Web (OHW) URL (See Section A.11.1.11, "Configuring the Help Site URL")

You can also register a custom file upload processor for uploading files. For information, see Section 11.6.5, "Configuring a Custom Uploaded File Processor".

Once you have configured elements in the `adf-faces-config.xml` file, you can retrieve the property values programmatically or by using JSF EL expressions. For more information, see Appendix A.11.1.12, "Retrieving Configuration Property Values From adf-faces-config.xml".

4.4.3 What You May Need to Know About Creating JSF Pages

Consider the following when you're developing JSF web pages:

- Do not use JSTL and HTML tags in a JSF page. JSTL tags cannot work with JSF at all prior to J2EE 1.5, and HTML tags inside of JSF tags often mean you need to use `f:verbatim`.

  For example you can't use `c:forEach` around JSF tags at all. When you nest a JSF tag inside a non-JSF tag that iterates over its body, the first time the page is processed the nested tag is invoked once for each item in the collection, creating a new component on each invocation. On subsequent requests because the number of items might be different, there is no good way to resolve the problem of needing a new component ID for each iteration: JSP page scoped variables cannot be seen by JSF; JSF request scoped variables in a previous rendering phase are not available in the current postback request.

  Other non-JSF tags may be used with JSF tags but only with great care. For example, if you use `c:if` and `c:choose`, the `id` attributes of nested JSF tags must be set; if you nest non-JSF tags within JSF tags, you must wrap the non-JSF tags in `f:verbatim`; if you dynamically include JSP pages that contain JSF content, you must use `f:subview` and also wrap all included non-JSF content in `f:verbatim`.

- In the SRDemo user interface, all String resources (for example, page titles and field labels) that are not retrieved from the ADF Model are added to a resource properties file in the ViewController project. If you use a resource properties file to hold the UI strings, use the `f:loadBundle` tag to load the properties file in the JSF page. For more information about resource bundles and the `f:loadBundle` tag, see Section 14.4, "Internationalizing Your Application".

- There is no requirement to use the ADF Faces `af:form` tag when you're using ADF Faces components—you can use the standard JSF `h:form` with all ADF Faces components. If you do use `af:form`, note that the `af:form` component does not implement the JSF `NamingContainer` API. This means a component's ID in the generated HTML does not include the form's ID as a prefix. For pages with multiple forms, this implies you can't reuse ID values among the forms. For example, this code snippet generates the component ID `foo:bar` for `inputText`:

  ```
  <h:form id="foo">
    <af:inputText id="bar"/>
  </h:form>
  ```

  But the following code snippet generates the component ID `bar2` for `inputText`:

  ```
  <af:form id="foo2">
    <af:inputText id="bar2"/>
  </af:form>
  ```

  The advantages of using `af:form` are:

  – It is easier to write JavaScript because it does not result in prefixed "name" and "id" attributes in its contents (as explained above).

  - It results in more concise HTML, for example, in cases where you may not know the form's ID.
  - You can use some CSS features on the fields.
  - You can set a default command for form submission. Set the `defaultCommand` attribute on `af:form` to the ID of the command button that is to be used as the default submit button when the Enter key is pressed. By defining a default command button for a form, when the user presses the Enter key, an `ActionEvent` is created and the form submitted. If a default command button is not defined for a form, pressing Enter will not submit the form, and the page simply redisplays.

- The `afh:body` tag enables partial page rendering (PPR) in a page. If a page cannot use the `afh:body` tag and PPR support is desired, use the `af:panelPartialRoot` tag in place of the `afh:body` tag. For information about PPR, see Section 11.4, "Enabling Partial Page Rendering".
- The `af:document` tag generates the standard root elements of an HTML page, namely `html`, `head`, and `body`, so you can use `af:document` in place of `afh:html`, `afh:head`, and `afh:body`.

For more tips on using ADF Faces components, see Section 4.6, "Best Practices for ADF Faces".

4.4.3.1 Editing in the Structure Window

In the Structure window while inserting, copying, or moving elements, you select an insertion point on the structure that is shown for the page, in relation to a target element. JDeveloper provides visual cues to indicate the location of the insertion point before, after, or contained inside a target element.

When dragging an element to an insertion point, do one of the following:

- To insert an element before a target element, drag it towards the top of the element until you see a horizontal line with an embedded up arrow, and then release the mouse button.
- To insert an element after a target element, drag it towards the bottom of the element until you see a horizontal line with an embedded down arrow, and then release the mouse button.
- To insert or contain an element inside a target element, drag it over the element until it is surrounded by a box outline, and then release the mouse button. If the element is not available to contain the inserted element, the element will be inserted after the target element.

Tip: A disallowed insertion point is indicated when the drag cursor changes to a circle with a slash.

4.4.3.2 Displaying Errors

Most of the SRDemo pages use the `af:messages` tag to display error messages. When you create databound pages using the Data Control Palette, ADF Faces automatically inserts the `af:messages` tag for you at the top of the page. When there are errors at runtime, ADF Faces automatically displays the messages in a message box

offset by color. For more information about error messages, see Section 12.7, "Displaying Error Messages".

In addition to reporting errors in a message box, you could use a general JSF error handling page for displaying fatal errors such as stack traces in a formatted manner. If you use a general error handling page, use the `<error-page>` element in `web.xml` to specify a type of exception for the error page (as shown in Example 4–9), or specify the error page using the JSP page directive (as shown in Example 4–10).

Example 4–9 Configuring Error-Page and Exception-Type in the web.xml File

```
<error-page>
  <exception-type>java.lang.Exception</exception-type>
  <location>/faces/infrastructure/SRError.jspx</location>
</error-page>
```

Example 4–10 Specifying ErrorPage in a JSF Page Using JSP Directive

```
<jsp:root ...>
  <jsp:output ...>
  <jsp:directive.page contentType="text/html;charset=windows-1252"
                      errorPage="faces/SRError.jspx"/>
  <f:view></f:view>
</jsp:root>
```

Consider the following if you intend to create and use a general JSF JSP error page:

- Due to a current limitation in Sun's JSF reference implementation, if you use the Create JSF JSP wizard in JDeveloper to create a JSF JSP error page, you need to replace `<f:view></f:view>` with `<f:subview></f:subview>`.
- In `web.xml` you need to add the following settings to ADF Faces filter mapping:

  ```
  <dispatcher>REQUEST</dispatcher>
  <dispatcher>ERROR</dispatcher>
  ```

- In the JSF page that uses the error page, `<jsp:directive errorPage=""/>` needs to include the `faces/` prefix in the `errorpage` URI, as shown in this code snippet:

  ```
  <jsp:directive.page contentType="text/html;charset=windows-1252"
                      errorPage="faces/SRError.jspx"/>
  ```

4.4.4 Using the PanelPage Component

The SRDemo pages use `panelPage` as the main ADF Faces layout component, which lets you lay out an entire page with specific areas for navigation menus, branding images, and page body contents, as illustrated in Figure 4–5.

The `panelPage` component uses facets (or JSF `f:facet` tags) to render children components in specific, predefined locations on the page. Consider a facet as a placeholder for one child component. Each facet has a name and a purpose, which determines where the child component is to be rendered relative to the parent component. The child component is often a container component for other child components.

The `panelPage` component uses `menu1`, `menu2`, and `menu3` facets for creating hierarchical, navigation menus that enable users to go quickly to related pages in the application. In the menu facets you could either:

- Manually insert the menu components (such `menuTabs` and `menuBar`) and their children menu items. By manually inserting individual children components, you need a lot of code in your JSF pages, which is time-consuming to create and maintain.

  For example, to create two menu tabs with subtabs, you would need code like this:

```
<f:facet name="menu1">
  <af:menuTabs>
    <af:commandMenuItem text="Benefits" selected="true"
                        action="go.benefits"/>
    <af:commandMenuItem text="Employee Data" action="go.emps"/>
  </af:menuTabs>
</f:facet>
<f:facet name="menu2">
  <af:menuBar>
    <af:commandMenuItem text="Insurance" selected="true"
                        action="go.insurance"/>
    <af:commandMenuItem text="Paid Time Off" selected="false"
                        action="go.pto"/>
  </af:menuBar>
</f:facet>
```

- Bind the menu components to a `MenuModel` object, and for each menu component use a `nodeStamp` facet to stamp out the menu items (which does not require having multiple menu item components in each menu component). By binding to a `MenuModel` object and using a `nodeStamp` facet, you use less code in your JSF pages, and almost any page (regardless of its place in the hierarchy) can be rendered using the same menu code. For example, to create the same two menu tabs shown earlier:

```
<f:facet name="menu1">
  <af:menuTabs var="menutab" value="#{menuModel.model}">
    <f:facet name="nodeStamp">
      <af:commandMenuItem text="#{menutab.label}"
                          action="#{menutab.getOutcome}"/>
    </f:facet>
  </af:menuList>
</f:facet>
<f:facet name="menu2">
  <af:menuBar startDepth="1" var="menusubtab" value="#{menuModel.model}">
    <f:facet name="nodeStamp">
      <af:commandMenuItem text="#{menusubtab.label}"
                          action="#{menusubtab.getOutcome}"/>
    </f:facet>
  </af:menuList>
</f:facet>
```

  In the SRDemo pages, the menu components are bound to a menu model object that is configured via managed beans. For information about how to create a menu structure using managed beans, see Section 11.2, "Using Dynamic Menus for Navigation".

In addition to laying out hierarchical menus, the `panelPage` component supports other facets for laying out page-level and application-level text, images, and action buttons in specific areas, as illustrated in Figure 4–7 and Figure 4–8.

For instructions on how to insert child components into facets or into `panelPage` itself, see Section 4.4.1, "How to Add UI Components to a JSF Page".

4.4.4.1 PanelPage Facets

Figure 4–7 shows `panelPage` facets (numbered 1 to 12) for laying out branding images, global buttons, menu tabs, bars, and lists, and application-level text.

Figure 4–7 Basic Page Layout with Branding Images, Navigation Menus, and Application-Level Text

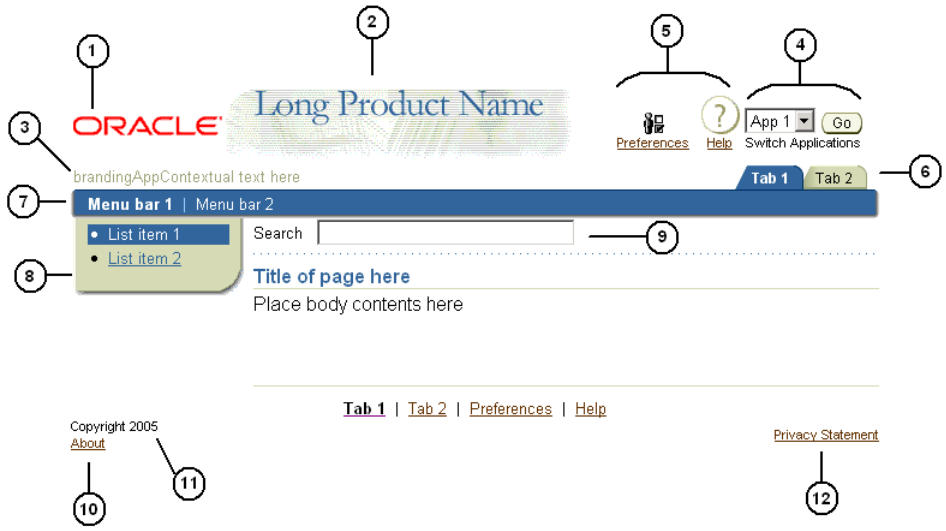


Table 4–3 shows the `panelPage` facets (as numbered in Figure 4–7), and the preferred children components that you could use in them. In JDeveloper, when you right-click a facet in the Structure window, the **Insert inside** context menu shows the preferred component to use, if any.

Table 4–3 PanelPage Facets for Branding Images, Navigation Menus, and Application-Level Text

No.	Facet	Description
1	`branding`	For a corporate logo or organization branding using `objectImage`. Renders its child component at the top left corner of the page.
2	`brandingApp`	For an application logo or product branding using `objectImage`. Renders its child component after a `branding` image, if used. If `chromeType` on `panelPage` is set to "expanded", the `brandingApp` image is placed below the `branding` image.
3	`brandingAppContextual`	Typically use with `outputFormatted` text to show the application's current branding context. Set the `styleUsage` attribute on `outputFormatted` to `inContextBranding`.
4	`menuSwitch`	For a `menuChoice` component that allows the user to switch to another application from any active page. Renders its child component at the top right corner of the page. The `menuChoice` component can be bound to a menu model object.

Table 4–3 (Cont.) PanelPage Facets for Branding Images, Navigation Menus, and Application-Level Text

No.	Facet	Description
5	`menuGlobal`	For a `menuButtons` component that lays out a series of menu items as global buttons. Global buttons are buttons that are always available from any active page in the application (for example a Help button). Renders its children components at the top right corner of the page, before a `menuSwitch` child if used. A text link version of a global button is automatically repeated at the bottom of the page. The `menuButtons` component can be bound to a menu model object.
6	`menu1`	For a `menuTabs` component that lays out a series of menu items as tabs. Renders its children components (right justified) at the top of the page, beneath any branding images, menu buttons, or menu switch. A text link version of a tab is automatically repeated at the bottom of the page. Menu tab text links are rendered before the text link versions of global buttons. Both types of text links are centered in the page. The `menuTabs` component can be bound to a menu model object.
7	`menu2`	For a `menuBar` component that lays out a series of menu items in a horizontal bar, beneath the menu tabs. The children components are left justified in the bar, and separated by vertical lines. The `menuBar` component can be bound to a menu model object.
8	`menu3`	For a `menuList` component that lays out a bulleted list of menu items. Renders the children components in an area offset by color on the left side of a page, beneath a menu bar. The `menuList` component can be bound to a menu model object.
9	`search`	For a search area using an `inputText` component. Renders its child component beneath the horizontal menu bar. A dotted line separates it from the page title below.
10	`appAbout`	For a link to more information about the application using `commandLink`. The link text appears at the bottom left corner of the page.
11	`appCopyright`	For copyright text using outputText. The text appears above the `appAbout` link.
12	`appPrivacy`	For a link to a privacy policy statement for the application using `commandLink`. The link text appears at the bottom right corner of the page.

Tip: Many UI components support facets, not only `panelPage`. To quickly add or remove facets on a component, right-click the component in the Structure window and choose **Facets - <component name>**, where `<component name>` is the name of the UI component. If the component supports facets, you'll see a list of facet names. A checkmark next to a name means the `f:facet` element for that facet is already inserted in the page, but it may or not contain a child component.

Figure 4–8 shows `panelPage` facets (numbered 1 to 7) for laying out page-level actions and text.

Figure 4–8 Basic Page Layout with Page-Level Actions and Informational Text

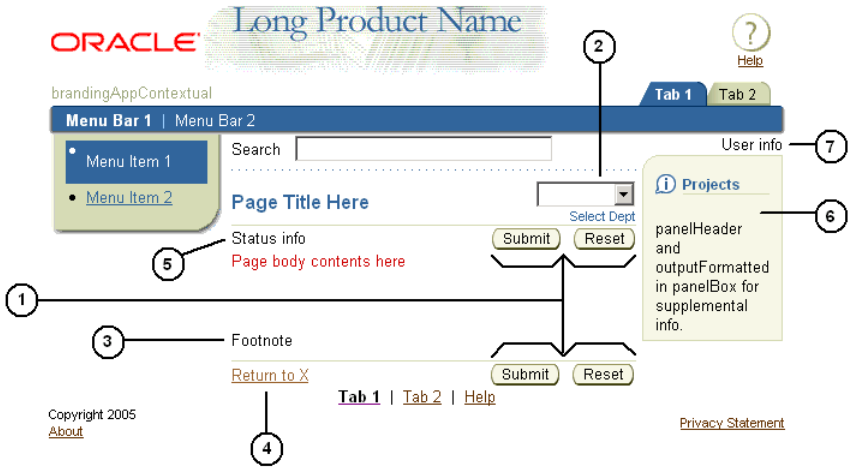


Table 4–4 shows the `panelPage` facets (as numbered in Figure 4–8), and the preferred children components that you could use in them.

Table 4–4 PanelPage Facets for Page-Level Actions and Informational Text

No.	Facet	Description
1	`actions`	For page-level actions that operate on the page content. Typically use with a `panelButtonBar` to lay out a series of buttons, a `processChoiceBar`, or a `selectOneChoice`. Renders its children components below the page title, right-justified. The children components are also automatically repeated near the bottom of the page (above any text link versions of menu tabs and global buttons) on certain devices and skins.
2	`contextSwitcher`	A context switcher lets the user change the contents of the page based on the context. For example, when a user is viewing company assets for a department, the user can use the context switcher to switch to the assets of another department. All the pages will then change to the selected context. Typically use with a `selectOneChoice` component. The facet renders its child component on the same level as the page title, right-justified.
3	`infoFootnote`	For page-level information that is ancillary to the task at hand. Typically use with an `outputFormatted` component, with `styleClass` or `styleUsage` set to an appropriate value. The facet renders its child component near the bottom of the page, left-justified and above the `infoReturn` link.
4	`infoReturn`	For a "Return to X" link using `commandLink`. For the user to move quickly back to the default page of the active menu tab. The facet renders its child component near the bottom of the page, left-justified and above the text link versions of menu tabs and global buttons.

Table 4–4 (Cont.) PanelPage Facets for Page-Level Actions and Informational Text

No.	Facet	Description
5	`infoStatus`	For page-level status information about the task at hand. Could also use to provide a key notation. A key notation is a legend used to define icons, elements, or terms used within the page contents. Typically use with an `outputFormatted` component, with `styleClass` or `styleUsage` set to an appropriate value. The facet renders its child component below the page title, left-justified.
6	`infoSupplemental`	For any other additional information. Typically use with a `panelBox` to show the information in an area offset by color. In the `panelBox` you could use for example `panelList` or `outputFormatted` text to provide additional information that might help the user, but is not required for completing a task. The facet renders its children components on the right side of the page, below the `infoUser` facet child component.
7	`infoUser`	For presenting user login and connection information. Typically use with an `outputFormatted` component, with `styleClass` or `styleUsage` set to an appropriate value. The facet renders its child component on the right side of the page, immediately below the menu bars.

Tip: Like `panelPage`, the `page` component also lets you lay out an entire page with specific content areas. Unlike `panelPage`, you can bind the `value` of `page` to a menu model object to create the page's hierarchical menus—you don't have to bind individual menu components to a menu model object.

4.4.4.2 Page Body Contents

After you've set up the `panelPage` facets, create your forms, tables, and other page body contents inside the `panelPage` component. ADF Faces panel components (and others) help you to organize content on a page. Use Table 4–5 to decide which components are suitable for your purposes.

For information about the component attributes you can set on each component, see the JDeveloper online help. For an image of what each component looks like, see the ADF Faces Core tag document at

`http://www.oracle.com/technology/products/jdev/htdocs/partners/addins/exchange/jsf/doc/tagdoc/core/imageIndex.html`

Table 4–5 ADF Faces Layout and Panel Components

To...	Use these components...
Align form input components in one or more columns, with the labels right-justified and the fields left-justified	`panelForm`
Arrange components horizontally, optionally specifying a horizontal or vertical alignment	`panelHorizontal`
Arrange components consecutively with wrapping as needed, horizontally in a single line, or vertically	`panelGroup`
Create a bulleted list in one or more columns	`panelList`

Table 4–5 (Cont.) ADF Faces Layout and Panel Components

To...	Use these components...
Lay out one or more components with a label, tip, and message	`panelLabelAndMessage` Place multiple `panelLabelAndMessage` components in a `panelForm` When laying out input component, the `simple` attribute on the input component must be set to `true`.
Place components in a container offset by color	`panelBox` Typically use a single child inside `panelBox` such as `panelGroup` or `panelForm`, which then contains the components for display
Place components in predefined locations using facets	`panelBorder`
Lay out a series of buttons	`panelButtonBar`
Display additional page-level or section-level hints to the user	`panelTip`
Create page sections and subsections with headers	`panelHeader`, `showDetailHeader`
Add quick links to sections in long pages	Set the `quickLinksShown` attribute on `panelPage` to `true`
Let the user toggle a group of components between being shown (disclosed) and hidden (undisclosed)	`showDetail`
Let the user select and display a group of contents at a time	A ShowOne component with `showDetailItem` components ShowOne components include `showOneTab`, `showOneChoice`, `showOneRadio`, and `showOnePanel`
Insert separator lines or space in your layout	`objectSeparator`, `objectSpacer`

4.5 Creating and Using a Backing Bean for a Web Page

In JSF, backing beans are JavaBeans used mainly to provide UI logic and to manage data between the web tier and the business tier of the application (similar to a data transfer object). Typically you have one backing bean per JSF page. The backing bean contains the logic and properties for the UI components used on the page. For example, to programmatically change a UI component as a result of some user activity or to execute code before or after an ADF declarative action method, you provide the necessary code in the page's backing bean and bind the component to the corresponding property or method in the bean.

For a backing bean to be available when the application starts, you register it as a managed bean with a name and scope in `faces-config.xml`. At runtime, whenever the managed bean is referenced on a page through a JSF EL value or method binding expression, the JSF implementation automatically instantiates the bean, populates it with any declared, default values, and places it in the managed bean scope as defined in `faces-config.xml`.

4.5.1 How to Create and Configure a Backing Bean

The Overview mode of the JSF Configuration Editor lets you create and configure a backing bean declaratively. Suppose you have a JSF page with the filename `SRDemopage.jspx`. Now you want to create a backing bean for the page.

To create and configure a backing bean as a managed bean:

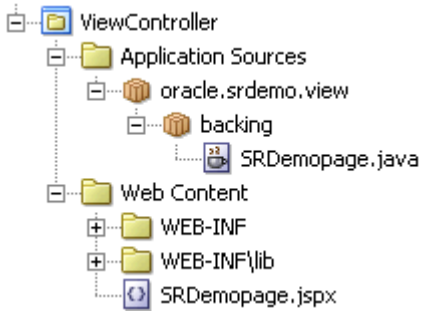
1. In the Application Navigator, double-click **faces-config.xml** to open it in the default mode of the JSF Configuration Editor.
2. At the bottom of the editor, select the **Overview** tab to switch to the Overview mode, if necessary.
3. In the element list on the left, select **Managed Beans**.
4. Click **New** to open the Create Managed Bean dialog.
5. In the dialog, specify the following for a managed bean:
   - **Name**: Enter a unique identifier for the managed bean (e.g., `backing_SRDemopage`). This identifier determines how the bean will be referred to within the application using EL expressions, instead of using the bean's fully-qualified class name.
   - **Class**: Enter the fully qualified class name (e.g., `oracle.srdemo.view.backing.SRDemopage`). This is the JavaBean that contains the properties that hold the data for the UI components used on the page, along with the corresponding accessor methods and any other methods (such as navigation or validation). This can be an existing or a new class.
   - **Scope**: This determines the scope within which the bean is stored. The valid scope values are:
     - **application**: The bean is available for the duration of the web application. This is helpful for global beans such as LDAP directories.
     - **request**: The bean is available from the time it is instantiated until a response is sent back to the client. This is usually the life of the current page. Backing beans for pages usually use this scope.
     - **session**: The bean is available to the client throughout the client's session.
     - **none**: The bean is instantiated each time it is referenced.
6. Select the **Generate Class If It Does Not Exist** checkbox to let JDeveloper create the Java class for you. If you've already created the Java class, don't select this checkbox.

Note: At this point, you haven't defined a strict relationship between the JSF page and the backing bean. You've simply configured a backing bean in `faces-config.xml`, which you can now reference via JSF EL expressions on a page. To define a strict relationship between a page and a backing bean, see Section 4.5.3, "How to Use a Backing Bean in a JSF Page".

4.5.2 What Happens When You Create and Configure a Backing Bean

If you select the **Generate Class If It Does Not Exist** checkbox, JDeveloper creates a new Java class using the fully qualified class name set as the value of **Class**. The new file appears within the **Application Sources** node of the **ViewController** project in the Application Navigator, as illustrated in Figure 4–9.

Figure 4–9 Backing Bean for SRDemopage.jspx in the Navigator



To edit the backing bean class, double-click the file in the Application Navigator (for example, **SRDemopage.java**) to open it in the source editor. If it's a new class, you would see something similar to Example 4–11.

Example 4–11 Empty Java Class Created by JDeveloper

```
package oracle.srdemo.view.backing;

public class SRDemopage {
    public SRDemopage() {
    }
}
```

In `faces-config.xml`, JDeveloper adds the backing bean configuration using the `<managed-bean>` element, as shown in Example 4–12.

Example 4–12 Registering a Managed Bean in the faces-config.xml File

```
<faces-config xmlns="http://java.sun.com/JSF/Configuration">
  ...
  <!-- Page backing beans typically use request scope-->
  <managed-bean>
    <managed-bean-name>backing_SRDemopage</managed-bean-name>
    <managed-bean-class>oracle.srdemo.view.backing.SRDemopage</managed-bean-class>
    <managed-bean-scope>request</managed-bean-scope>
  </managed-bean>
  ...
</faces-config>
```

> **Note:** For a backing bean to access the ADF Model binding layer at runtime, the backing bean could inject the ADF binding container. For information about how this is done, see Section 4.5.7, "Using ADF Data Controls and Backing Beans".

4.5.3 How to Use a Backing Bean in a JSF Page

Once a backing bean is defined with the relevant properties and methods, you use JSF EL expressions such as `#{someBean.someProperty}` or `#{someBean.someMethod}` to bind a UI component attribute to the appropriate property or method in the bean.

For example, the following code snippets illustrate value binding expressions and method binding expressions:

```
<af:inputText value="#{someBean.someProperty}"/>
```

```
..
<af:inputText disabled="#{someBean.anotherProperty}"/>
..
<af:commandButton action=#{someBean.someMethod}"/>
..
<af:inpuText valueChangeListener="#{someBean.anotherMethod}"/>
```

When such expressions are encountered at runtime, JSF instantiates the bean if it does not already exist in the bean scope that was configured in `faces-config.xml`.

In addition to value and method bindings, you can also bind the UI component's instance to a bean property using the `binding` attribute:

```
<af:commandButton binding="#{backing_SRDemopage.commandButton1}"
```

When the `binding` attribute of a UI component references a property in the bean, the bean has direct access to the component, and hence, logic in the bean can programmatically manipulate other attributes of the component, if needed. For example, you could change the color of displayed text, disable a button or field, or cause a component not to render, based on some UI logic in the backing bean.

To reiterate, you can bind a component's `value` attribute or any other attribute value to a bean property, or you can bind the component instance to a bean property. Which you choose depends on how much control you need over the component. When you bind a component attribute, the bean's associated property holds the value for the attribute, which can then be updated during the Update Model Values phase of the component's lifecycle. When you bind the component instance to a bean property, the property holds the value of the entire component instance, which means you can dynamically change any other component attribute value.

4.5.4 How to Use the Automatic Component Binding Feature

JDeveloper has a feature that lets you automatically bind a UI component instance on a JSF page to a backing bean property. When you turn on the Auto Bind feature for a page, for any UI component that you insert into the page, JDeveloper automatically adds property code in the page's backing bean, and binds the component's `binding` attribute to the corresponding property in the backing bean. If your backing bean doesn't have to modify the attributes of UI components on a page programmatically, you don't need to use the automatic component binding feature.

To turn on automatic component binding for a JSF page:

1. Open the JSF page in the visual editor. Select **Design** at the bottom of the editor window.
2. Choose **Design > Page Properties** to display the Page Properties dialog.
3. Select **Component Binding**.
4. Select **Auto Bind**.
5. Select a managed bean from the dropdown list or click **New** to configure a new managed bean for the page.

Note: By turning on automatic component binding in a JSF page, you are defining a strict relationship between a page and a backing bean in JDeveloper.

4.5.5 What Happens When You Use Automatic Component Binding in JDeveloper

If the Auto Bind feature is turned on for a JSF page, you'll see a special comment line near the end of the page:

```
...
   </f:view>
   <!--oracle-jdev-comment:auto-binding-backing-bean-name:backing_SRDemopage-->
</jsp:root>
```

In `faces-config.xml`, a similar comment line is inserted at the end of the page's backing bean configuration:

```
<managed-bean>
  <managed-bean-name>backing_SRDemopage</managed-bean-name>
  <managed-bean-class>oracle.srdemo.view.backing.SRDemopage</managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
  <!--oracle-jdev-comment:managed-bean-jsp-link:1SRDemopage.jspx-->
</managed-bean>
```

When you turn on the Auto Bind feature for a page, JDeveloper does the following for you every time you add a UI component to the page:

- Adds a property and property accessor methods for the component in the backing bean. For example, the next code snippet shows the code added for an `inputText` and a `commandButton` component:

  ```
  private CoreInputText inputText1;
  private CoreCommandButton commandButton1;
  public void setInputText1(CoreInputText inputText1) {
      this.inputText1 = inputText1;
  }

  public CoreInputText getInputText1() {
      return inputText1;
  }

  public void setCommandButton1(CoreCommandButton commandButton1) {
      this.commandButton1 = commandButton1;
  }

  public CoreCommandButton getCommandButton1() {
      return commandButton1;
  }
  ```

- Binds the component to the corresponding bean property using an EL expression as the value for the `binding` attribute, as shown in this code snippet:

  ```
  <af:inputText binding="#{backing_SRDemopage.inputText1}"
  <af:commandButton binding="#{backing_SRDemopage.commandButton1}"
  ```

When you turn off the Auto Bind feature for a page, JDeveloper removes the special comments from the JSF page and `faces-config.xml`. The `binding` EL expressions on the page and the associated backing bean code are not deleted.

> **Tip:** When Auto Bind is turned on and you delete a UI component from a page, JDeveloper automatically removes the corresponding property and accessor methods from the page's backing bean.

4.5.6 What You May Need to Know About Backing Beans and Managed Beans

Managed beans are any application JavaBeans that are registered in the JSF `faces-config.xml` file. *Backing beans* are managed beans that contain logic and properties for some or all UI components on a JSF page. If you place, for example, validation and event handling logic in a backing bean, then the code has programmatic access to the UI components on the page when the UI components are bound to properties in the backing bean via the `binding` attribute.

In this guide, the term *backing bean* might be used interchangeably with the term *managed bean*, because all backing beans are managed beans. You can, however, have a managed bean that is not a backing bean—that is, a JavaBean that does not have properties and property getter and setter methods for UI components on a page, but the bean is configured in `faces-config.xml`, and has code that is not specific to any single page. Examples of where managed beans that are not backing beans are used in the SRDemo application include beans to:

- Access authenticated user information from the container security
- Create the navigation menu system (such as menu tabs and menu bars).
- Expose String resources in a bundle via EL expressions

Managed bean properties are any properties of a bean that you would like populated with a value when the bean is instantiated. The set method for each declared property is run once the bean is constructed. To initialize a managed bean's properties with set values, use the `<managed-property>` element in `faces-config.xml`. When you configure a managed property for a managed bean, you declare the property name, its class type, and its default value, as shown in Example 4–13.

Example 4–13 Managed Bean Property Initialization Code in the faces-config.xml File

```
<managed-bean>
  <managed-bean-name>tax</managed-bean-name>
  <managed-bean-class>com.jsf.databeans.TaxRateBean</managed-bean-class>
  <managed-bean-scope>application</managed-bean-scope>
  <managed-property>
    <property-name>rate</property-name>
    <property-class>java.lang.Float</property-class>
    <value>5</value>
  </managed-property>
</managed-bean>
```

In Example 4–13, the `rate` property is initialized with a value of 5 (converted to a Float) when the bean is instantiated using the EL expression `#{tax.rate}`.

Managed beans and managed bean properties can be initialized as lists or maps, provided that the bean or property type is a List or Map, or implements `java.util.Map` or `java.util.List`. The default types for the values within a list or map is `java.lang.String`.

Example 4–14 shows an example of a managed bean that is a List.

Example 4–14 Managed Bean List in the faces-config.xml File

```
<managed-bean>
  <managed-bean-name>options</managed-bean-name>
  <managed-bean-class>java.util.ArrayList</managed-bean-class>
  <managed-bean-scope>application</managed-bean-scope>
  <list-entries>
    <value>Text Only</value>
```

```
    <value>Text + HTML</value>
    <value>HTML Only</value>
  </list-entries>
</managed-bean>
```

When the application encounters the EL expression `#{options.text}`, a List object is created and initialized with the values from the declared list-entries' values. The `managed-property` element is not declared, but the `list-entries` are child elements of the `managed-bean` element instead.

> **Tip:** Managed beans can only refer to managed properties in beans that have the same scope or a scope with a longer lifespan. For example a `session` scope bean cannot refer to a managed property on a `request` scoped bean.

4.5.7 Using ADF Data Controls and Backing Beans

When you create databound UI components by dropping items from the Data Control Palette on your JSF page, JDeveloper does many things for you, which are documented in Section 5.2.3, "What Happens When You Use the Data Control Palette". The databound UI components use ADF data binding EL expressions, such as `#{bindings.ProductName.inputValue}`, to reference the associated binding objects in the page's binding container, where `bindings` is the reference to the ADF binding container of the current page.

In the backing bean of a page that uses ADF data bindings, sometimes you might want to reference the binding container's binding objects. To reference the ADF binding container, you can resolve a JSF value binding to the `#{bindings}` EL expression and cast the result to an `oracle.binding.BindingContainer` interface. Or for convenience, you can add a managed property named `bindings` that references the same `#{bindings}` EL expression, to the page's managed bean configuration in `faces-config.xml` so that the backing bean can work programmatically with the ADF binding container at runtime. Example 4–15 shows the `bindings` managed property in the `backing_SRMain` managed bean for the SRMain page.

Example 4–15 Bindings Managed Property in the faces-config.xml File

```
<faces-config xmlns="http://java.sun.com/JSF/Configuration">
  ...
  <managed-bean>
    <managed-bean-name>backing_SRMain</managed-bean-name>
    <managed-bean-class>oracle.srdemo.view.backing.SRMain</managed-bean-class>
    <managed-bean-scope>request</managed-bean-scope>
    <managed-property>
      <property-name>bindings</property-name>
      <value>#{bindings}</value>
    </managed-property>
  </managed-bean>
  ...
</faces-config>
```

In the backing bean, add the getter and setter methods for the binding container. Example 4–16 shows the part of `SRMain.java` that contains the relevant code for `bindings`.

Example 4–16 Bindings Getter and Setter Methods in a Backing Bean

```
...
import oracle.binding.BindingContainer;
```

```
    private BindingContainer bindings;

    public BindingContainer getBindings() {
        return this.bindings;
    }

    public void setBindings(BindingContainer bindings) {
        this.bindings = bindings;
    }

...
```

At runtime, when the application encounters an ADF data binding EL expression that refers to the ADF binding container, such as `#{bindings.bindingObject.propertyName}`, JSF evaluates the expression and gets the value from the binding object.

For more information about ADF data binding EL expressions and ADF binding properties, see Section 5.6, "Creating ADF Data Binding EL Expressions".

For an overview of how JSF backing beans work with the ADF Model layer, see Chapter 1, "Introduction to Oracle ADF Applications".

4.6 Best Practices for ADF Faces

Consider the following best practices when developing with ADF Faces:

- While both JSP documents (`.jspx`) and JSP pages (`.jsp`) can be used, Oracle recommends working with JSP documents (`.jspx`) when using ADF Faces components in your JSF pages because JSP documents are well-formed XML documents. The XML standard offers many benefits such as validating against a document type definition, and parsing to create documentation or audit reports.
- Use token-based client-side state saving instead of server-side state saving by setting the value of `javax.faces.STATE_SAVING_METHOD` in `web.xml` to `client` (which matches the default server-side behavior that will be provided in JSF 1.2).

  While server-side state saving can provide somewhat better performance, client-side state saving is recommended as it provides better support for failover and the back button, and for displaying multiple windows simultaneously. Token-based client-side state saving results in better server performance because CPU and I/O consumption is lower.

  Note that `javax.faces.STATE_SAVING_METHOD` must be set to `server` for ADF Telnet applications because the Industrial Telnet Server does not currently support saving state on the client.
- Remove or disable debug features to improve the performance of deployed applications:
  - In `web.xml`, disable `oracle.adf.view.faces.CHECK_FILE_MODIFICATION`. By default, this parameter is `false`. If it is set to `true`, ADF Faces automatically checks the modification date of your JSPs, and discards saved state when they change. For testing and debugging in JDeveloper's embedded OC4J, you don't need to explicitly set this parameter to `true` because ADF Faces automatically detects the embedded OC4J and runs with the file modification checks enabled. But when you deploy the application, you should set the parameter to `false`.

For testing and debugging in JDeveloper's embedded OC4J, you don't need to explicitly set this parameter to `true` because ADF Faces automatically detects the embedded OC4J and runs with the file modification checks enabled.

  - In `web.xml`, disable `oracle.adf.view.faces.DEBUG_JAVASCRIPT`. The default value of this parameter is `false`. This means that by default, ADF Faces obfuscates JavaScript and removes comments and whitespace to reduce the size of the JavaScript download to the client. During application development, you might set the parameter to `true` (to turn off obfuscation) so that you can debug JavaScript easier, but when you deploy the application, you should set the parameter to `false`.
  - In `adf-faces-config.xml`, set `<debug-output>` to `false`. ADF Faces enhances debugging output when `<debug-output>` is `true`, by adding automatic indenting and extra comments, and detecting for malformed markup problems, unbalanced elements, and common HTML errors. The enhanced debug output is not necessary in deployed applications.

- ADF Faces input components provide support for automatic form submission via the `autoSubmit` attribute. When the `autoSubmit` attribute is set to `true`, and an appropriate action takes place (such as a value change), the input component automatically submits the form it is enclosed in through a partial page submit. Thus you can update a portion of a page without having to redraw the entire page, which is known as *partial page rendering*. For information about using partial page rendering, see Section 11.4, "Enabling Partial Page Rendering".
- ADF Faces performs client-side and server-side validation upon an auto submit execution. But if both `autoSubmit` and `immediate` attributes on ADF Faces input components are set to `true`, then ADF Faces doesn't perform client-side validation.
- When laying out ADF Faces input components inside `panelLabelAndMessage` components, you must set the `simple` attributes on the input components to `true`. For accessibility purposes, set the `for` attribute on `panelLabelAndMessage` to the first input component. For proper alignment, place multiple `panelLabelAndMessage` components in a `panelForm`.
- Although ADF Faces ignores label and message attributes on "simple" input components, you must set the `label` attribute on a "simple" component in this version of ADF Faces for component-generated error messages to display correctly.
- If both `styleClass` and `styleUsage` attributes are set on a component, `styleClass` has precedence over `styleUsage`.
- ADF Faces provides three levels of page accessibility support, which is configured in `adf-faces-config.xml` using the `<accessibility-mode>` element. The acceptable values for `<accessibility-mode>` are:
  - `default`: By default ADF Faces generates HTML code that is accessible to disabled users.
  - `screenReader`: ADF Faces generates HTML code that is optimized for the use of screen readers. The `screenReader` mode facilitates disabled users, but it may degrade the output for regular users. For example, access keys are disabled in screen reader mode.
  - `inaccessible`: ADF Faces removes all code that does not affect sighted users. This optimization reduces the size of the generated HTML. The application, however, is no longer accessible to disabled users.

- Images that are automatically generated by ADF Faces components have built-in descriptions that can be read by screen readers or nonvisual browsers. For images generated from user-supplied icons and images, make sure you set the `shortDesc` or `searchDesc` attribute. Those attributes transform into HTML `alt` attributes. For images produced by certain ADF Faces components such as `menuTabs` and `menuButtons`, make sure you set the `text` or `icon` attribute on `commandMenuItem` because ADF Faces uses those values to generate text that describes the menu name as well as its state.

  Similarly for `table` and `outputText` components, set the `summary` and `description` attribute, respectively, for user agents rendering to nonvisual media. If you use frames, provide links to alternative pages without frames using the `alternateContent` facet on `frameBorderLayout`. Within each frame set the `shortDesc` and `longDescURL` attributes.

- Specify an access key for input, command, and go components such as `inputText`, `commandButton`, and `goLink`.

  – Typically, you use the component's `accessKey` attribute to set a keyboard character. For command and go components, the character specified by the attribute must exist in the `text` attribute of the component instance. If it does not exist, ADF Faces does not display the visual indication that the component has an access key

  – You can also use `labelAndAccessKey` on input components, or `textAndAccessKey` on command and go components. Those attributes let you set the `label` or `text` value, and an access key for the component at the same time. The conventional ampersand notation to use is `&` in JSP documents (`.jspx`). For example, in this code snippet:

    ```
    <af:commandButton textAndAccessKey="&amp;Home"/>
    ```

    ... the button text is `Home` and the access key is `H`, the letter that is immediately after the ampersand character.

  – Using access keys on `goButton` and `goLink` components may immediately activate them in some browsers. Depending on the browser, if the same access key is assigned to two or more go components on a page, the browser may activate the first component instead of cycling among the components that are accessed by the same key.

  – If you use a space as the access key, you need to provide a way to tell the user that Alt+Space or Alt+Spacebar is the access key because there is no good way to present a blank or space visually in the component's label or textual label. For example, you could provide some text in a component tooltip using the `shortDesc` attribute.

  – Access keys are not displayed if the accessibility mode is set to screen reader mode.

- Enable application view caching by setting the value of `oracle.adf.view.faces.USE_APPLICATION_VIEW_CACHE` in `web.xml` to `true`.

  When application view caching is enabled, the first time a page is viewed by any user, ADF Faces caches the initial page state at an application level. Subsequently, all users can reuse the page's cached state coming and going, significantly improving application performance.

  While application view caching can improve a deployed application's performance, it is difficult to use during development and there are some coding

issues that should be considered. For more detailed information about using application view caching, see "Configuring ADF Faces for Performance" in the "Configuring ADF Faces" section of the *ADF Faces Developer's Guide*.

- For ADF Faces deployment best practices, see Chapter 22, "Deploying ADF Applications".
- Increase throughput and shorten response times by caching content with the ADF Faces Cache tag library. Caching stores all or parts of a web page in memory for use in future responses. It significantly reduces response time to client requests by reusing cached content for future requests without executing the code that created it. For more information, see Chapter 15, "Optimizing Application Performance with Caching".

5 Displaying Data on a Page

This chapter describes how to use the Data Control Palette to create databound UI components that display data on a page. It also describes how to work with all the objects that are created when you use the Data Control Palette.

This chapter includes the following sections:

- Section 5.1, "Introduction to Displaying Data on a Page"
- Section 5.2, "Using the Data Control Palette"
- Section 5.3, "Working with the DataBindings.cpx File"
- Section 5.4, "Configuring the ADF Binding Filter"
- Section 5.5, "Working with Page Definition Files"
- Section 5.6, "Creating ADF Data Binding EL Expressions"

The remaining chapters in this part of this guide describe how to create specific types of pages using databound components.

5.1 Introduction to Displaying Data on a Page

The ADF data controls, which are described in Section 3.10, "Exposing Services with ADF Data Controls", provide an abstraction of an application's business services, giving the ADF binding layer access to the service data. Data controls define the data model returned by the business service. You can bind UI components to data controls to populate a page with data from your data model at runtime.

The JDeveloper Data Control Palette exposes an application's data controls in the IDE and enables you to use drag and drop to create a variety of UI components on a JSF page. The UI components created by the Data Control Palette use declarative data binding, which means that the data binding expressions are automatically configured and that, in most cases, you do not have to write any additional code.

The advantages of binding to ADF data controls, instead of binding to the JavaServer Faces standard managed beans, include:

- Declarative data binding using drag and drop from the Data Control Palette that requires little to no additional coding.
- A uniform (standards-based) approach to UI data binding for multiple UI technologies

Read this chapter to understand:

- How to use the Data Control Palette to create databound UI components
- The items that appear on the Data Control Palette
- The objects that JDeveloper creates for you when you use the Data Control Palette
- How to construct an ADF data binding EL expression
- The content of the page definition file and its relationship to EL expressions

5.2 Using the Data Control Palette

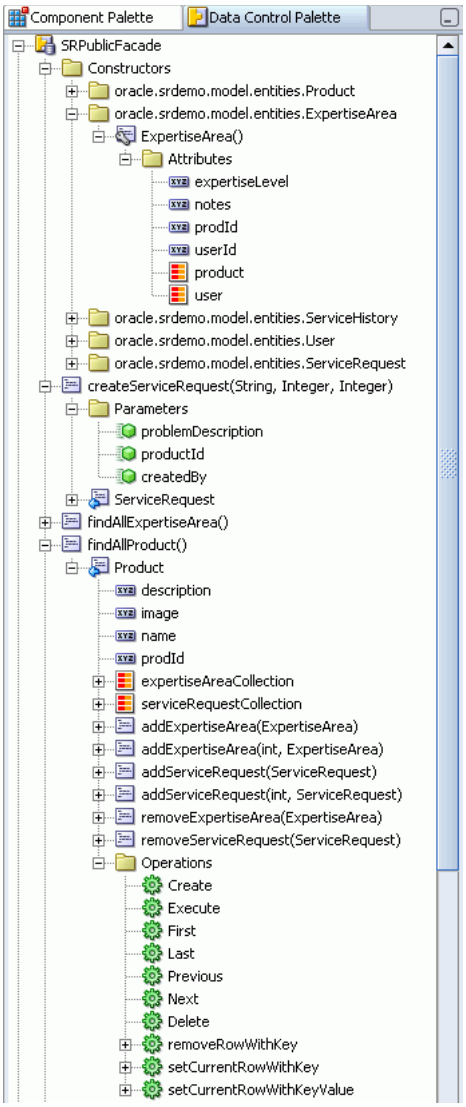
You can design a databound user interface by dragging an item from the Data Control Palette and dropping it on a page as a specific UI component. When you use the Data Control Palette to create a UI component, JDeveloper automatically creates the various code and objects needed to bind the component to the data control you selected.

To display the Data Control Palette, open a JSF page in the Design page of the visual editor and choose **View** > **Data Control Palette**. By default, JDeveloper displays the Data Control Palette in the same window as the Component Palette.

Figure 5–1 shows the Data Control Palette for the SRDemo application, which uses TopLink as the business service and data controls created from EJB session facades.

Note: If no data controls have been created for the application's business services, the Data Control Palette will be empty. For information about creating data controls, see Chapter 3, "Building and Using Application Services".

Figure 5–1 Data Control Palette



5.2.1 How to Understand the Items on the Data Control Palette

The Data Control Palette shows all the data controls that have been created for the application's business services and exposes all the data objects, data collections, methods, and built-in operations that are available for binding to UI components. A *data collection* represents a set of data objects (also known as a *rowset*) in the data model. Each object in a data collection represents a specific structured data item (also known as a *row*) in the data model.

Each root node in the Data Control Palette represents a specific data control. Under each data control is a hierarchical list of objects, collections, methods, and operations. How this hierarchy appears on the Data Control Palette depends on the type of business service represented by the data control and how the business services were defined.

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

Table 5–1 The Data Control Palette 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, each representing a logical grouping of data functions.	Not used to create anything. Serves as a container for the other objects.
	Create Method	Represents a built-in method that creates a new instance of an object in a data collection. Create method icons are located in a folder named after the data collection to which they belong. These data collection folders are located in the Constructors folder under the data control. The Attributes folder, which appears as a child under a create method, contains all the attributes of the data collection in which the object will be created. If the objects in a collection contain an attribute from another collection (called a *foreign key* in relational databases), that attribute is represented by an accessor return icon. I this case, the accessor returns a single value and has no children. For more information about using constructors, see Section 10.7, "Creating an Input Form for a New Record".	Creation forms
	Method	Represents a custom method on the data control that may accept parameters, perform some action or business logic, and return single values or data collections. If the method is a `get` method of a `Map` and returns a value or a collection, a method return icon appears as a child under it. If a method requires a parameter, a folder appears under the method, which lists the required parameters. For more information about using methods that accept parameters, see Section 10.6, "Creating a Form or Table Using a Method that Takes Parameters".	Command components For methods that accept parameters: command components and parameterized forms
	Method Return	Represents an object that is returned by a custom method. The returned object may be a single value or a collection. A method return appears as a child under the method that returns it. The objects that appear as children under a method return may be attributes of the collection, accessor returns that represent collections related to the parent collection, other methods that perform actions related to the parent collection, and operations that can be performed on the parent collection.	For collections: forms, tables, trees, and range navigation components For single values: text fields and selection lists
	Accessor Return	Represents an object returned by an accessor method on the business service. An accessor method is used when the objects returned are JavaBeans. Accessor returns appear as children under method returns, other accessor returns, or in the Attributes folder under built-in create methods. Accessor returns are objects that are related to the current object in the parent collection. This relationship is usually based on a common unique attribute in both objects. For example, if a method returns a collection of users, an accessor return that is a child of that collection might be a collection of service requests that are assigned to a particular user. In ADF, the relationship between parent and child collections is called a master-detail relationship. For more information about accessor returns and master-detail objects, see Chapter 8, "Displaying Master-Detail Data". The children under an accessor return may be attributes of the object, other accessor returns, custom methods that return a value from the accessor return, and operations that can be performed on the accessor return.	For collections: forms, tables, trees, range navigation components, and master-detail widgets For single objects: forms, master-detail widgets, and selection lists For single objects under a constructor: selection lists only

Table 5–1 (Cont.) The Data Control Palette Icons and Object Hierarchy

Icon	Name	Description	Used to Create...
	Attribute	Represents a discrete data element in an object. Attributes appear as children under the method returns or accessor returns to which they belong.	Label, text field, and selection list components.
	Operation	Represents a built-in data control operation that performs actions on the parent object. Data control operations are located in an Operations folder under method returns or accessor returns and under the root data control node. The operations that are children of a particular method or accessor return operate on that return object only, while operations under the data control node operate on all the objects represented by the data control. If an operation requires one or more parameters, they are listed in a Parameters folder under the operation.	UI actions such as buttons or links.
	Parameter	Represents a parameter value that is declared by the method or operation under which it appears. Parameters appear in the Parameters folder under a method or operation.	Label, text, and selection list components.

5.2.2 How to Use the Data Control Palette

To create a databound UI component, drag an item from the Data Control Palette and drop it on a JSF page.

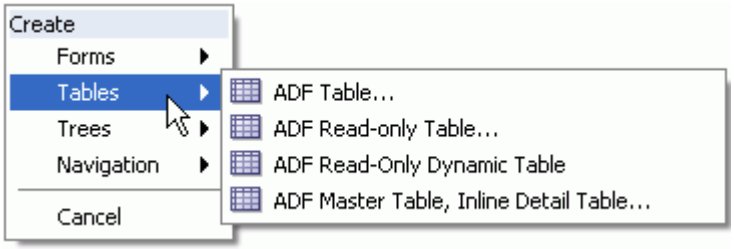
When you drag an item from the Data Control Palette and drop it on a page, JDeveloper displays a context menu of all the default UI components available for the item you dropped. From the context menu, select the component you want to create.

Note: When you drag and drop a built-in create method from the Data Control Palette, the context menu does not appear. This is because create methods can be used to create only one type of component—creation forms.

Figure 5–2 shows the context menu displayed when a method return from the Data Control Palette is dropped on a page.

Tip: By default, the Data Control Palette context menu displays only ADF Faces components. However, you can use equivalent JSF components instead. To have JSF components appear in the Data Control Palette context menu, select the **Include JSF HTML Widgets for JSF Databinding** option in the ADF View Settings page of the project properties. However, using ADF Faces components, especially with ADF bindings, provide greater functionality than JSF components.

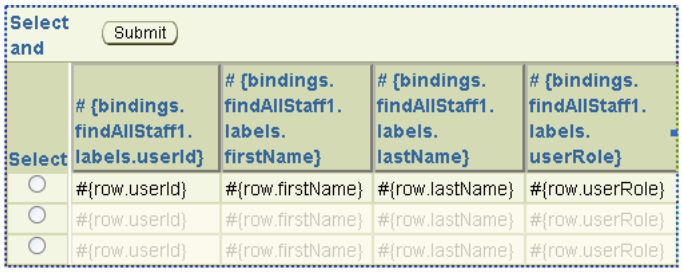
Figure 5–2 Data Control Palette Context Menu



Depending on the component you select from the context menu, JDeveloper may display a dialog that enables you to define how you want the component to look. For example, if you select **ADF Read-only Table** from the context menu, the Edit Table Columns dialog appears. This dialog enables you to define which attributes you want to display in the table columns, the column labels, what types of text fields you want use for each column, and what functionality you want to include, such as selection facets or column sorting. (For more information about creating tables, see Chapter 7, "Adding Tables".)

The resulting UI component appears in the JDeveloper visual editor. For example, if you drag a method return from the Data Control Palette, and choose **ADF Read-only Table** from the context menu, a read-only table appears in the visual editor, as shown in Figure 5–3.

Figure 5–3 Databound UI Component: ADF Read-only Table



Notice that the column labels in the sample table contain binding expressions, which bind each column label to an attribute in the data collection. The default table includes a selection facet, which is bound to a data collection iterator through an iterator binding. The selection facet was included by selecting the **Enable Selection** option in the Edit Table Columns dialog, which appears after you drop the table component. (Binding expressions are discussed later in Section 5.6, "Creating ADF Data Binding EL Expressions".)

By default, the UI components created when you use the Data Control Palette use ADF Faces components, are bound to collections and collection attributes in the ADF data control, and may have one or more built-in features including:

- Databound labels
- Tooltips
- Formatting
- Basic navigation buttons
- Validation (if validation rules are attached to a particular attribute; see Chapter 12, "Using Validation and Conversion" for information about validation)

The default components are fully functional without any further modifications. However, you can modify them to suit your particular needs. Each component and its various features are discussed further in the remaining chapters in this part of this guide.

Tip: If you want to change the type of ADF databound component used on a page, the easiest method is to delete the component and drag and drop a new one from the Data Control Palette. When you delete a databound component from a page, if the related binding objects in the page definition file are not referenced by any other component, JDeveloper automatically deletes those binding objects for you.

5.2.3 What Happens When You Use the Data Control Palette

While an ADF web application is built using the JSF framework, it requires a few additional application object definitions to render and process a page containing ADF databound UI components. If you do not use the Data Control Palette, you will have to manually configure these various files yourself. However, when you use the Data Control Palette, JDeveloper does all the required steps for you, which are:

- Create a `DataBindings.cpx` file in the view package in the Application Sources directory (if one does not already exist), and add an entry for the page.

  The `DataBindings.cpx` file defines the binding context for the application. It maps individual pages to their corresponding page definition files and registers the data controls used by those pages. The data controls are defined in the `DataControls.dcx` file. For more information, see Section 5.3, "Working with the DataBindings.cpx File".

- Register the ADF binding filter in the `web.xml` file.

  The ADF binding filter preprocesses any HTTP requests that may require access to the binding context. For more information about the binding filter configuration, see Section 5.4, "Configuring the ADF Binding Filter".

- Register the ADF phase listener in the `faces-config.xml` file, as shown in Example 5–1.

Example 5–1 ADF Phase Listener Entry in the faces-config.xml File

```
<lifecycle>
   <phase-listener>oracle.adf.controller.faces.lifecycle.ADFPhaseListener
   </phase-listener>
</lifecycle>
```

  The ADF phase listener is used to execute the ADF page lifecycle. It listens for all the JSF phases before which and after which it needs to execute its own phases, which are concerned with preparing the model, validating model updates, and preparing pages to be rendered. For more information about the ADF lifecycle, see Section 6.2.3, "What Happens at Runtime: The JSF and ADF Lifecycles".

- Add the following ADF runtime libraries to the project properties of the view project:
  - ADF Model Runtime (`adfm.jar`)
  - ADF Controller (`adf-controller.jar`)
- Add a page definition file (if one does not already exist for the page) to the page definition subpackage, the name of which is defined in the ADFm settings of the project properties. The default subpackage is `view.pageDefs` in the `Application Sources` directory.

  The page definition file (`<pageName>PageDef.xml`) defines the ADF binding container for each page in an application's view layer. The binding container

provides runtime access to all the ADF binding objects. In later chapters, you will see how the page definition files are used to define and edit the binding object definitions for specific UI components. For more information about the page definition file, see Section 5.5, "Working with Page Definition Files".

- Configure the page definition file, which includes adding definitions of the binding objects referenced by the page.
- Add prebuilt components to the JSF page.

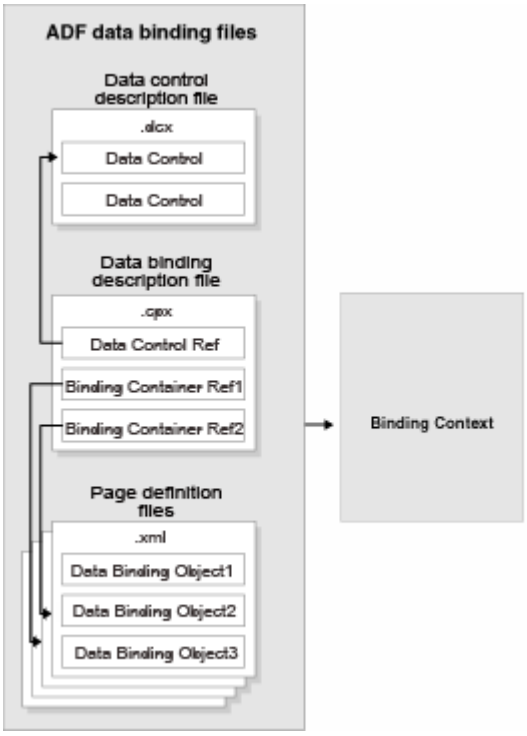
  These prebuilt components include ADF data binding expression language (EL) expressions that reference the binding objects in the page definition file. For more information, see Section 5.6, "Creating ADF Data Binding EL Expressions".
- Add all the libraries, files, and configuration elements required by ADF Faces components, if ADF Faces components are used. For more information, see Section 4.4.2, "What Happens When You First Insert an ADF Faces Component".

5.2.4 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 a single object known as the Oracle ADF binding context. The ADF binding context is a container object that contains a list of data controls and data binding objects derived from the Oracle ADF Model layer.

The ADF lifecycle creates the Oracle ADF binding context from the `DataControls.dcx`, `DataBindings.cpx`, and page definition files, as shown in Figure 5–4. The `DataControls.dcx` file defines all the data controls available to the application. The `DataBindings.cpx` file references the data controls that are currently being used by pages in the application and maps the binding containers, which contain the binding objects defined in the page definition files, to web page URLs. The page definition files define the binding objects used the application pages. There is one page definition for each page. For information about the ADF lifecycle, see Section 6.2.3, "What Happens at Runtime: The JSF and ADF Lifecycles".

Figure 5–4 ADF Binding File Runtime Usage



5.3 Working with the DataBindings.cpx File

The `DataBindings.cpx` file maps individual pages to page definition files and declares which data controls defined in the `DataControls.dcx` file are being used by the application. (For information about the `DataControls.dcx` file, see Section 3.10.2, "Understanding the Data Control Files".) The `DataBindings.cpx` file defines the Oracle ADF binding context for the entire application and provides the metadata from which the Oracle ADF binding objects are created at runtime.

5.3.1 How to Create a DataBindings.cpx File

The first time you use the Data Control Palette to add a component to a page in an application, JDeveloper automatically creates the `DataBindings.cpx` file in the view package of the `Application Sources` directory of the view project. Once the `DataBindings.cpx` file is created, JDeveloper adds an entry for the first page. Each subsequent time you use the Data Control Palette to add a component to a page, JDeveloper adds an entry to the `DataBindings.cpx` for that page, if one does not already exist.

CAUTION: If you change the name of a JSF page, a page definition file, or a data control, the `Databindings.cpx` file is *not* automatically refactored. You must manually update the page mapping in the `DataBindings.cpx` file.

5.3.2 What Happens When You Create a DataBindings.cpx File

Example 5–2 shows the `DataBindings.cpx` file for the SRDemo application. The `pageMap` element maps each JSF page to its corresponding page definition file. The `pageDefintionUsages` element identifies each page definition file in the application. The `dataControlUsages` element identifies the data controls being used by the binding objects defined in the page definition files. For more information about the elements and attributes in the `DataBindings.cpx` file, see Appendix A, "Reference ADF XML Files".

Example 5–2 DataBindings.cpx File

```
<?xml version="1.0" encoding="UTF-8" ?>
<Application xmlns="http://xmlns.oracle.com/adfm/application"
             version="10.1.3.35.65" id="DataBindings" SeparateXMLFiles="false"
             Package="oracle.srdemo.view" ClientType="Generic">
  <pageMap>
    <page path="/app/SRList.jspx" usageId="SRListPageDef"/>
    <page path="/app/SRCreate.jspx" usageId="SRCreatePageDef"/>
    <page path="/app/SRCreateConfirm.jspx" usageId="SRCreateConfirmPageDef"/>
    ...
  </pageMap>
  <pageDefinitionUsages>
    <page id="SRListPageDef"
          path="oracle.srdemo.view.pageDefs.app_SRListPageDef"/>
    <page id="UserInfoPageDef"
          path="oracle.srdemo.view.pageDefs.headless_UserInfoPageDef"/>
    <page id="SRCreatePageDef"
          path="oracle.srdemo.view.pageDefs.app_SRCreatePageDef"/>
    <page id="SRCreateConfirmPageDef"
          path="oracle.srdemo.view.pageDefs.app_SRCreateConfirmPageDef"/>
    ...
  </pageDefinitionUsages>
  <dataControlUsages>
    <dc id="EmailService" path="oracle.srdemo.emailService.EmailService"/>
    <dc id="SRDemoFAQ" path="oracle.srdemo.faq.SRDemoFAQ"/>
    <dc id="SRAdminFacade" path="oracle.srdemo.model.SRAdminFacade"/>
    <dc id="SRPublicFacade"
        path="oracle.srdemo.model.SRPublicFacade"/>
  </dataControlUsages>
</Application>
```

5.4 Configuring the ADF Binding Filter

The ADF binding filter is a servlet filter that is an instance of the `oracle.adf.model.servlet.ADFBindingFilter` class. ADF web applications use the ADF binding filter to preprocess any HTTP requests that may require access to the binding context.

5.4.1 How to Configure the ADF Binding Filter

The first time you add a databound component to a page using the Data Control Palette, JDeveloper automatically configures the filter for you in the application's `web.xml` file.

5.4.2 What Happens When You Configure an ADF Binding Filter

To configure the binding filter, JDeveloper adds the following elements to the `web.xml` file:

- A Servlet context parameter: Specifies which `DataBindings.cpx` file the binding filter reads at runtime to define the application binding context.

  The servlet context parameter is defined in the `web.xml` file, as shown in Example 5–3. The `param-name` element must contain the value `CpxFileName`, and the `param-value` element must contain the fully qualified name of the application's `DataBindings.cpx` file without the `.cpx` extension.

Example 5–3 Servlet Context Parameter Defined in the web.xml File

```
<context-param>
   <param-name>CpxFileName</param-name>
   <param-value>oracle.srdemo.view.DataBindings</param-value>
</context-param>
```

- An ADF binding filter class: Specifies the name of the binding filter object, which implements the `javax.servlet.Filter` interface.

  The ADF binding filter is defined in the `web.xml` file, as shown in Example 5–4. The `filter-name` element must contain the value `adfBindings`, and the `filter-class` element must contain the fully qualified name of the binding filter class, which is `oracle.adf.model.servlet.ADFBindingFilter`.

Example 5–4 Binding Filter Class Defined in the web.xml File

```
<filter>
   <filter-name>adfBindings</filter-name>
   <filter-class>oracle.adf.model.servlet.ADFBindingFilter</filter-class>
</filter>
```

- Filter mappings: Link filters to static resources or servlets in the web application.

  At runtime, when a mapped resource is requested, a filter is invoked. Filter mappings are defined in the `web.xml` file, as shown in Example 5–5. The `filter-name` element must contain the value `adfBindings`. Notice that in the example there is a filter mapping for both types of page formats: `jsp` and `jspx`.

Example 5–5 Filter Mapping Defined in the web.xml File

```
 <filter-mapping>
    <filter-name>adfBindings</filter-name>
    <url-pattern>*.jsp</url-pattern>
 </filter-mapping>
 <filter-mapping>
    <filter-name>adfBindings</filter-name>
    <url-pattern>*.jspx</url-pattern>
 </filter-mapping>
```

> **Tip:** If you have multiple filters defined in the `web.xml` file, be sure to list them in the order in which you want them to run. At runtime, the filters are executed in the sequence they appear in the `web.xml` file.

5.4.3 What Happens at Runtime

At runtime, the ADF binding filter performs the following functions:

- Overrides the character encoding when the filter is initialized with the name specified as a filter parameter in the `web.xml` file. The parameter name of the filter `init-param` element is `encoding`.
- Instantiates the `ADFContext` object, which is the execution context for an ADF application and contains context information about ADF, including the security context and the environment class that contains the request and response object.
- Initializes the binding context for a user's HTTP session.
- Serializes incoming HTTP requests from the same browser (for example, from framesets) to prevent multithreading problems.
- Notifies data control instances that they are about to receive a request, allowing them to do any necessary per-request setup.
- Notifies data control instances after the response has been sent to the client, allowing them to do any necessary per-request cleanup.

5.5 Working with Page Definition Files

Page definition files define the binding objects that populate the data in UI components at runtime. For every page that has ADF bindings, there must be a corresponding page definition file that defines the binding object used by that page. Page definition files provide design time access to all the ADF bindings. At runtime, the binding objects defined by a page definition file are instantiated in a binding container, which is the runtime instance of the page definition file.

5.5.1 How to Create a Page Definition File

The first time you use the Data Control Palette, JDeveloper automatically creates a page definition file for that page and adds definitions for each binding object referenced by the component. For each subsequent databound component you add to the page, JDeveloper automatically adds the necessary binding object definitions to the page definition file.

By default, the page definition files are located in the `view.PageDefs` package in the `Application Sources` directory of the view project. You can change the location of the page definition files using the ADFm Settings page of the project properties.

JDeveloper names the page definition files using the following convention:

<pageName>`PageDef.xml`

where *<pageName>* is the name of the JSF page. For example, if the JSF page is named `SRList.jsp`, the default page definition filename is `SRListPageDef.xml`. If you organize your pages into subdirectories, JDeveloper prefixes the directory name to the page definition filename using the following convention:

<directoryName>`_`*<pageName>*`PageDef.xml`

For example, in the SRDemo application, the name of the page definition file for the SRMain page, which is in the `app` subdirectory of the `Web Content` folder is `app_SRMainPageDef.xml`.

Caution: The `DataBindings.cpx` file maps JSF pages to their corresponding page definition files. If you change the name of a page definition file or a JSF page, JDeveloper does *not* automatically refactor the `DataBindings.cpx` file. You must manually update the page mapping in the `DataBindings.cpx` file.

To open a page definition file, right-click on the page in the visual editor or in the Application Navigator, and choose **Go to Page Definition**.

5.5.2 What Happens When You Create a Page Definition File

Example 5–6 shows a sample page definition file that was created for the SRList page in the SRDemo application. Notice that the page definition file groups the binding object definitions under the following wrapper elements:

- `parameters` (for more information, see Section 5.5.2.1, "Binding Objects Defined in the parameters Element")
- `executables` (for more information, see Section 5.5.2.2, "Binding Objects Defined in the executables Element")
- `bindings` (for more information, see Section 5.5.2.3, "Binding Objects Defined in the bindings Element")

Each wrapper element contains specific types of binding object definitions. The `id` attribute of each binding object definition specifies the name of the binding object. Each binding object name must be unique within the page definition file. By default, the binding objects are named after the data control object that was used to create it. If a data control object is used more than once on a page, JDeveloper adds a number to the default binding object names to keep them unique. In Section 5.6, "Creating ADF Data Binding EL Expressions", you will see how the ADF data binding EL expressions reference the binding object names.

Example 5–6 Page Definition File

```
<?xml version="1.0" encoding="UTF-8" ?>
<pageDefinition xmlns="http://xmlns.oracle.com/adfm/uimodel"
                version="10.1.3.35.65" id="SRListPageDef"
                Package="oracle.srdemo.view.pageDefs">
  <parameters/>
  <executables>
    <methodIterator id="findServiceRequestsIter"
                    Binds="findServiceRequests.result"
                    DataControl="SRPublicFacade" RangeSize="10"
                    BeanClass="oracle.srdemo.model.entities.ServiceRequest"/>
    <invokeAction Binds="findServiceRequests" id="tableRefresh"
                  Refresh="ifNeeded"
                  RefreshCondition="${(userState.refresh) and
                        (!adfFacesContext.postback)}"/>
    <variableIterator id="variables">
      <variable Type="java.lang.String" Name="setCurrentRowWithKey_rowKey"
                IsQueriable="false"/>
    </variableIterator>
  </executables>
```

```
  <bindings>
    <methodAction id="findServiceRequests"
                  InstanceName="SRPublicFacade.dataProvider"
                  DataControl="SRPublicFacade"
                  MethodName="findServiceRequests" RequiresUpdateModel="true"
                  Action="999"
                  ReturnName="SRPublicFacade.methodResults.
                          SRPublicFacade_dataProvider_findServiceRequests_result">
      <NamedData NDName="userIdParam" NDValue="#{userInfo.userId}"
                 NDType="java.lang.Integer"/>
      <NamedData NDName="statusParam" NDValue="#{userState.listMode}"
                 NDType="java.lang.String"/>
    </methodAction>
    <table id="findServiceRequests1" IterBinding="findServiceRequestsIter">
      <AttrNames>
        <Item Value="assignedDate"/>
        <Item Value="problemDescription"/>
        <Item Value="requestDate"/>
        <Item Value="status"/>
        <Item Value="svrId"/>
      </AttrNames>
    </table>
    <action id="setCurrentRowWithKey" IterBinding="findServiceRequestsIter"
            InstanceName="SRPublicFacade.dataProvider"
            DataControl="SRPublicFacade" RequiresUpdateModel="false"
            Action="96">
      <NamedData NDName="rowKey" NDValue="${row.rowKeyStr}"
                 NDType="java.lang.String"/>
    </action>
  </bindings>
</pageDefinition>
```

In later chapters, you will see how the page definition file is used to define and edit the bindings for specific UI components. For a description of all the possible elements and attributes in the page definition file, see Appendix A.7, "<pageName>PageDef.xml".

5.5.2.1 Binding Objects Defined in the parameters Element

The `parameters` element of the page definition file defines the parameters for the page. Parameter binding objects are argument values used by:

- A method action binding when invoking a bound method. For information about method action bindings, see Section 5.5.2.3, "Binding Objects Defined in the bindings Element".
- An iterator binding to fetch its data set. For information about iterator bindings, see Section 5.5.2.2, "Binding Objects Defined in the executables Element".

The parameter binding objects declare the parameters that the page evaluates at the beginning of a request (in the Prepare Model phase of the ADF lifecycle). In a web application, the page parameters are evaluated once during the Prepare Model phase. (For more information about the ADF lifecycle, see Section 6.2.3, "What Happens at Runtime: The JSF and ADF Lifecycles".) You can define the value of a parameter in the page definition file using static values, binding expressions, or EL expressions that assign a static value.

The SRDemo application does not use parameter bindings. However, Example 5–7 shows how parameter binding objects can be defined in a page definition file.

Example 5–7 The parameters Element of a Page Definition File

```
<parameters>
  <parameter id="filedBy"
             value="${bindings.userId}"/>
  <parameter id="status"
             value="${param.status != null ?  param.status : 'Open'}"/>
</parameters>
```

The value of the `filedBy` parameter is defined by a binding on the `userID` data attribute, which would be an attribute binding defined later in the `bindings` wrapper element. The value of the `status` parameter is defined by an EL expression, which assigns a static value.

> **Tip:** By default, JDeveloper uses the dollar sign (`$`), which is a JSP EL syntax standard, as the prefix for EL expressions that appear in the page definition file. However, you can use the hash sign (`#`) prefix, which is a JSF EL syntax standard, as well.

For more information about passing parameters to methods, see Chapter 10, "Creating More Complex Pages".

5.5.2.2 Binding Objects Defined in the executables Element

The `executables` element of the page definition file defines the following types of executable binding objects:

- `methodIterator`: Binds to an iterator that iterates over the collections returned by custom methods in the data control.

  A method iterator binding is always related to a `methodAction` binding defined in the `bindings` element. The `methodAction` binding encapsulates the details about how to invoke the method and what parameters (if any) the method is expecting. For more information about `methodAction` bindings, see Section 5.5.2.3, "Binding Objects Defined in the bindings Element".

- `accessorIterator`: Binds to an iterator that iterates over the detail objects returned by accessors on the data control. When you drop an accessor return or an attribute of an accessor return on a page, an `accessorIterator` is added to the executables element.

  Accessor iterators are always related to a method iterator that iterates over the master objects. The `MasterBinding` attribute of the accessor iterator binding defines the method iterator to which the accessor iterator is related. For more information about master-detail objects and iterators, see Chapter 8, "Displaying Master-Detail Data".

- `variableIterator`: Binds to an iterator that exposes all the variables in the binding container to the other bindings.

  Page variables are local to the binding container and exist only while the binding container object exists. When you use a data control method or operation that requires a parameter that is to be collected from the page, JDeveloper automatically defines a variable for the parameter in the page definition file. Attribute bindings can reference the page variables.

- `invokeAction`: Binds to a method that invokes the operations or methods defined in `action` or `methodAction` bindings during any phase of the page lifecycle.

Action and method action bindings are defined in the `bindings` element. For more information about `methodAction` objects, see Section 5.5.2.3, "Binding Objects Defined in the bindings Element".

Iterator binding objects bind to an underlying ADF `RowSetIterator` object, which manages the current object and current range information. The iterator binding exposes the current object and range state to the other binding objects used by the page. The iterator *range* represents the current set of objects to be displayed on the page. The maximum number of objects in the current range is defined in the `rangeSize` attribute of the iterator. For example, if a collection in the data control contains service requests and the iterator range size is 10, the first ten service requests in the collection are displayed on the page. If the user scrolls down, the next set of 10 service requests are displayed, and so on. If the user scrolls up, the previous set of 10 are displayed.

There is one iterator binding for each collection used on the page, but there is only one `variablesIterator` binding for all variables used on the page. (The variables iterator is like an iterator pointing to a collection that contains only one object whose attributes are the binding container variables.) All of the value bindings on the page must refer to an iterator binding to have the component values populated with data at runtime. (For information about value bindings, see Section 5.5.2.3, "Binding Objects Defined in the bindings Element".)

At runtime, the bindings in the `executables` element are refreshed in the order in which they appear in the page definition file. Refreshing an iterator binding reconnects it with its underlying `RowSetIterator` object. Refreshing an `invokeAction` binding invokes the action. However, before refreshing any bindings, the ADF runtime evaluates any `Refresh` and `RefreshCondition` attributes specified in the iterator and `invokeAction` elements. The `Refresh` attribute specifies the ADF lifecycle phase within which the executable should be invoked. The `RefreshCondition` attribute specifies the conditions under which the executable should be invoked. You can specify the `RefreshCondition` value using a boolean EL expression. If you leave the `RefreshCondition` attribute blank, it evaluates to `true`.

> **Tip:** Use the Structure window to re-order bindings in the `executables` element using drag and drop.

Example 5–8 shows an example of an `executables` element, which defines two types of iterators, method and variable, and an `invokeAction` object.

Example 5–8 The executables Element in a Page Definition File

```
<executables>
  <methodIterator id="findServiceRequestsIter"
                  Binds="findServiceRequests.result"
                  DataControl="SRPublicFacade" RangeSize="10"
                  BeanClass="oracle.srdemo.model.entities.ServiceRequest"/>
  <invokeAction Binds="findServiceRequests" id="tableRefresh"
                Refresh="ifNeeded"
                RefreshCondition="${(userState.refresh) and
                                    (!adfFacesContext.postback)}"/>
  <variableIterator id="variables">
    <variable Type="java.lang.String" Name="setCurrentRowWithKey_rowKey"
              IsQueriable="false"/>
  </variableIterator>
</executables>

<bindings>
  <methodAction id="findServiceRequests"
```

```
                InstanceName="SRPublicFacade.dataProvider"
                DataControl="SRPublicFacade"
                MethodName="findServiceRequests" RequiresUpdateModel="true"
                Action="999"
                ReturnName="SRPublicFacade.methodResults.
                        SRPublicFacade_dataProvider_findServiceRequests_result">
    <NamedData NDName="userIdParam" NDValue="#{userInfo.userId}"
               NDType="java.lang.Integer"/>
    <NamedData NDName="statusParam" NDValue="#{userState.listMode}"
               NDType="java.lang.String"/>
  </methodAction>

  <table id="findServiceRequests1" IterBinding="findServiceRequestsIter">
  ...
</bindings>
```

The `findServiceRequestIter` method iterator binding was created by dragging the `findServiceRequest` method return from the Data Control Palette onto the page. It iterates over the collection returned by the `findServiceRequest` method, which is defined in the `Binds` attribute. The `RangeSize` attribute defines the number of objects to fetch at one time. A `RangeSize` value of `-1` causes the iterator binding to display *all* the objects from the collection. In the `bindings` wrapper element, notice that the `IterBinding` attribute of the table binding object references the `findServiceRequestsIter` iterator binding, which populates the table with data.

In the example, the method iterator binding is related to the `findServiceRequests` method action binding, which is defined in the `bindings` element. The `NamedData` elements define the parameters to be passed to the method. The `table` binding in the `bindings` element references the `findServiceRequestIter` iterator binding in the `IterBinding` attribute to populate the data in the table.

The `invokeAction` object invokes the `findServiceRequests` method defined in the method action. The `Refresh` attribute determines when in the ADF lifecycle the method is executed, while the `RefreshCondition` attribute provides a condition for invoking the action. (For more information about the `Refresh` and `RefreshCondition` attributes, see Section A.7, "<pageName>PageDef.xml". For examples of using the `Refresh` and `RefreshCondition` attributes, see Section 10.8, "Creating Search Pages".)

The variable iterator iterates over the variable called `setCurrentRowWithKey_rowKey`, which is a parameter required by the `setCurrentRowWithKey` operation.

5.5.2.3 Binding Objects Defined in the bindings Element

The `bindings` element of the page definition file defines the following types of binding objects:

- Value: Display data in UI components by referencing an iterator binding. Each discrete UI component on a page that will display data from the data control is bound to a value binding object. Value binding objects include:
  - `table`, which binds an entire table to a data collection
  - `list`, which binds the list items to an attribute in a data collection
  - `tree`, which binds the root node of a tree to a data collection
  - `attributeValues`, which binds text fields to a specific attribute in an object (also referred to as an *attribute binding*)

- `methodAction`: Bind command components, such as buttons or links, to custom methods on the data control. A `methodAction` binding object encapsulates the details about how to invoke a method and what parameters (if any) the method is expecting.

  Method iterator bindings are bound to `methodAction` binding objects. There is one method action binding for each method iterator binding used in the page.

- `action`: Bind command components, such as buttons or links, to built-in data control operations (such as, `Commit` or `Rollback`) or to built-in collection-level operations (such as, `Create`, `Delete`, `Next`, `Previous`, or `Save`).

Collectively, the binding objects defined in the `bindings` element are referred to as *control* bindings, because each databound control on a page is bound to one of these objects, which in turn is bound to an object defined in the `executables` element.

Example 5–9 shows a sample `bindings` element, which defines one `methodAction` binding called `findServiceRequest`, one value binding for a table called `findServiceRequest1` and one action binding called `setCurrentRowWithKey`.

Example 5–9 The bindings Element of a Page Definition File

```
<bindings>
  <methodAction id="findServiceRequests"
                InstanceName="SRPublicFacade.dataProvider"
                DataControl="SRPublicFacade"
                MethodName="findServiceRequests" RequiresUpdateModel="true"
                Action="999"
                ReturnName="SRPublicFacade.methodResults.
                            SRPublicFacade_dataProvider_findServiceRequests_
                            result">
    <NamedData NDName="userIdParam" NDValue="#{userInfo.userId}"
               NDType="java.lang.Integer"/>
    <NamedData NDName="statusParam" NDValue="#{userState.listMode}"
               NDType="java.lang.String"/>
  </methodAction>
  <table id="findServiceRequests1" IterBinding="findServiceRequestsIter">
    <AttrNames>
      <Item Value="assignedDate"/>
      <Item Value="problemDescription"/>
      <Item Value="requestDate"/>
      <Item Value="status"/>
      <Item Value="svrId"/>
    </AttrNames>
  </table>
  <action id="setCurrentRowWithKey" IterBinding="findServiceRequestsIter"
          InstanceName="SRPublicFacade.dataProvider"
          DataControl="SRPublicFacade" RequiresUpdateModel="false"
          Action="96">
    <NamedData NDName="rowKey" NDValue="${row.rowKeyStr}"
               NDType="java.lang.String"/>
  </action>
</bindings>
```

Because the `findServiceRequests` method used in the sample accepts parameters, the method action binding contains `NamedData` elements that define the parameters expected by this method. For more information about passing parameters to methods, see Chapter 10, "Creating More Complex Pages".

In the table binding object, the `IterBinding` attribute references the appropriate iterator binding that displays the data in the component. Notice that a table is handled

by a single binding object that references the iterator binding once for all the attributes displayed in the table. This means that the table can only display data from a single data collection. However, for container components like forms, each individual attribute has a binding object, which references an iterator binding. Forms, unlike tables, can contain attributes from multiple data collections. The `AttrNames` element defines all the attribute values in the collection.

The action binding is bound to a command link that returns a specific service request when the user clicks the link. It references the `findServiceRequestsIter` iterator binding and passes it the current row key as a parameter, which is defined in the `namedData` element.

5.5.3 What Happens at Runtime

At runtime, the ADF page lifecycle passes the page URL to the ADF binding context, which matches the URL to a page definition file using the information in the `DataBindings.cpx` file. Next, the binding context instantiates the binding container if it does not already exist in the current session. The binding container is the runtime instance object that contains all of the binding objects defined in the page definition file. All the data that is displayed by a page's UI components is provided by the binding objects in the binding container. The ADF data binding expressions used by components on a page are evaluated at runtime and are replaced by values supplied by the binding objects when the page is rendered.

5.5.4 What You May Need to Know About Binding Container Scope

By default, the binding container and the binding objects it contains are defined in session scope. However, the values referenced by value bindings and iterator bindings are undefined between requests and for scalability reasons do not remain in session scope. Therefore, the values that binding objects refer to are only valid during a request in which that binding container has been prepared by the ADF lifecycle. What stays in session scope are only the binding container and binding objects themselves.

Upon each request, the iterator bindings are refreshed to rebind them to the underlying `RowSetIterator` objects. By default, the rowset iterator state and the data caches are maintained between requests.

5.6 Creating ADF Data Binding EL Expressions

In the previous section, you saw how the page definition is used to define the binding objects that are created in the binding container at runtime. To display data from the data model, web page UI components, are bound to binding objects using JSF Expression Language (EL) expressions. These EL expressions reference a specific binding object in a binding container. At runtime, the JSF runtime evaluates EL expression and pulls the value from the binding object to populate the component with data when the page is displayed. If the user updates data in the UI component, the JSF runtime pushes the value back into the corresponding binding object based on the same EL expression.

5.6.1 How to Create an ADF Data Binding EL Expression

When you use the Data Control Palette to create a component, the ADF data binding expressions are created for you. The expressions are added to every component attribute that will either display data from or reference properties of a binding object. Each prebuilt expression references the appropriate binding objects defined in the page definition file.

You can edit these binding expressions or create your own, as long as you adhere to the basic ADF binding expression syntax. ADF data binding expressions can be added to any component attribute that you want to populate with data from a binding object.

In JSF pages, a typical ADF data binding EL expression uses the following syntax to reference any of the different types of binding objects in the binding container:

```
#{bindingVariable.BindingObject.propertyName}
```

where:

- *`bindingVariable`* is a variable that identifies where to find the binding object being referenced by the expression. The `bindings` variable is the most common variable used in ADF binding expressions. It references the binding container of the *current* page. By default, all components created from the Data Control Palette use the `bindings` variable in the binding expressions.
- *`BindingObject`* is the name of the binding object as it is defined in the page definition file. The binding object name appears in the `id` attribute of the binding object definition in the page definition and is unique to that page definition. An EL expression can reference any binding object in the page definition file, including parameters, executables, or value bindings. When you use the Data Control Palette to create a component, JDeveloper assigns the names to the binding objects based on the names of the items in the data control.
- *`propertyName`* is a variable that determines the default display characteristics of each databound UI component and sets properties for the binding object at runtime. There are different binding properties for each type of binding object. For more information about binding properties, see Section 5.6.4, "What You May Need to Know About ADF Binding Properties".

For example, in the following expression:

```
#{bindings.SvrId.inputValue}
```

the `bindings` variable references a bound value in the current page's binding container. The binding object being referenced is `SvrId`, which is an attribute binding object. The binding property is `inputValue`, which returns the value of the first `SvrId` attribute.

> **Tip:** While the binding expressions in the page definition file can use either a dollar sign (`$`) or hash sign (`#`) prefix, the EL expressions in JSF pages can use only the hash sign (`#`) prefix.

For more examples of various types of ADF data binding expressions, see Section 5.6.3, "What Happens When You Create ADF Data Binding Expressions".

To create or edit an ADF Data Binding EL Expression

You can create or edit an expression in JDeveloper using any of the following techniques:

- Double-click the UI component in the Structure window, and edit the value field in the displayed editor. (Click the **Bind** button to go to the Expression Builder, where you can select from available binding objects and properties. For more information, see Section 5.6.2, "How to Use the Expression Builder".)

- View the web page using the source view of the visual editor and edit the expression directly in the source. JDeveloper provides Code Insight for EL expressions in the source editor. Code Insight is also available in the Property Inspector and the Tag Editor. To invoke Code Insight, type the leading characters of an EL expression (for example, `#{`). Code Insight displays a list of valid items for each segment of the expression from which you can select the one you want.
- Select a UI component in the visual editor or the Structure window and open the Property Inspector (**View** > **Property Inspector**). You can edit the expression directly in the Property Inspector, or click the ellipses next the expression to open the Expression Builder.

5.6.2 How to Use the Expression Builder

The JDeveloper Expression Builder is a dialog that helps you build EL expressions by providing lists of binding objects defined in the page definition files, as well as lists of managed beans and binding properties. It is particularly useful when creating or editing ADF databound expressions because it provides a hierarchical list of ADF binding objects and their most commonly used properties from which you can select the ones you want to use in an expression. For information about binding properties, see Section 5.6.4, "What You May Need to Know About ADF Binding Properties".

You can open the Expression Builder from either the Structure window or the Property Inspector.

To open the Expression Builder from the Structure window:

1. Double-click an ADF databound UI component in the Structure window.
2. In the ensuing dialog, click the **Bind** button next to a component property to display the Expression Builder.

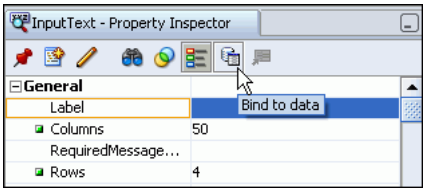
To open the Expression Builder from the Property Inspector:

1. Select a UI component in the Structure window or the visual editor and open the Property Inspector.
2. In the Property Inspector, take one of the following actions to display the Expression Builder:
   - Click the ellipses next to an existing binding expression.

     OR
   - Select a property to which you want to add a binding, and click the **Bind to data** button, as shown in Figure 5–5.

     (JDeveloper activates the Bind to data button only if it is valid to add a binding expression to the selected property.)

Figure 5–5 Bind to data Button in the Property Inspector

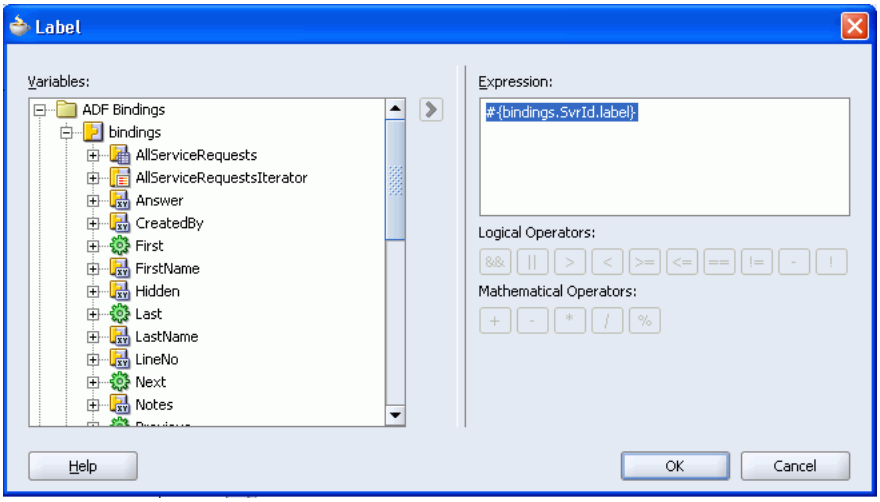


To use the Expression Builder:

1. Open the Expression Builder dialog.
2. In the Expression Builder, open the **ADF Bindings** > **bindings** node to display the ADF binding objects for the current page, as shown in Figure 5–6.

   For information about using the **ADF Bindings** > **data** node, see Section 5.6.5, "What You May Need to Know About Binding to Values in Other Pages".

Figure 5–6 The Expression Builder Dialog



3. Use the Expression Builder to edit or create ADF binding expressions using the following features:
   - Use the **Variables** tree to select items that you want to include in the binding expression. The tree contains a hierarchical representation of the binding objects. Each icon in the tree represents various types of binding objects that you can use in an expression (see Table 5–2 for a description of each icon). Select an item in the tree and click the shuttle button to move it to the **Expression** box.
   - If you are creating a new expression, begin typing the expression in the **Expression** box. JDeveloper provides Code Insight in the Expression Builder. To invoke Code Insight, type the leading characters of an EL expression (for example, `#{`) or a period separator. Code Insight displays a list of valid items for each segment of the expression from which you can select the one you want.

- Use the operator buttons under the expression to add logical or mathematical operators to the expression.

Table 5–2 Icons Under the ADF Bindings Node of the Expression Builder

Icon	Description
	Represents the `bindings` container variable, which references the binding container of the current page. Opening the **bindings** node exposes all the binding objects for the current page.
	Represents the `data` binding variable, which references the entire binding context. Opening the **data** node exposes all the page definition files in the application. Opening a page definition file exposes the binding objects it defines. Before using the objects under this node, see Section 5.6.5, "What You May Need to Know About Binding to Values in Other Pages" for more information and cautions.
	Represents a binding container. Each binding container node is named after the page definition file that defines it. These nodes appear only under the **data** node. Opening a binding container node exposes the binding objects defined for that page. Before using the object under this node, see Section 5.6.5, "What You May Need to Know About Binding to Values in Other Pages" for more information and cautions.
	Represents an action binding object. Opening a node that uses this icon exposes a list of valid action binding properties.
	Represents an iterator binding object. Opening a node that uses this icon exposes a list of valid iterator binding properties.
	Represents an attribute binding object. Opening a node that uses this icon exposes a list of valid attribute binding properties.
	Represents a list binding object. Opening a node that uses this icon exposes a list of valid list binding properties are displayed.
	Represents a table binding object. Opening a node that uses this icon exposes a list of valid table binding properties.
	Represents a tree binding object. Opening a node that uses this icon exposes a list of valid tree binding properties.
	Represents an ADF binding object property. For more information about ADF properties, see Section 5.6.4, "What You May Need to Know About ADF Binding Properties".
	Represents a parameter binding object.

5.6.3 What Happens When You Create ADF Data Binding Expressions

As was previously mentioned, when you create a component using the Data Control Palette, the ADF data binding expressions are added for you. Each expression is slightly different depending on the type of binding object being referenced.

5.6.3.1 EL Expressions That Reference Attribute Binding Objects

Example 5–10 shows a text field that was created when a data collection was dropped on a page as an **ADF Read-only Form**. Each UI component in the form, including the text field shown in the example, contains an EL expression that references an attribute binding object on a specific attribute in the data collection.

Example 5–10 EL Expressions That Reference an Attribute Binding Object

```
<af:inputText value="#{bindings.SvrId.inputValue}"
              label="#{bindings.SvrId.label}"/>
```

In this example, the UI component is bound to the `SvrId` binding object, which is a specific attribute in a data collection. The `inputValue` binding property returns the value of the first attribute to which the binding is associated, which in this case is `SvrId`. In the `label` attribute, the EL expression references the `label` binding property, which returns the label currently assigned to the data attribute.

The value binding object, `SvrId`, referenced by the EL expressions is defined in the page definition file, as shown in Example 5–11. The name of the binding object, which is referenced by the EL expression, is defined in the `id` attribute of the binding object definition.

Example 5–11 Attribute Binding Object Defined In the Page Definition File

```
<attributeValues id="SvrId" IterBinding="FindAllServiceRequestsIter"
                 isDynamic="true">
  <AttrNames>
     <Item Value="SvrId"/>
  </AttrNames>
</attributeValues>
```

> **Tip:** For a value binding that was created by dragging an attribute from an accessor return from the Data Control Palette, JDeveloper prefixes the accessor method name to the attribute name. For example, in the expression `#{bindings.ServiceRequestsvrId.label}`, the binding object name is a combination of the accessor method name, `ServiceRequest`, and the attribute name, `svrId`.

5.6.3.2 EL Expressions That Reference Table Binding Objects

When you drag a data collection from the Data Control Palette and drop it on a JSF page as an **ADF Read-only Table**, the resulting table tag typically contains a set of EL expressions that bind the table to a table value-binding object, as shown in Example 5–12.

Example 5–12 EL Expression That References a Table Binding Object

```
<af:table value="#{bindings.findAllStaff1.collectionModel}" var="row"
                 rows="#{bindings.findAllStaff1.rangeSize}"
                 first="#{bindings.findAllStaff1.rangeStart}"
                 emptyText="#{bindings.findAllStaff1.viewable ?
                             \'No rows yet.\' : \'Access Denied.\'}"
```

Each attribute of the `table` tag contains a binding expression that references the table binding object and an appropriate binding property for that tag attribute. The binding expression in the `rows` attribute references the iterator binding `rangeSize` property (which defines the number of rows in each page of the iterator) so that the number of rows rendered in the table matches the number of rows per page defined by the iterator binding.

The table is bound to the `findAllStaff1` table binding object, which is defined in the page definition file as shown in Example 5–13.

Example 5–13 Table Binding Object Defined in the Page Definition File

```
<table id="findAllStaff1" IterBinding="findAllStaffIter">
   <AttrNames>
     <Item Value="city"/>
     <Item Value="countryId"/>
     <Item Value="email"/>
     <Item Value="firstName"/>
     <Item Value="lastName"/>
     <Item Value="postalCode"/>
     <Item Value="stateProvince"/>
     <Item Value="streetAddress"/>
     <Item Value="userId"/>
     <Item Value="userRole"/>
   </AttrNames>
</table>
```

The `IterBinding` attribute in the table binding object refers to the iterator binding that will display data in the table.

5.6.3.3 EL Expressions That Reference Action Binding Objects

Example 5–14 shows a command button that was created by dragging a built-in operation from the Data Control Palette and dropping it on the page. The button contains an EL expression that binds to a built-in operation, `First`, which displays the first data object in the data collection to which the operation belongs.

Example 5–14 EL Expression That References an Action Binding Object for an Operation

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

The button's action listener is bound to the `execute()` method on the action binding named `First` in the binding container. When the user clicks the button, the action listener mechanism resolves the binding expression and then invokes the `execute()` method, which executes the operation. By default, the button label contains the name of the operation being called. You can change the label as needed. The `disabled` attribute determines if the button should be disabled on the page. Because of the not operator (`!`) at the beginning of the expression, the `disabled` attribute evaluates to the negation of the value of the `enabled` property of the action binding.

In other words, if the `enabled` property evaluates to `false`, the `disabled` attribute evaluates to `true`. For example, in an action binding that is bound to the `First` operation, if the current data object is the first one, the `enabled` property evaluates to `false`, which causes the `disabled` attribute to evaluate to `true`, thus disabling the button. However, if the current data object is not the first one, the `enabled` property evaluates to `true`, which causes the disabled attribute to evaluate to false, thus enabling the button.

Example 5–15 shows the action binding object defined in the page definition for the command button.

Example 5–15 Action Binding Object Defined in the Page Definition File for an Operation

```
<executables>
    <methodIterator id="findAllStaffIter" Binds="findAllStaff.result"
                    DataControl="SRPublicFacade" RangeSize="10"
                    BeanClass="oracle.srdemo.model.entities.User"/>
</executables>
<bindings>
    <methodAction id="findAllStaff" InstanceName="SRPublicFacade.dataProvider"
                  DataControl="SRPublicFacade" MethodName="findAllStaff"
                  RequiresUpdateModel="true" Action="999"
                  ReturnName="SRPublicFacade.methodResults.
                              SRPublicFacade_dataProvider_findAllStaff_result"/>
    <action id="First" IterBinding="findAllStaffIter"
            InstanceName="SRPublicFacade.dataProvider"
            DataControl="SRPublicFacade" RequiresUpdateModel="true"
            Action="12"/>
</bindings>
```

The `action` element, `First`, defines the action binding that is directly referenced by the EL expression in the command button. The `IterBinding` attribute of the action binding references the method iterator binding, `findAllStaffIter`, which iterates over the data collection being operated on by the action. The `findAllStaffIter` is bound to the `methodAction`, `findAllStaff`, which encapsulates the information required to invoke the `findAllStaff` method.

> **Tip:** The numerical value of the `Action` attribute of the `action` element references the number constants in the `OperationDefinition` interface in the `oracle.adf.model.meta` package.

Example 5–16 shows a command button that was created by dragging a method from the Data Control Palette and dropping it on a JSF page. In this example, the command button is bound to the `removeServiceHistory` method, which removes an object from the data collection. Parameters passed to the method when it is invoked identify which object to remove. The `execute` binding property in the EL expression in the `actionListener` attribute invokes the method when the user clicks the button

Example 5–16 EL Expression That References an Action Binding Object for a Method

```
<af:commandButton actionListener="#{bindings.removeServiceHistory.execute}"
                          text="removeServiceHistory"
                          disabled="#{!bindings.removeServiceHistory.enabled}"/
```

Example 5–17 shows the binding object created in the page definition file for the command button. When a command component is bound to a method, only one binding object is created in the page definition file—a `methodAction`. The `methodAction` binding defines the information needed to invoke the method, including any parameters, which are defined in the `NamedData` element.

Example 5–17 Method Action Binding Defined in the Page Definition File

```
<bindings>
    <methodAction id="removeServiceHistory"
                  InstanceName="SRPublicFacade.dataProvider"
                  DataControl="SRPublicFacade" MethodName="removeServiceHistory"
                  RequiresUpdateModel="true" Action="999">
      <NamedData NDName="serviceRequest"
                 NDType="oracle.srdemo.model.entities.ServiceRequest"/>
      <NamedData NDName="serviceHistory"
                 NDType="oracle.srdemo.model.entities.ServiceHistory"/>
    </methodAction>
</bindings>
```

5.6.4 What You May Need to Know About ADF Binding Properties

When you create a databound component using the Data Control Palette, the EL expression references specific ADF binding properties. At runtime, these binding properties can define such things as the default display characteristics of a databound UI component or specific parameters for iterator bindings. The ADF binding properties are defined by Oracle APIs. For a full list of the available properties for each binding type, see Appendix B, "Reference ADF Binding Properties".

Values assigned to certain properties are defined in the page definition file. For example, iterators can reference a property called `RangeSize`, which specifies the number of rows the iterator should display at one time. The value assigned to `RangeSize` is specified in the page definition file, as shown in Example 5–18

Example 5–18 Iterator Binding Object with the RangeSize Property

```
<accessorIterator id="serviceHistoryCollectionIterator" RangeSize="10"
                    Binds="serviceHistoryCollection"
                    DataControl="SRDemoSessionDataControl"
                    BeanClass="oracle.srdemo.model.ServiceHistory"
                    MasterBinding="findAllServiceRequestIter"/>
```

Use the JDeveloper Expression Builder to display a list of valid binding properties for each binding object. For information about how to use the Expression builder, see Section 5.6.2, "How to Use the Expression Builder".

5.6.5 What You May Need to Know About Binding to Values in Other Pages

While Oracle does not recommend this approach, you can access the bound values in another page's binding container from the current page using the `data` binding variable in an EL expression. The `data` binding variable references the binding context itself, which provides access to all the binding containers that are available. Use this variable when you want to bind to an object in the binding container of another page. The `data` variable must be immediately followed by the name of a page definition file that defines the binding container being referenced. For example:

```
#{data.mypagePageDef.BindingObject.propertyName}
```

At runtime, only the current incoming page's (or if the rendered page is different from the incoming, the rendered page's) binding container is automatically prepared by the framework during the current request. Therefore, to successfully access a bound value in another page from the current page, you must programmatically prepare that page's binding container *in the current request* (for example, using a backing bean). Otherwise, the bound values in that page may not be available or valid in the current request.

You may find cases, where you need to use the `data` variable to bind to values across binding containers. However, Oracle recommends that instead you use a backing bean to store page values and make them available to other pages. For more information about storing values in backing beans, see Section 10.2, "Using a Managed Bean to Store Information".

Caution: As was mentioned in Section 5.5.4, "What You May Need to Know About Binding Container Scope", the binding container, the binding objects it contains are defined in session scope, but the values referenced by the binding objects are not. By default, the `RowSetIterator` state and the data caches are maintained between requests, which makes the bound value referenced by a binding object available across pages.

However, when referring to binding objects across pages, you cannot rely on the bound values at session scope. The lifecycle of bound values is managed by the data control. The availability of a bound value during a given request depends on whether the data control itself is available and whether the referenced binding container has been prepared in the lifecycle. So, before referencing a bound value in one binding container from another page, be sure that the binding container being referenced is prepared during a given request.

Also, your application can, programmatically or through the use of the `CacheResults` or `Refresh` attributes on an iterator binding, re-execute or clear an iterator during a request. In this case, the binding object values would no longer be available to other pages. For more information about the iterator binding attributes that clear (`CacheResults`) or refresh (`Refresh`) the iterator, see Section A.7, "<pageName>PageDef.xml".

6

Creating a Basic Page

This chapter describes how to use the Data Control Palette to create databound forms using ADF Faces components.

This chapter includes the following sections:

- Section 6.1, "Introduction to Creating a Basic Page"
- Section 6.2, "Using Attributes to Create Text Fields"
- Section 6.3, "Creating a Basic Form"
- Section 6.4, "Incorporating Range Navigation into Forms"
- Section 6.5, "Modifying the UI Components and Bindings on a Form"

6.1 Introduction to Creating a Basic Page

You can create UI widgets that allow you to display and collect information using data controls created for your business services. For example, using the Data Control Palette, you can drag an attribute for an item, and then choose to display the value as either read-only text or as an input text field with a label.

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.

Once you drop the UI components, you can then drop built-in operations as command UI components that allow users to operate on the data. For example, you can create buttons that allow users to navigate between data objects displayed in the form. You can also modify the default components to suit your needs.

This chapter explains the following:

- How to create individual databound text fields
- How to create a form consisting of multiple text fields
- How to add operations that allow you to navigate between the data objects displayed in a form
- How to modify the form once it has been created

6.2 Using Attributes to Create Text Fields

To create text fields, you bind ADF Faces text UI components to attributes on a data control using an attribute binding. JDeveloper allows you to do this declaratively without the need to write any code. Additionally, JDeveloper provides a complete WYSIWYG development environment for your JSF pages, meaning you can design most aspects of your pages without needing to look at the code.

6.2.1 How to Use the Data Control Palette to Create a Text Field

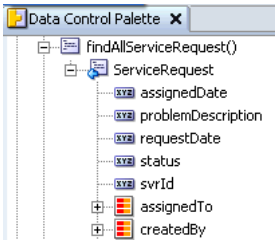
To create a text field that can display or update an attribute, you must bind the UI component to the attribute using a data control. JDeveloper allows you to do this declaratively by dragging and dropping an attribute of a collection from the Data Control Palette.

To create a bound text field:

1. From the Data Control Palette, select an attribute for a collection (for a description of which icon represents attributes and other objects in the Data Control Palette, see Section 5.2.1, "How to Understand the Items on the Data Control Palette").

   For example, Figure 6–1 shows the `problemDescription` attribute under the `ServiceRequest` collection for the `findAllServiceRequest()` method of the `SRPublicFacade` data control in the SRDemo application. This is the attribute to drop to display the problem description for a service request.

Figure 6–1 Attributes Associated with a Returned Object in the Data Control Palette



   If you wish to create input text fields used to collect data, you can use either a custom method or one of the data control's built-in creation methods. For procedures, see Section 10.7, "Creating an Input Form for a New Record".

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:

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

     Tip: For more information about validators, see Chapter 12, "Using Validation and Conversion".

  - **ADF Input Text**: Creates an ADF Faces `inputText` component with a `validator`. The `label` attribute is not populated.
  - **ADF Label**: An ADF Faces `outputLabel` component.

- **Single selections**

  These widgets display lists. For the purposes of this chapter, only the text widgets will be discussed. To learn about lists and their bindings, see Section 11.7, "Creating Databound Dropdown Lists".

6.2.2 What Happens When You Use the Data Control Palette to 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), using the name of the JSF page including the page's package name, and appending `PageDef` as the name of the page definition file. For example, the page definition file for the SREdit page is `app_staff_SREditPageDef.xml`. For a complete account of what happens when you drag an attribute onto a page, see Section 5.2.3, "What Happens When You Use the Data Control Palette". Bindings for the iterators and methods are created and added to the page definition file if they do not already exist, as are the bindings for each attribute. Additionally, the necessary JSPX page code for the UI component is added to the JSF page.

6.2.2.1 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 Control Palette (or you drop the whole collection as a form or table), JDeveloper creates a method iterator binding if it does not already exist. A method 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.

> **Tip:** There is one iterator created for each collection. This means that when you drop two attributes from the same collection (or drop the collection twice), they use the same iterator. This is fine, unless you need the iterator to behave differently for the different components. In that case, you will need to manually create separate iterators. For procedures and an example, see Section 10.9, "Conditionally Displaying the Results Table on a Search Page".

For example, if you drop the `problemDescription` attribute under the `ServiceRequest` collection for the `findAllServiceRequest()` method, JDeveloper creates a method iterator binding for the returned `ServiceRequest` collection.

The iterator binding's `rangeSize` attribute determines how many records will be available for the page 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 rowset. By default, it is set to `10`. For more information about using this attribute, see Section 6.4.2.2, "Iterator RangeSize Attribute". Example 6–1 shows the method iterator binding created when you drop an attribute from the `ServiceRequest` collection for the `findAllServiceRequest()` method.

Example 6–1 Page Definition Code for a Method Iterator Binding When You Drop an Attribute from a Method Return Collection

```
<executables>
    <methodIterator id="findAllServiceRequestIter"
                    Binds="findAllServiceRequest.result"
                    DataControl="SRPublicFacade" RangeSize="10"
                    BeanClass="oracle.srdemo.model.entities.ServiceRequest"/>
</executables>
```

For information regarding the iterator binding element attributes, see Section A.2.2, "Oracle ADF Data Binding Files".

JDeveloper also creates an action binding for the `findAllServiceRequest` method used to return the collection. Note that this action binding is created as a binding element and not an executable element. Example 6–2 shows the action binding created when you drop an attribute of the `ServiceRequest` collection for the `findAllServiceRequest()` method.

Example 6–2 Page Definition code for an Action Binding Used by the Iterator

```
  <bindings>
    <methodAction id="findAllServiceRequest"
                  InstanceName=""SRPublicFacade.dataProvider"
                  DataControl="SRPublicFacade"
                  MethodName="findAllServiceRequest" RequiresUpdateModel="true"
                  Action="999"
                  ReturnName="SRPublicFacade.methodResults.SRPublicFacade_
                            dataProvider_findAllServiceRequest_result"/>
```

This metadata allows the ADF binding container to access the attribute by allowing the iterator to access the result property on the associated method binding. Because the iterator is an executable, it is created when the page is loaded, and thus causes the method referenced in the method binding to execute.

In Example 6–1, the iterator is bound to the result property of the `findAllServiceRequest` method binding. This means that the iterator will manage all the returned service requests, including determining the current service request or range of service requests.

For information regarding the action binding element attributes, see Section A.2.2, "Oracle ADF Data Binding Files".

6.2.2.2 Creating and Using Value Bindings

When you drop an attribute from the Data Control Palette, 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 to both display and collect attribute values.

For example, if you drop the `problemDescription` attribute under the `ServiceRequest` collection for the `findAllServiceRequest()` method as an ADF Output Text w/Label widget onto the SREdit page, JDeveloper creates an attribute binding for the `problemDescription` attribute. Note that the attribute value references the `findAllServiceRequestIter` iterator. This allows the binding to access the attribute value of the current record. Example 6–3 shows the attribute binding for `problemDescription` created when you drop the attribute from the `ServiceRequest` collection for the `findAllServiceRequest()` method.

Example 6–3 Page Definition Code for an Attribute Binding

```
<bindings>
   ...
   <attributeValues id="ServiceRequestproblemDescription"
                    IterBinding="findAllServiceRequestIter">
     <AttrNames>
       <Item Value="problemDescription"/>
     </AttrNames>
   </attributeValues>
 </bindings>
```

For information regarding the attribute binding element attributes, see Section A.2.2, "Oracle ADF Data Binding Files".

6.2.2.3 Using EL Expressions to Bind UI Components

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

For example, when you drop the `ProblemDescription` attribute as an Output Text w/Label widget, JDeveloper creates an EL expression that binds the `panelLabelAndMessage` `label's` attribute to the `label` property of the `ProblemDescription` binding. It creates another expression that binds the `panelLabelAndMessage` value attribute to the `inputValue` property of the `ProblemDescription` binding, which in turn is the value of the `ProblemDescription` attribute. For more information about binding object properties, see Section A.2.2, "Oracle ADF Data Binding Files".

Example 6–4 shows the code generated on the JSF page when you drop an attribute as an Output Text w/Label widget.

Example 6–4 JSF Page Code for an Attribute Dropped as an Output Text w/Label

```
<af:panelLabelAndMessage
         label="#{bindings.ServiceRequestproblemDescription.label}">
    <af:outputText
         value="#{bindings.problemDescription.inputValue}"/>
</af:panelLabelAndMessage>
```

If instead you drop the `problemDescription` attribute as an Input Text w/Label widget, JDeveloper creates an `inputText` component. As Example 6–5 shows, similar to the output text component, the value is bound to the `inputValue` property of the `problemDescription` binding. Additionally, the following properties are also set:

- `label`: bound to the binding's `label` property.
- `required`: bound to the binding's `mandatory` property. See Section 12.3, "Adding Validation" for more information about this property.
- `columns`: bound to the `displayWidth` property. This determines how wide the text box will be.

Example 6–5 JSF Page Code for an Attribute Dropped as an Input Text w/Label

```
<af:inputText value="#{bindings.problemDescription.inputValue}"
                      label="#{bindings.problemDescription.label}"
                      required="#{bindings.problemDescription.mandatory}"
                      columns="#{bindings.problemDescription.displayWidth}">
  <af:validator binding="#{bindings.problemDescription.validator}"/>
</af:inputText>
```

For more information about the properties, see Appendix B, "Reference ADF Binding Properties".

6.2.3 What Happens at Runtime: The JSF and ADF Lifecycles

When a page is submitted and a new page requested, the application invokes both the JSF lifecycle and the ADF lifecycle. The JSF lifecycle handles the components at the view layer, while the ADF lifecycle handles the data at the model layer.

Specifically, the JSF lifecycle handles the submission of values on the page, validation for components, navigation, and displaying the components on the resulting page and saving and restoring state. The JSF lifecycle phases use a UI component tree to manage the display of the faces components. This tree is a runtime representation of a JSF page: each UI component tag in a page corresponds to a UI Component instance in the tree. The `FacesServlet` object manages the request processing lifecycle in JSF applications. `FacesServlet` creates an object called `FacesContext`, which contains the information necessary for request processing, and invokes an object that executes the lifecycle.

The ADF lifecycle handles preparing and updating the data model, validating the data at the model layer, and executing methods on the business layer. The ADF lifecycle uses the binding container to make data available for easy referencing by the page during the current page request.

The lifecycles are divided into phases. For the two lifecycles to work together, the ADF lifecycle phases are integrated with the JSF lifecycle phases using the JSF event listener mechanism. The ADF lifecycle listens for phase events using the ADF phase listener, which allows the appropriate ADF phases to be executed before or after the appropriate JSF phases.

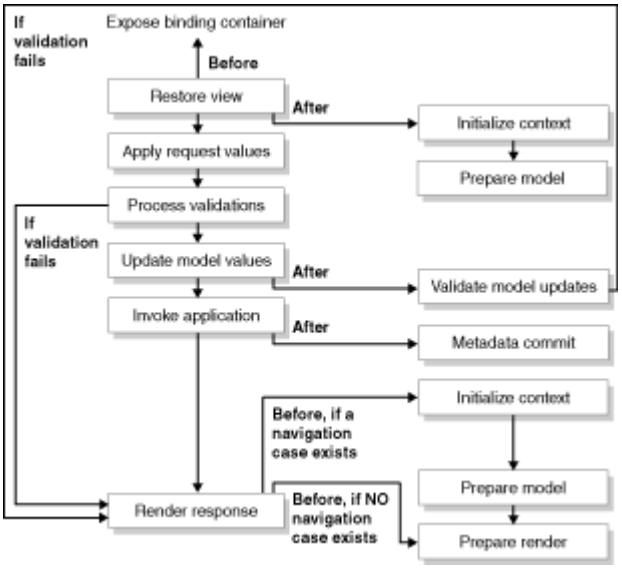
When an ADF Faces component bound to an ADF data control is inserted into a JSF page for the first time, JDeveloper adds the ADF PhaseListener to `faces-config.xml`. Example 6–6 shows the ADF PhaseListener configuration in `faces-config.xml`.

Example 6–6 Registering the ADF PhaseListener in faces-config.xml

```
<faces-config xmlns="http://java.sun.com/JSF/Configuration">
  ...
<lifecycle>
  <phase-listener>
    oracle.adf.controller.faces.lifecycle.ADFPhaseListener
  </phase-listener>
</lifecycle>
  ...
</faces-config>
```

Figure 6–2 shows how the JSF and ADF phases integrate in the lifecycle.

Figure 6–2 The Lifecycle of a Page Request in an ADF Application Using ADF Faces Components



In a JSF application that uses the ADF Model layer, the lifecycle is as follows:

- Restore View: The URL for the requested page is passed to the `bindingContext`, which finds the page definition that matches the URL. The component tree of the requested page is newly built or restored. All the component tags, event handlers, converters, and validators on the submitted page have access to the `FacesContext` instance. If it's a new empty tree (that is, there is no data from the submitted page), the lifecycle proceeds directly to the Render Response phase. Otherwise, the Restore View phase issues an event which the Initialize Context phase of the ADF Model layer lifecycle listens for and then executes.

  For example, for a page that contains an `inputText` UI component bound to the `ProblemDescription` attribute of a `ServiceRequest` returned collection, when the URL is passed in, the page definition is exposed. The UI component is then built. If data is to be displayed, the Initialize Context phase executes. Otherwise, the lifecycle jumps to the Render Response phase.

- Initialize Context: The page definition file is used to create the `bindingContainer` object, which is the runtime representation of the page definition for the requested page. The `LifecycleContext` class used to persist information throughout the ADF lifecycle phases is instantiated and initialized.

- Prepare Model: The binding container is refreshed, which sets any page parameters contained in the page definition. Any entries in the executables section of the page definition are refreshed, depending on the value of the `Refresh` and `RefreshCondition` attributes.

  The `Refresh` and `RefreshCondition` attributes are used to determine when and whether to invoke an executable. For example, there maybe an executable that should only be invoked under certain conditions. Refresh determines the phase in which to invoke the executable, while the refresh condition determines whether the condition has been met. Set the `Refresh` attribute to `prepareModel` when your bindings are dependent on the outcome from the operation.

If `Refresh` is set to `prepareModel`, or if no value is supplied (meaning it uses the default, `ifneeded`), then the `RefreshCondition` attribute value is evaluated. If no `RefreshCondition` value exists, the executable is invoked. If a value for `RefreshCondition` exists, then that value is evaluated, and if the return value of the evaluation is true, then the executable is invoked. If the value evaluates to false, the executable is not invoked. The default value always enforces execution. If the incoming request contains no POST data or query parameters, then the lifecycle forwards to the Render Response phase.

For more information, see Section 10.5.5.1, "Correctly Configuring the Refresh Property of Iterator Bindings". For details about the refresh attribute, see Section A.7.1, "PageDef.xml Syntax".

In the problem description example, the `bindingContainer` invokes the `findAllServiceRequestIter` method iterator, which in turn invokes the `findAllServiceRequest` method that returns the `ServiceRequest` collection. The iterator then iterates over the data and makes the data for the first found record available to the UI component by placing it in the binding container. Because there is a binding for the `problemDescription` attribute in the page definition that can access the value from the iterator (see Example 6–3), and since the UI component is bound to the `problemDescription` binding using an EL expression (`#{bindings.problemDescription.inputValue}`), that data is displayed by that component.

- Apply Request Values: Each component in the tree extracts new values from the request parameters (using its decode method) and stores it locally. Most associated events are queued for later processing. If a component has its `immediate` attribute set to `true`, then the validation, conversion, and events associated with the component are processed during this phase. For more information about validation and conversion, see Chapter 12, "Using Validation and Conversion".

  For example, if a user enters a new value into the `inputText` component, that value is stored locally using the `setSubmittedValue` method on the `inputText` component.

- Process Validations: Local values of components are converted and validated. If there are errors, the lifecycle jumps to the Render Response phase. At the end of this phase, new component values are set, any validation or conversion error messages and events are queued on `FacesContext`, and any value change events are delivered.

  For a detailed explanation of this and the next two phases of the lifecycle, see Chapter 12, "Using Validation and Conversion".

- Update Model Values: The component's validated local values are moved to the model and the local copies are discarded.

- Validate Model Updates: The updated model is now validated against any validation routines set on the model.

- Invoke Application: Any action bindings for command components or events are invoked. For a detailed explanation of this and the next two phases of the lifecycle, see Section 9.4, "Using Dynamic Navigation". For a description of action bindings used to invoke business logic, see Section 6.4, "Incorporating Range Navigation into Forms".

- Metadata Commit: Changes to runtime metadata are committed. This phase is not used in this release, but will be used in future releases.

- Initialize Context (only if navigation occurred in the Invoke Application lifecycle): The page definition for the next page is initialized.

- Prepare Model (only if navigation occurred in the Invoke Application lifecycle): Any page parameters contained in the next page's definition are set. Any entries in the executables section of the page definition are used to invoke the corresponding methods in the order they appear.
- Prepare Render: The binding container is refreshed to allow for any changes that may have occurred in the Apply Request Values or Validation phases. The `prepareRender` event is sent to all registered listeners.

> **Note:** Instead of displaying `prepareRender` as a valid phase for a selection, JDeveloper displays `renderModel`, which represents the `refresh(RENDER_MODEL)` method called on the binding container.

You should set the `Refresh` attribute of an executable to `renderModel` when the `refreshCondition` is dependent on the model. For example, if you want to use the `#{adfFacesContext.postback}` expression in a `RefreshCondition` of an executable, you must set the `Refresh` property to either `renderModel` or `renderModelIfNeeded`, which will cause the method to be executed during the `prepareRender` phase. For more information, see Section 10.5.5.1, "Correctly Configuring the Refresh Property of Iterator Bindings".

- Render Response: The components in the tree are rendered as the J2EE web container traverses the tags in the page. State information is saved for subsequent requests and the Restore View phase.

6.3 Creating a Basic Form

Instead of dropping individual attributes to create a form, JDeveloper allows you to drop a complete form that displays or collects data for all the attributes on an object. For example, the SREdit page was created by dropping the return from the `findServiceRequestById` method, which contains all the attributes necessary to edit a given service request.

This section provides information on creating a form that returns data to be viewed or edited. You can also use a constructor or the method itself to create a form used to populate data instead of return data. For more information about creating that type of form, see Section 10.7, "Creating an Input Form for a New Record".

6.3.1 How to Use the Data Control Palette 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 collection or a structured attribute from the Data Control Palette.

These procedures are for creating a form that displays all objects from a collection returned by a method that takes no parameters. If you want to use a collection returned from a method that takes parameters, you need to have those parameters set in order for the form to display the proper records. For procedures and information about setting parameters, see Section 10.6.1, "How to Create a Form or Table Using a Method That Takes Parameters".

To create a form that allows a user to create a new record, you need to use a method that creates the new instance, given some values for that instance. If your data control was configured to support updates, then it will include constructors, which are objects you can use to create a form that creates a new object, populating values for all attributes on the object. For more information, see Section 10.7, "Creating an Input Form for a New Record".

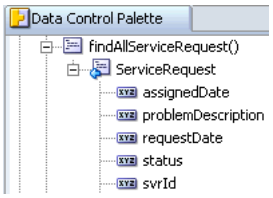
Whether you use a collection, a constructor, or a method to create a form, you may also want to add a command button that invokes a method to, for example, insert the data into the data source or update the data. For procedures and more information, see Section 10.3, "Creating Command Components to Execute Methods".

To create a basic form:

1. From the Data Control Palette, select a collection that is a return of a `findAll` method.

   To display the value of existing attributes, drop the returned collection from a method used to find records. Figure 6–3 shows the `ServiceRequest` collection for the `findAllServiceRequest()` method from the SRDemo application. This method creates a form with data already populated in the input text fields.

Figure 6–3 Attributes Associated with a Returned Collection in the Data Control Palette



2. Drag the collection onto the page, and from the context menu choose the type of form 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 creating 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, except for dates, which use the `selectInputDate` component. Each `inputText` component contains a validator tag that allows you to set up validation for the attribute. For more information, see Section 12.3, "Adding Validation".

     You can elect to include navigational controls that allow users to navigate through all the data objects in the collection. For more information, see Section 6.4, "Incorporating Range Navigation into Forms". You can also elect to 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 `panelForm` component.

   – **ADF Read-Only Form**: Same as the ADF Form, but by default, `outputText` components are used. Since the form is meant to display data, no `validator` tags are added. The `label` attribute is populated for each component. Attributes of type `Date` also use the `outputText` component. All components are placed inside `panelLabelAndMessage` components, which are in turn placed inside a `panelForm` component.

- **ADF Creation Form**: Not to be used when using TopLink Java objects and an EJB session bean. Use constructors or custom methods instead. For more information, see Section 10.7, "Creating an Input Form for a New Record".

3. 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 10.3, "Creating Command Components to Execute Methods".

6.3.2 What Happens When You Use the Data Control Palette to Create a Form

Dropping an object as a form from the Data Control Palette 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`). Example 6–7 shows the code generated on the JSF page when you drop the `ServiceRequest` collection for the `findAllServiceRequest()` method as a default ADF Form.

Example 6–7 Code on a JSF Page for an Input Form

```
<af:panelForm>
      <af:inputText value="#{bindings.svrId.inputValue}"
                    label="#{bindings.svrId.label}"
                    required="#{bindings.svrId.mandatory}"
                    columns="#{bindings.svrId.displayWidth}">
        <af:validator binding="#{bindings.svrId.validator}"/>
        <f:convertNumber groupingUsed="false"
                         pattern="#{bindings.svrId.format}"/>
      </af:inputText>
      <af:inputText value="#{bindings.status.inputValue}"
                    label="#{bindings.status.label}"
                    required="#{bindings.status.mandatory}"
                    columns="#{bindings.status.displayWidth}">
        <af:validator binding="#{bindings.status.validator}"/>
      </af:inputText>
      <af:selectInputDate value="#{bindings.requestDate.inputValue}"
                          label="#{bindings.requestDate.label}"
                          required="#{bindings.requestDate.mandatory}">
        <af:validator binding="#{bindings.requestDate.validator}"/>
        <f:convertDateTime pattern="#{bindings.requestDate.format}"/>
      </af:selectInputDate>
      <af:inputText value="#{bindings.problemDescription.inputValue}"
                    label="#{bindings.problemDescription.label}"
                    required="#{bindings.problemDescription.mandatory}"
                    columns="#{bindings.problemDescription.displayWidth}">
        <af:validator binding="#{bindings.problemDescription.validator}"/>
      </af:inputText>
      <af:selectInputDate value="#{bindings.assignedDate.inputValue}"
                          label="#{bindings.assignedDate.label}"
                          required="#{bindings.assignedDate.mandatory}">
        <af:validator binding="#{bindings.assignedDate.validator}"/>
        <f:convertDateTime pattern="#{bindings.assignedDate.format}"/>
      </af:selectInputDate>
      <f:facet name="footer">
        <af:commandButton text="Submit"/>
      </f:facet>
</af:panelForm>
```

Note: For information regarding the validator and converter tag, see Chapter 12, "Using Validation and Conversion".

6.3.2.1 Using Facets

JSF components use facet tags to hold other components that require a special relationship with the parent component, for example, headers and footers for columns within a table, or the `footer` facet for the `panelForm` component. When you use the Data Control Palette to drop a widget, any preferred facets are included.

Many components use facets, and when you use widgets to create complex components (such as `panelForm`), output tags are often automatically created and inserted into the facets. You can manually edit these components or add other components to facets.

When you choose to include a **Submit** button when you drop a collection as an input form, a command button is added to the `panelForm's footer` facet. This command button causes the form that holds the `panelForm` to be submitted. By default, the text is **Submit**. Figure 6–4 shows the command button in the `panelForm's footer` facet.

Figure 6–4 Footer Facet for the Panel Form

Example 6–8 shows the corresponding code in the JSF page.

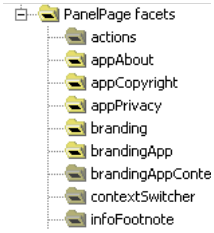
Example 6–8 Facet in a JSF Page

```
<af:panelForm>
...
  <f:facet name="footer">
    <af:commandButton text="Submit"/>
  </f:facet>
</af:panelForm>
```

Each facet can hold only one component. If you need a facet to hold more than one component, then you need to nest those components in a container component, which can then be nested in the facet. For an example of how the `panelGroup` and `panelButtonBar` components are used to group all buttons in the `footer` facet of a form, see Section 6.4.2.3, "Using EL Expressions to Bind to Navigation Operations".

Also note that JDeveloper displays all facets available to the component in the Structure window. However, only those that contain UI components appear activated. Any empty facets are "grayed" out. Figure 6–5 shows both full and empty facets for a `panelPage` component

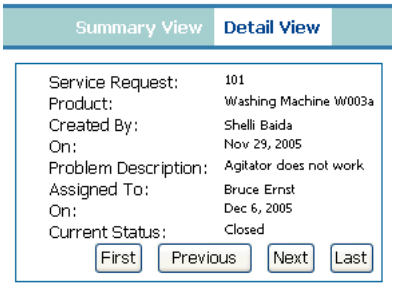
Figure 6–5 Empty and Full Facet Folders in the Structure Window



6.4 Incorporating Range Navigation into Forms

When you choose to add navigation when you use the Data Control Palette to create an input form, 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 the data objects in the collection. Figure 6–6 shows a form that contains navigation buttons.

Figure 6–6 Navigation in a Form



6.4.1 How to Insert Navigation Controls into a Form

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

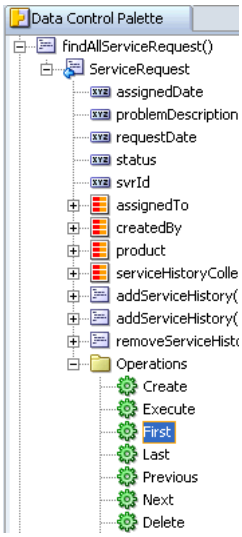
You can also add navigation buttons to an existing form manually.

To manually add navigation buttons:

1. From the Data Control Palette, 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 service requests, you would drag the **Next** operation associated with the `ServiceRequest` collection of the `findAllServiceRequest()` method. Figure 6–7 shows the navigation operations associated with a collection.

Figure 6–7 Navigation Operations Associated With a Collection



2. Choose either **Command Button** or **Command Link** from the context menu.

6.4.2 What Happens When Command Buttons Are Created Using the Data Control Palette

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

- Defines an action binding in the page definition file for the associated operations
- Inserts code in the JSF page for the command components

6.4.2.1 Using Action Bindings for Built-in Navigation Operations

Action bindings execute business logic. Action bindings can invoke methods on a business service (for example, the method action binding for a method used by an iterator to access a collection) or as in the case of navigation controls, 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 Control Palette. JDeveloper provides navigation operations that allow users to navigate forward, backwards, to the last object in the collection, and to the first object.

Note: For more information about using methods to add, remove, or update data, see Chapter 10.3, "Creating Command Components to Execute Methods".

Like value bindings, action bindings for navigation operations must also contain a reference to the iterator binding, as it is used to determine the current object and can therefore determine the correct object to display when each of the navigation buttons is clicked. Example 6–9 shows the action bindings for the navigation operations.

Tip: The numerical values of the `Action` attribute in the `<action>` tags (as shown in Figure 6–9) are defined in the `oracle.adf.model.meta.OperationDefinition` class. However, when you use the ADF Model layer's action binding editor, you never need to set the numerical code by hand.

Example 6–9 Page Definition Code for an Operation Action Binding

```
<action id="First" RequiresUpdateModel="true" Action="12"
          IterBinding="findAllServiceRequestIterator"/>
<action id="Previous" RequiresUpdateModel="true" Action="11"
          IterBinding="findAllServiceRequestIterator"/>
<action id="Next" RequiresUpdateModel="true" Action="10"
          IterBinding="findAllServiceRequestIterator"/>
<action id="Last" RequiresUpdateModel="true" Action="13"
          IterBinding="findAllServiceRequestIterator"/>
```

6.4.2.2 Iterator RangeSize Attribute

Iterator bindings have a `rangeSize` attribute used to determine the number of data objects to return 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, only a set number are returned and accessible by the other bindings. Once the iterator reaches the end of the range, it accesses another set, creating a new range. Example 6–10 shows the range size for the `findAllServiceRequestIter` iterator.

> **Note:** This `rangeSize` attribute is not the same as the `row` attribute on a table component. For more information, see Table 7–1, " ADF Faces Table Attributes and Populated Values".

Example 6–10 RangeSize Attribute for an Iterator

```
<executables>
  <methodIterator id="findAllServiceRequestIter"
                      Binds="findAllServiceRequest.result"
                      DataControl="SRPublicFacade" RangeSize="10"
                      BeanClass="oracle.srdemo.model.entities.ServiceRequest"/>
</executables>
```

By default, the `rangeSize` attribute is set to `10`. This means that a user can view 10 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 10), the binding object executes its associated method against the iterator, and the iterator retrieves another set of 10 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. The default is -1 for iterator bindings that furnish a list of valid choices for list bindings.

> **Tip:** You can also set a range of records directly in the query you write on your business service. However, doing so means every page that uses the query will return the same range size. By setting the range size on the iterator, you can control the size per page.

Table 6–1 shows the built-in navigation operations provided on data controls, along with the action attribute value set in the page definition, and the result of invoking the operation or executing an event bound to the operation. For more information about action events, see Section 6.4.3, "What Happens at Runtime: About Action Events and Action Listeners".

Table 6–1 Built-in Navigation Operations

Operation	Action Attribute Value	When invoked, the associated iterator binding will...
Next	10	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	11	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.
First	12	Move its current pointer to the beginning of the result set.
Last	13	Move its current pointer to the end of the result set.
Next Set	14	Move the range forward a number of objects equal to the range size attribute.
Previous Set	15	Move the range backward a number of objects equal to the range size attribute.
SetCurrentRow WithKey	96	Set the row key as a `String` converted from the value specified by the input field. The row key is used to set the currency of the data object in the bound data collection. For an example of when this is used, see Section 7.7.1, "How to Manually Set the Current Row".
SetCurrentRow WithKeyValue	98	Set the current object on the iterator, given a key's value.

Every action binding for an operation has an `enabled` boolean property that the ADF framework sets to `false` when the operation should not be invoked. You can then bind the UI component to this value to determine whether or not the component should be enabled. For more information about the `enabled` property, see Appendix B, "Reference ADF Binding Properties".

6.4.2.3 Using EL Expressions to Bind to Navigation Operations

When you create command components using navigation operations, the components are placed in a `panelButtonBar` 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. This expression causes the binding's operation to be invoked on the iterator when a user clicks the button.

For more information about the command button's `actionListener` attribute, see Section 6.4.3, "What Happens at Runtime: About Action Events and Action Listeners". For example, the **First** command button's `actionListener` attribute is bound to the `execute` method on the `First` action binding.

The `disabled` attribute is used to determine if the button should be inactivated. For example, if the user is currently displaying the first record, the **First** button should not be able to be clicked. The code uses an EL expression that evaluates to the `enabled` property on the action binding. If the property value is not `true` (for example, if the current record is the first record, the `First` operation should not be enabled), then the button is disabled. Example 6–11 shows the code generated on the JSF page for navigation operation buttons.

Example 6–11 JSF Code for Navigation Buttons Bound to ADF Operations

```
<f:facet name="footer">
   <af:panelButtonBar>
     <af:commandButton actionListener="#{bindings.First.execute}"
                       text="First"
                       disabled="#{!bindings.First.enabled}"/>
     <af:commandButton actionListener="#{bindings.Previous.execute}"
                       text="Previous"
                       disabled="#{!bindings.Previous.enabled}"/>
     <af:commandButton actionListener="#{bindings.Next.execute}"
                       text="Next"
                       disabled="#{!bindings.Next.enabled}"/>
     <af:commandButton actionListener="#{bindings.Last.execute}"
                       text="Last"
                       disabled="#{!bindings.Last.enabled}"/>
   </af:panelButtonBar>
   <af:commandButton text="Submit"/>
 </f:facet>
```

6.4.3 What Happens at Runtime: About Action Events and Action Listeners

An action event occurs when a command component is activated. For example, when a user clicks a button, the form the component is enclosed in is submitted, and subsequently an action event is fired. Action events might affect only the user interface (for example, a link to change the locale, causing different field prompts to display), or they might involve some logic processing in the back end (for example, a button to navigate to the next record).

An action listener is a class that wants to be notified when a command component fires an action event. An action listener contains an action listener method that processes the action event object passed to it by the command component.

In the case of the navigation operations, when a user clicks, for example, the **Next** button, an action event is fired. This event stores currency information about the current data object, taken from the iterator. Because the component's `actionListener` attribute is bound to the `execute` method of the `Next` action binding, the `Next` operation is invoked. This method takes the currency information passed in the event object to determine what the next data object should be.

6.4.4 What You May Need to Know About the Browser Back Button

When a user clicks the navigation buttons, the iterator determines the next data object to display. However, when the user clicks the browser's **Back** button, the action and/or event is not shared outside the browser, and the iterator is bypassed. Therefore, when a user clicks a browser's **Back** button instead of using navigation buttons on the page, the iterator becomes out of sync with the page displayed, causing unexpected results.

For example, say a user browses to object 103, and then uses the browser's **Back** button. Because the browser shows the page last visited, object 102 is shown. However, the iterator still thinks the current object is 103 because the iterator was bypassed. If the user were to then to click the **Next** button, object 104 would display because that is what the iterator believes to be the next object, and not 103 as the user would expect.

Because the iterator and the page are out of sync, problems can arise when a user edits records. For example, if the user were to have edited object 102 after clicking the browser's **Back** button, the changes would have actually been posted to 103, because this is what the iterator thought was the current object.

To prevent a user making changes to the wrong object instances, you can use token validation. When you enable token validation for a page, that page is annotated with the current object for all the iterators used to render that page. This annotation is encoded onto the HTML payload rendered to the browser and is submitted to the server along with any data. At that point, the current object of the iterator is compared with the annotation. If they are different, an exception is thrown.

For example, in the earlier scenario, when the user starts at 103 but then clicks the browser's **Back** button to go to 102, as before, the previous page is displayed. However, that page was (and still is) annotated with 102. Therefore, when the user clicks the **Next** button to submit the page and navigate forward, the annotation (102) does not match the iterator (which is still at 103), an exception is thrown, and the `Next` operation is not executed. The page renders with 103, which is the object the iterator believed to be current. An error displays on the page stating that 102 was expected, since the server expected 102 based on the annotation submitted with the data. Since 103 is now displayed, both the annotation and the iterator are now at 103, and are back in sync.

Token validation is set on the page definition for a JSF page. By default, token validation is on.

To set token validation:

1. Open the page definition file for the page.
2. In the Structure window, select the root node for the page definition itself.
3. In the Property Inspector, use the dropdown list for the `EnableTokenValidation` attribute to set validation to **true** to turn on token validation, or **false** to turn off token validation.

Example 6–12 shows a page definition file after token validation was set to true.

Example 6–12 Enable Token Validation in the Page Definition File

```
<pageDefinition xmlns="http://xmlns.oracle.com/adfm/uimodel"
                version="10.1.3.35.29" id="createProductPageDef"
                Package="oracle.srdemo.view.pageDefs"
                EnableTokenValidation="true">
```

6.5 Modifying the UI Components and Bindings on a Form

Once you use the Data Control Palette to create a form, you can then delete attributes, change the order in which they are displayed, change the component used to display them, and change the attribute to which they are bound.

6.5.1 How to Modify the UI Components and Bindings

You can modify certain aspects of the default components dropped from the Data Control Palette. You can use the Structure window to change the order in which components are displayed, to add new components or change existing components, or to delete components. You can use the Property Inspector to change or delete bindings, or to change the label displayed for a component.

To modify default components and bindings:

1. Use the Structure window to do the following:

   - Change the order of the UI components by dragging them up or down the tree. A black line with an arrowhead denotes where the UI component will be placed.
   - Add a UI component for a new attribute. Right-click an existing UI component in the Structure window and choose to place the new component before, after, or inside the selected component. You then choose from a list of UI components.

     To bind the new component to an attribute, you need to use the Property Inspector. See the first bullet point in step 2 for details.
   - Delete a UI component. Right-click the component and choose **Delete**. If you wish to keep the component, but delete just the binding, you need to use the Property Inspector. See the second bullet point in step 2.

2. With the UI component selected in the Structure window, you can then do the following in the Property Inspector:

   - Add a binding for the UI component. Enter an EL expression in the **Value** field, or click the ellipsis (**...**) button in the **Value** field to open the EL Expression Builder. To select a binding available from the data control, select the **ADF Bindings > Bindings** node. This node shows the operations, iterators, and attributes available from the collection currently bound, as well as the binding properties. For more information about using EL expressions, see Section 5.6, "Creating ADF Data Binding EL Expressions".
   - Delete a binding for the UI component by deleting the EL expression.
   - Change the binding. You can rebind the component to any other attribute, or any property on another attribute. For procedures, see Section 6.5.1.1, "Changing the Value Binding for a UI Component".
   - Change the label for the UI component. By default, the label is bound to the binding's `label` property (for more information about this property, see Appendix B, "Reference ADF Binding Properties".

     You can also change the label just for the current page. To do so, select the `Label` attribute. You can enter text or an EL expression to bind the label value to something else, for example, a key in a properties or resource file.

     For example, the `inputText` component used to enter the status of a service request would have the following for its `Label` attribute:

     ```
     #{bindings.status.label}
     ```

     In this expression, `status` is the ID for the attribute binding in the page definition file.

     However, you could change the expression to instead bind to a key in a properties file, for example:

     ```
     #{srproperties['sr.status']}
     ```

     In this example, `srproperties` is a variable defined in the JSF page used to load a properties file. The SREdit page uses a variable named `res`. The label for the cancel button has the following value:

     ```
     #{res['srdemo.cancel']}
     ```

For more information about using resource bundles, see Section 14.4, "Internationalizing Your Application".

6.5.1.1 Changing the Value Binding for a UI Component

Instead of modifying a binding, you can completely change the object to which the UI component in a form is bound.

To rebind a UI component:

1. From the Data Control palette, drag the collection or attribute that you now want the component to be bound to, and drop it on the component.

   OR

   Right-click the UI component in the Structure window and choose **Edit Binding**. Either the Attribute, Table, or List Binding Editor launches, depending on the UI component for which you are changing the binding.

2. In the context menu, select **Bind existing <*component name*>**.

6.5.1.2 Changing the Action Binding for a UI Component

When a component is bound to a built-in operation, you can change the action using the Action Binding Editor.

To rebind a UI Command component:

1. Right-click the command component in the Structure window and choose **Edit Binding**, which launches the Action Binding Editor.
2. In the editor, use the dropdown menu to select a different action.

6.5.2 What Happens When You Modify Attributes and Bindings

When you modify how an attribute is displayed by moving the UI component or changing the UI component, JDeveloper changes the corresponding code on the JSF page. When you use the binding editors to add or change a binding, JDeveloper adds the code to the JSF page, and also adds the appropriate elements to the page definition file.

7

Adding Tables

This chapter describes how to use the Data Control Palette to create databound tables using ADF Faces components.

This chapter includes the following sections:

- Section 7.1, "Introduction to Adding Tables"
- Section 7.2, "Creating a Basic Table"
- Section 7.3, "Incorporating Range Navigation into Tables"
- Section 7.4, "Modifying the Attributes Displayed in the Table"
- Section 7.5, "Adding Hidden Capabilities to a Table"
- Section 7.6, "Enabling Row Selection in a Table"
- Section 7.7, "Setting the Current Object Using a Command Component"

7.1 Introduction to Adding Tables

Unlike forms, tables allow you to display more than one data object from a collection at a time. Figure 7–1 shows the SRList page in the SRDemo application, which uses a browse table to display the current service requests for a logged in user.

Figure 7–1 The Service Request Table

My Service Requests

Select and View Edit

Select	Request Id	Status	Requested On	Problem	Assigned On
◉	200	Open	Dec 19, 2005	Seal not working	Jan 11, 2006
○	201	Open	Dec 20, 2005	Dryer is spitting out lots of lint.	Dec 20, 2005
○	202	Open	Dec 20, 2005	Leaking at the sides	Dec 21, 2005

Once you drop a collection as a table, you can then add selection components that allow users to select a specific row. When you add command buttons bound to methods, users can then click those buttons to execute some business logic on the selected row. For more information, see Section 10.3, "Creating Command Components to Execute Methods". You can also modify the default components to suit your needs.

Read this chapter to understand:

- How to create a basic table
- How to add navigation between sets of returned objects
- How to modify the default table once it's created
- How to add components that allow users to show or hide data
- How to include a column that allows users to select one, or one or more, rows in the table
- How to manually set the current row in the table

7.2 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 columns in the table 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.

7.2.1 How to Create a Basic Table

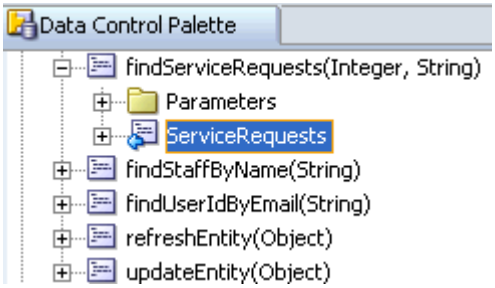
To create a table using a data control, you must bind to a method on the data control that returns a collection. JDeveloper allows you to do this declaratively by dragging and dropping a collection from the Data Control Palette.

To create a databound table:

1. From the Data Control Palette, select a method return that returns a collection.

   For example, to create the SRList table in the SRDemo application, you drag the ServiceRequest collection that the `findServiceRequest(Integer, String)` method returns. This method takes an `Integer` parameter value that represents the user ID of the current user and a `String` value that represents the status of `open`, and returns all open requests for that user. Figure 7–2 shows the `ServiceRequests` collection in the Data Control Palette. For more information about how the parameters are set to determine the records to display, see Section 10.6.1, "How to Create a Form or Table Using a Method That Takes Parameters".

Figure 7–2 Collection Returned for a Method



2. Drag the method return 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 wish your editable table columns to display, and what UI components to use to display the data. By default, each attribute on the collection object is displayed in an `inputText` component, thus enabling the table to be editable.
   - **ADF Read-Only Table:** Same as the **ADF Table**; however, each attribute is displayed in an `outputText` component.
   - **ADF Read-Only Dynamic Table**: The attributes returned and displayed are determined dynamically. 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. For example, if you have a method that returns a polymorphic collection (i.e. `getAnimals()` can return a collection of mammals or a collection of birds), the dynamic table can display the different attributes accordingly.

   Note: You can also drop a collection as an ADF Master Table, Inline Detail Table. For more information, see Section 8.6, "Using an Inline Table to Display Detail Data in a Master Table".

3. From the ensuing Edit Table Columns dialog, you can do the following:

   - Change the display label for a column. By default, the label is bound to the `label` property of the table binding. For more information about this property, see Section B, "Reference ADF Binding Properties". This binding to the property allows you to change the value of the label text once and have it appear the same on all pages that display the label. In this dialog, you can instead enter text or an EL expression to bind the label value to something else, for example, a key in a resource file.

     For example, the headings for the status columns in the table on the SRList page are bound to the label property of the status attribute binding:

     ```
     #{bindings.findServiceRequests1.labels.status}
     ```

     However, you could change the headings to instead be bound to a key in a properties file, for example:

     ```
     #{srlist['sr.status']}
     ```

     In this example, `srlist` would be a variable defined in the JSF page used to load a properties file. For more information about using resource bundles, see Section 14.4, "Internationalizing Your Application".

   - Change the attribute binding for a column.

     For example, you can change the status column to instead be bound to the `requestDate` attribute. Note the following:

     – If you change the binding, the label for the column also changes.

     – If you change the binding to an attribute currently bound to another column, the UI component changes to a component different from that used for the column currently bound to that attribute.

If you simply want to rearrange the columns, you should use the order buttons. See the fourth bullet point below for more information.

- Change the UI component used to display an attribute. The UI components are either `inputText` or `outputText` and are set based on the table you selected when you dropped the collection onto the page. You can change to the other component using the dropdown menu. If you want to use a different component, such as a command link or button, you need to use this dialog to select the `outputText` component, and then in the Structure window, add that other UI component (such as a command link) as a parent to this component.
- Change the order of the columns using the order buttons. **Top** moves the column to the first column at the left of the table. **Up** moves the column one column to the left. **Down** moves the column one to the right. **Bottom** moves the column to the very right.
- Add a column using the **New** button. Doing so adds a new column at the bottom of the dialog and populates it by default with values from the next sequential attribute in the collection. You then need to edit the values. You can only select an attribute associated with the object to which the table is bound.
- Delete a column using the **Delete** button. Doing so deletes the column from the table.
- Add a `tableSelectOne` component to the table's `selection` facet by selecting **Enable selection**. For more information, see Section 7.6, "Enabling Row Selection in a Table".
- Allow sorting for all columns by selecting **Enable sorting**.

Note: If you choose to enable sorting, the table can only sort through the number of objects returned by the iterator, as determined by the iterators `rangeSize` attribute.

7.2.2 What Happens When You Use the Data Control Palette to Create a Table

Dropping a table from the Data Control Palette has the same effect as dropping a text field or form. For more information, see Section 6.2.2, "What Happens When You Use the Data Control Palette to Create a Text Field". 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.

7.2.2.1 Iterator and Value Bindings for Tables

When you drop a table from a the Data Control Palette, a table value binding is created. Like an attribute binding used in forms, the table value binding references the iterator binding. However, instead of creating a separate binding for each attribute, only the table binding is created. This table binding has a child attribute name element for each attribute. Example 7–1 shows the table binding for the table created when you drop the `ServiceRequest` collection.

Example 7–1 Value Binding Entries for a Table in the Page Definition File

```
<table id="findServiceRequest1" IterBinding="findServiceRequestsIter">
  <AttrNames>
    <Item Value="svrId"/>
    <Item Value="status"/>
```

```
    <Item Value="requestDate"/>
    <Item Value="problemDescription"/>
    <Item Value="assignedDate"/>
  </AttrNames>
</table>
```

Only the table value binding is needed because only the table UI component needs access to the data. The table columns derive their information from the table binding.

7.2.2.2 Code on the JSF Page for an ADF Faces Table

When you use the Data Control Palette to drop a table onto a JSF page, JDeveloper creates a table that contains a column for each attribute on the object to which it is bound. To do this, JDeveloper inserts an ADF Faces `table` component. This component contains an ADF Faces `column` component for each attribute named in the table binding. Each column then contains either an `input` or `outputText` component bound to the attribute's value. Each column's heading attribute is bound to the `label` property for each attribute on the table binding. Example 7–2 shows a simplified code excerpt from the table on the SRList page.

Example 7–2 Simple Example of JSF Code for an ADF Faces Table

```
<af:table var="row"
              value="#{bindings.findServiceRequests1.collectionModel}
  <af:column headerText="#{bindings.findServiceRequests1.labels.svrId}"
      <af:outputText value="#{row.svrId}"/>
  </af:column>
  <af:column headerText="#{bindings.findServiceRequests1.labels.status}"
    <af:outputText value="#{row.status}"/>
  </af:column>
         ...
</af:table>
```

An ADF Faces table itself iterates over the data accessed by the iterator binding. In order to do this, the table wraps the result set from the iterator binding in an `oracle.adf.view.faces.model.CollectionModel` object. As the table iterates, it makes each item in the collection available within the `table` component using the `var` attribute.

In the example above, the table iterates over the collection from the `findServiceRequests1` table binding, which in turn references the `findServiceRequestsIter` iterator binding. The iterator binding is what determines the current data object. 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 `outputText` tag:

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

Table 7–1 shows the other attributes defined by default for ADF Faces tables created using the Data Control Palette.

Table 7–1 ADF Faces Table Attributes and Populated Values

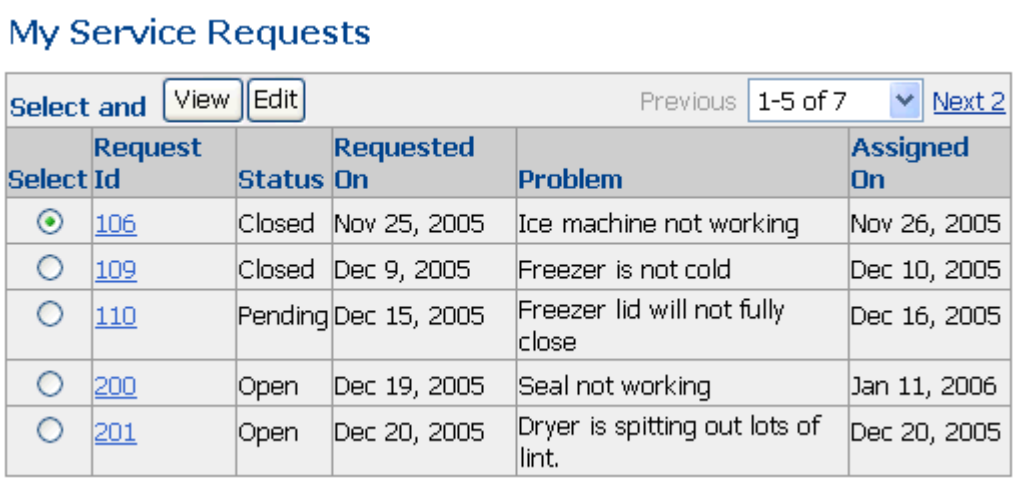
Attribute	Description	Default Value
`rows`	Determines how may rows to display at one time.	An EL expression that evaluates to the `rangeSize` property of the associated iterator binding. For more information on this attribute, see Section 7.3, "Incorporating Range Navigation into Tables". Note that the value of the `rows` attribute is equal to or less than the corresponding iterator's `rangeSize` value.
`first`	Index of the first row in a range (based on 0).	An EL expression that evaluates to the `rangeStart` property of the associated iterator binding. For more information on this attribute, see Section 7.3, "Incorporating Range Navigation into Tables".
`emptyText`	Text to display when there are no rows to return.	An EL expression that evaluates to the viewable property on the iterator. If the table is viewable, displays **No rows yet** when no objects are returned. If the table is not viewable (for example if there are authorization restrictions set against the table), displays **Access Denied**.
Column Attributes		
`sortProperty`	Determines the property on which to sort the column.	Set to the columns corresponding attribute binding value.
`sortable`	Determines whether a column can be sorted	Set to `false`. When set to `true`, the table will sort only the rows returned by the iterator.

Additionally, a table may also have a `selection` facet, and `selection` and `selectionListener` attributes if you chose to enable selection when you created your table. For more information, see Section 7.6, "Enabling Row Selection in a Table".

7.3 Incorporating Range Navigation into Tables

Instead of using built-in operations to perform navigation as forms do, ADF Faces tables provide built-in navigation using the `selectRangeChoiceBar` component that is automatically included with `table` components. The `selectRangeChoiceBar` component renders a dropdown menu and Previous and Next links for selecting a range of records to display in the current page. Figure 7–3 shows an example of how the `selectRangeChoiceBar` component might look like in a table.

Figure 7–3 SelectRangeChoiceBar in a Table



7.3.1 How to Use Navigation Controls in a Table

The `rows` attribute on a `table` component determines the maximum number of rows to display in a range. When you use the Data Control Palette to create a table, by default JDeveloper sets the table to display a range of rows equal to the iterator's `rangeSize` value, as shown in the following code snippet for the `rows` attribute on the SRList table:

```
#{bindings.findServiceRequests1.rangeSize}
```

You can modify the `rows` attribute to display a different range size. For example, you may want the iterator to return 50 records, but you want the table to display only 5 at a time. However, if you plan on displaying the same amount you are retrieving, instead of changing the table's range size, you should keep this attribute bound to the iterator's range size, and then change the iterator. For more information, see Section 6.4.2.2, "Iterator RangeSize Attribute".

To change the table's range size:

1. Select the table in the Structure window.
2. In the Property Inspector, for the `rows` attribute, enter a value for the number of rows to display at a time.

   Alternatively, you can manually set the `rows` attribute in the JSF code:

   ```
   <af:table rows="5">
   ```

> **WARNING:** The value of the `rows` attribute must be equal to or less than the corresponding iterator's `rangeSize` value.

7.3.2 What Happens When You Use Navigation Controls in a Table

The `selectRangeChoiceBar` component provides navigational links that allow a user to select the next and previous range of objects in the collection. If the total size of the collection is known, the component provides a dropdown menu that lets the user navigate directly to a particular range set in the collection (as illustrated in Figure 7–3).

When you change the `RangeSize` attribute on the iterator, the `selectRangeChoiceBar` component automatically changes to show the new range sets.

You use the `rows` attribute on a `table` component in conjunction with the `first` attribute to set the ranges. The `first` attribute determines the current range to display. This attribute is an index (based at zero) of each row in the list. By default, the `rows` attribute uses an EL expression that binds its value to the value of the `rangeSize` attribute of the associated iterator. The `first` attribute uses an EL expression that binds its value to the value of the iterator's `rangeStart` attribute. For example, the `rows` and `first` attribute on the table on the SRList page have the following values:

```
<af:table rows="#{bindings.findServiceRequests1.rangeSize}"
          first="#{bindings.findServiceRequests1.rangeStart}"
```

Each range starts with the row identified by `first`, and contains only as many rows as indicated by the `rows` attribute.

7.3.3 What Happens at Runtime

When the total number of data objects in the collection exceeds the value of the `rows` attribute, the table displays the `selectRangeChoiceBar` component, which allows the user to navigate through the row sets.

Unlike navigation operations which rely on logic in an action binding to provide navigation, the `selectRangeChoiceBar` component sends a `RangeChangeEvent` event. When a user navigates to a different range by selecting one of the navigation links provided by the `selectRangeChoiceBar` component, (such as **Previous** or **Next**), the table generates a `RangeChangeEvent` event. This event includes the index of the object that should now be at the top of the range. The table responds to this event by changing the value of the `first` attribute to this new index.

The `RangeChangeEvent` event has an associated listener. You can bind the `RangeChangeListener` attribute on the table to a method on a managed bean. This method will then be invoked in response to the `RangeChangeEvent` event, in other words whenever the table has changed the `first` attribute in response to the user changing a range on the table. This binding can be helpful when some complementary action needs to happen in response to user navigation, for example, if you need to release cached data created for a previous range. For information about adding logic before or after built-in operations, see Section 10.5, "Overriding Declarative Methods".

7.3.4 What You May Need to Know About the Browser Back Button

Note that using the browser **Back** button has the same issues as described in Chapter 6. For more information, see Section 6.4.4, "What You May Need to Know About the Browser Back Button". Because the iterator keeps track of the current object, when a user clicks a browser's **Back** button instead of using navigation buttons on the page, the iterator becomes out of sync with the page displayed because the iterator has been bypassed. Like in forms, in tables the current row (or range or rows) displayed in the page you see when you use the browser **Back** button may no longer correspond with the iterator binding's notion of the current row and range.

For example, in the SRList page shown in Figure 7–1, if you select the service request with the ID of `4` and then navigate off the page using either the ID's link or the **View** or **Edit** buttons, the iterator is set to the object that represents service request 4.

If you set `EnableTokenValidation` to be `true` (as described in the procedure in Section 6.4.4, "What You May Need to Know About the Browser Back Button"), then the page's token is also set to 4. When you use the browser's **Back** button, everything seems to be fine, the same range is displayed. However, if you click another button, an error indicating that the current row is out of sync is shown. This is because the page displayed is the previous page, whose token was set to 0, while the iterator is at 4.

7.4 Modifying the Attributes Displayed in the Table

Once you use the Data Control Palette 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. Before you add new attributes, make sure the table binding includes the attribute you want to display in the table.

7.4.1 How to Modify the Displayed Attributes

You can modify the following aspects of a table that was created using the Data Control Palette.

- Change the binding for the label of a column
- Change the attribute to which a UI component is bound
- Change the UI component bound to an attribute
- Reorder the columns in the table
- Delete a column in the table
- Add a column to the table

To change the attributes for a table:

1. In the Structure window, right-click **af:table** and choose **Edit Columns**.
2. In the Edit Columns dialog, you can do the following:
   - Change the label for the column. By default, the label is bound to the `label` property of the table binding. For more information about this property, see Appendix B, "Reference ADF Binding Properties". This binding allows you to change the label once and have it appear the same on all pages that display the label. In this dialog, you can instead enter text or an EL expression to bind the label value to something else, for example, a key in a resource file.

     For example, the headings for the status columns in the table on the SRList page are bound to the label property of the status attribute binding:

     ```
     #{bindings.findServiceRequests1.labels.status}
     ```

     However, you could change it to instead be bound to a key in a properties file, for example:

     ```
     #{srlist['sr.status']}
     ```

     In this example, `srlist` would be a variable defined in the JSF page used to load a properties file. For more information about using resource bundles, see Section 14.4, "Internationalizing Your Application".
   - Change the attribute binding for the column.

For example, you can change the status column to instead be bound to the `requestDate` attribute. Note the following:

- If you change the binding, the label for the column also changes.
- If you change the binding to an attribute currently bound to another column, the UI component changes to a component different from that used for the column currently bound to that attribute.

If you simply want to rearrange the columns, you should use the order buttons, as described later in the section.

- Change the UI component used to display the attribute. The UI components are either `inputText` or `outputText` and are set based on the widget you selected when you dropped the collection onto the page. You can change to the other component using the dropdown menu. If you want to use a different component, such as a command link or button, you need to use this dialog to change to an `outputText` component, and then in the Structure window, add that other UI component (such as a command link) as a parent to this component.

  Tip: You can use the following UI components in a table with the noted caveats:

  - The `selectBooleanCheckbox` component can be used inside a table if it is only handling `boolean` or `java.lang.Beoolean` types.
  - The `selectOneList/Choice/Radio` components can be used inside the table if you manually add the list of choices as an enumeration. If instead you want to use a list binding, then the `selectOne` UI component cannot be used inside a table. For more information on list bindings, see Section 11.7, "Creating Databound Dropdown Lists".

- Change the order of the columns using the order buttons. **Top** moves the column to the first column at the left of the table. **Up** moves the column one column to the left. **Down** moves the column one to the right. **Bottom** moves the column to the very right.
- Add a column using the **New** button. Doing so adds a new column at the bottom of the dialog and populates it by default with values from the next sequential attribute in the collection. You then need to edit the values. You can only select an attribute associated with the object to which the table is bound.
- Delete a column using the **Delete** button. Doing so deletes the column from the table.
- Add a `tableSelectOne` component to the table's `selection` facet by selecting **Enable selection**. For more information, see Section 7.6, "Enabling Row Selection in a Table".
- Add sorting capabilities by selecting **Enable sorting**.

Note: If you choose to enable sorting, the table can only sort through the number of objects returned by the iterator, as determined by the iterators `rangeSize` attribute.

7.4.2 How to Change the Binding for a Table

Instead of modifying a binding, you can completely change the object to which the table is bound.

To rebind a table:

1. Right-click the table in the Structure window and choose **Edit Binding** to launch the Table Binding Editor.
2. In the editor, select the new collection to which you want to bind the table. Note that changing the binding for the table will also change the binding for all the columns. You can then use the procedures in Section 7.4.1, "How to Modify the Displayed Attributes" to modify those bindings.

7.4.3 What Happens When You Modify Bindings or Displayed Attributes

When you simply modify how an attribute is displayed, by moving the UI component or changing the UI component, JDeveloper changes the corresponding code on the JSF page. When you use the binding editors to add or change a binding, JDeveloper adds the code to the JSF page, and also adds the appropriate elements to the page definition file.

7.5 Adding Hidden Capabilities to a Table

You can use the `detailStamp` facet in a table to include data that can be displayed or hidden. When you add a component to this facet, the table displays an additional column labeled **Details** with a toggle. When the user activates the toggle, the component added to the facet is shown. When the user clicks on the toggle again, the component is hidden. For more information about facets in general, see Section 6.3.2.1, "Using Facets". Figure 7–4 shows how the description of a service request in an `outputText` component can be hidden or shown in the table (note that this functionality does not currently exist in the SRDemo application).

Figure 7–4 Table with an Output UI Component in the DetailStamp Facet

If you wish to show details of another object that has a master-detail relationship (for example, if you wanted to show the details of the person to whom the service request is assigned), you could use the `Master Table-Inline Detail` composite component. For more information about master-detail relationships and the use of the master-detail composite component, see Section 8.6, "Using an Inline Table to Display Detail Data in a Master Table".

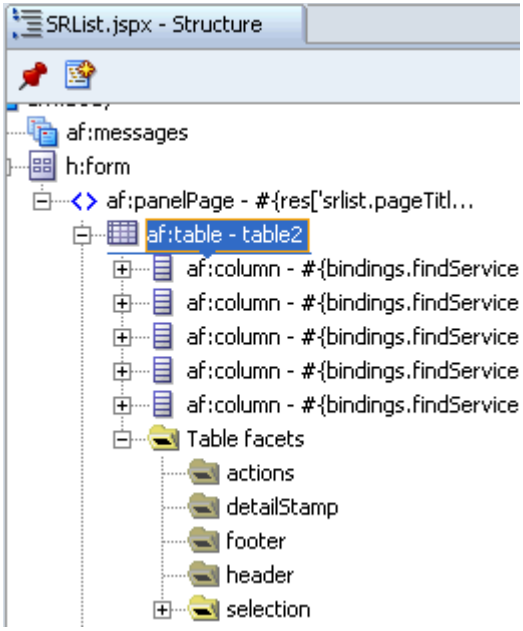
7.5.1 How to Use the DetailStamp Facet

To use the `detailStamp` facet, you insert a component that is bound to the data to be displayed or hidden into the facet. You can also set an attribute on the table that creates a link that allows a user to show or hide all details at once.

To use the detailStamp facet:

1. Drag the attribute to be displayed in the facet from the Data Control Palette onto the **detailStamp** facet folder. Figure 7–5 shows how the **detailStamp** facet folder appears in the Structure window.

Figure 7–5 The detailStamp Facet Folder in the Structure Window



2. From the ensuing context menu, choose the UI component to display the attribute.
3. If you want a link to allow users to hide or show all details at once, select the table in the Structure window. Then in the Property Inspector, set the `allDetailsEnabled` attribute to `true`.
4. If the attribute to be displayed is specific to a current record, then you need to replace the JSF code (which simply binds the component to the attribute), so that it uses the table's variable to display the data for the current record.

   For example, when you drag an attribute, JDeveloper inserts the following code:

```
<f:facet name="detailStamp">
  <af:outputText value="#{bindings.<attributename>.inputValue}"/>
</f:facet>
```

You need to change it to the following:

```
<f:facet name="detailStamp">
  <af:outputText value="#{row.<attributename>}"/>
</f:facet>
```

7.5.2 What Happens When You Use the DetailStamp Facet

When you drag an attribute in the `detailStamp` facet folder, JDeveloper adds the attribute value binding to the page definition file if it did not already exist, and it also adds the code for facet to the JSF Page.

For example, say on the SRList page you want the user to be able to optionally hide the service request description as shown in Figure 7–4. Since the table was created using the `findServiceRequest(Integer, String) method`, you can drag the `problemDescription` attribute and drop it inside the **detailStamp** facet folder in the Structure window.

Example 7–3 shows the code JDeveloper then adds to the JSF page.

Example 7–3 JSF Code for a detailStamp Facet

```
<f:facet name="detailStamp">
  <af:outputText value="#{bindings.problemDescription.inputValue}"
                 id="outputText7"/>
</f:facet>
```

You then need to change the code so that the component uses the table's variable to access the correct problem description for each row. Example 7–4 shows how the code should appear after using the row variable.

Example 7–4 Modified JSF Code for a detailStamp Facet

```
<f:facet name="detailStamp">
  <af:outputText value="#{row.problemDescription}"
                  id="outputText7"/>
</f:facet>
```

7.5.3 What Happens at Runtime

When the user hides or shows the details of a row, the table generates a `DisclosureEvent` event (or a `DisclosureAllEvent` event when the `allDetailsEnabled` attribute on the table is set to `true`). The event tells the table to toggle the details (that is, either expand or collapse).

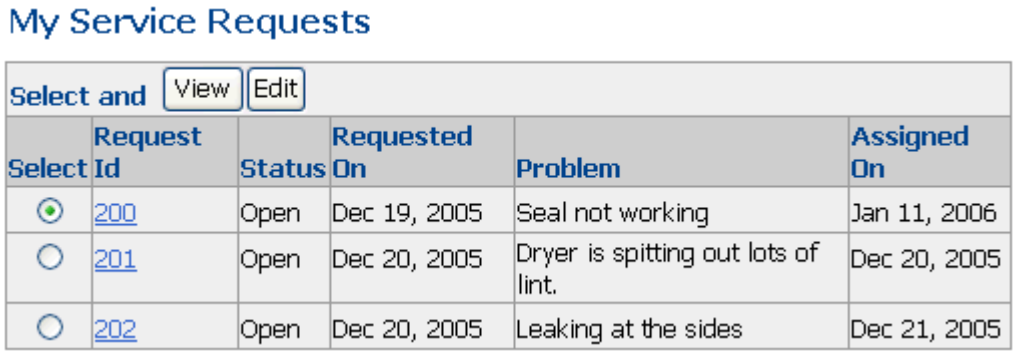
The `DisclosureEvent` event has an associated listener. You can bind the `DisclosureListener` attribute on the table to a method on a managed bean. This method will then be invoked in response to the `DisclosureEvent` event to execute any needed post-processing.

7.6 Enabling Row Selection in a Table

When the `tableSelectOne` component or the `tableSelectMany` component is added to the table's `selection` facet, the table displays a **Select** column that allows a user to select one row, or one or more rows, and then take some action on those rows via command buttons.

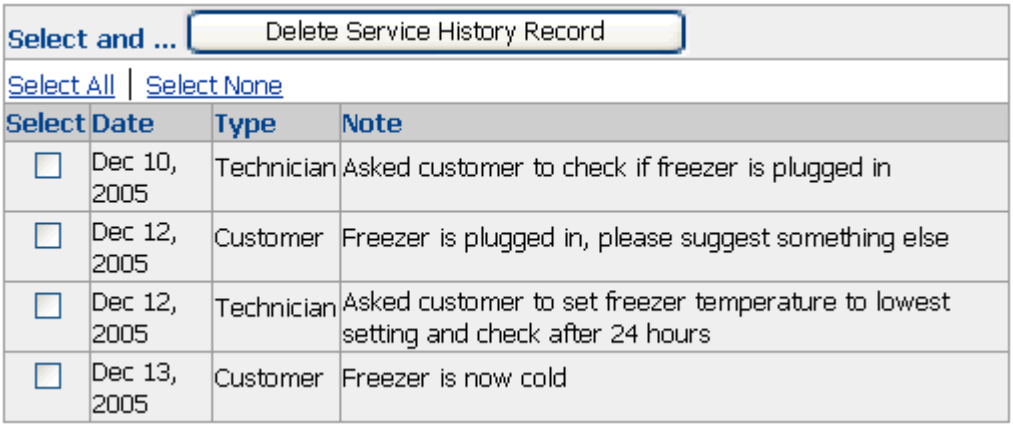
The `tableSelectOne` component allows the user to select just one row. This component provides a radio button for each row in the **Select** column, as shown in Figure 7–6. For example, the table in the SRList page has a `tableSelectOne` component that allows a user to select a row, and then click either the **View** or **Edit** command button to view or edit the details for the selected service request.

Figure 7–6 The SRList Table Uses the TableSelectOne Component



The `tableSelectMany` component displays a checkbox for each row in the **Select** column, allowing the user to select one or more rows. When you use the `tableSelectMany` component, text links are also added that allow the user to select all or none of the rows, as shown in Figure 7–7. For example, the table on the SRMain page has a `tableSelectMany` component that allows a user to select multiple records, and then click the **Delete Service History Record** command button to delete the selected records.

Figure 7–7 The Service History Table Uses the TableSelectMany Component



Both table row selection components have a `text` attribute whose value can be instructions for the user. The table row selection components also usually have command button or command links as children, which are used to perform some action on the selected rows. For example, the table on the SRList page has command buttons that allows a user to view or edit the selected service request.

You can set the `required` attribute on both the `tableSelectOne` and the `tableSelectMany` components to `true`. This value will cause an error to be thrown if the user does not select a row. However, if you set the `required` attribute, you must also set the `summary` attribute on the table in order for the required input error message to display correctly. For more information about the `required` attribute, see Section 12.3.1.1.1, "Using Validation Attributes".

You can also set the `autoSubmit` attribute on the `tableSelectOne` and the `tableSelectMany` components. When the `autoSubmit` attribute is set to `true`, the form that holds the table automatically submits when the user makes a selection. For more information, see Section 4.6, "Best Practices for ADF Faces".

The procedures for using the `tableSelectOne` and `tableSelectMany` are quite different. In ADF applications, operations (such as methods) work on the current data object, which the iterator keeps track of. The `tableSelectOne` component is able to show the current data object as being selected, and is also able to set a newly selected row to the current object on the iterator. If the same iterator is used on a subsequent page (for example, if the user selects a row and then clicks the command button to navigate to a page where the object can be edited), the selected object will be displayed. This works because the iterator and the component are working with a single object; the notion of the current row is the same because the different iterator bindings in different binding containers are bound to the same row set iterator.

However, with the `tableSelectMany` component, there are multiple selected objects. The ADF Model layer has no notion of "selected" as opposed to "current." You must add logic to the model layer that loops through each of the selected objects, making each in turn current, so that the operation can be executed against that object.

Instead of using the selection facet components to set the current object and then providing a `commandButton` to navigate to the next page, you can use a `commandLink` component that allows the user to click a link to both perform an operation on a selection and navigate to another page, which saves the user the step of having to first select a row and then click the command button to perform the action and navigate. However, you must then manually set the current object on the iterator binding. For more information about manually setting the current object, see Section 7.7, "Setting the Current Object Using a Command Component".

> **Tip:** If the subsequent page does not use the same iterator, you will most likely have to set the parameter that represents the selected row for the subsequent page manually. For example, in the SRDemo application, the form on the SREdit page is created using the `findServiceRequestById(Integer)` method. An `Integer` that represents the ID for the selected row must be passed to that method in order for the form to properly display. If the parameter is not set, the form displays the first row in the iterator. For more information, see Section 10.4, "Setting Parameter Values Using a Command Component".

7.6.1 How to Use the TableSelectOne Component in the Selection Facet

When you drop a collection from the Data Control Palette as a table, you have the choice to include the `selection` facet. If you select **Enable selection**, a `tableSelectOne` component is inserted into the `selection` facet, along with a **Submit** `commandButton` component as a child of `tableSelectOne`.

> **Note:** You cannot insert a `tableSelectMany` component when you create a table using the Data Control Palette. You need to manually add it after creating the table. Note however, that you must create additional code in order to use multi-select processing in an ADF application. For more information, see Section 7.6.4, "How to Use the TableSelectMany Component in the Selection Facet".

If you wish to have the **Submit** button bound to a method, you need to rebind the `commandButton` component to the method or operation of your choice. For rebinding procedures, see Section 13.6, "Adding ADF Bindings to Actions".

You can also manually add a `tableSelectOne` component to a `selection` facet.

To manually use the selection facet:

1. In the Structure window, select **af:table** and choose **Edit Columns** from the context menu.
2. In the Edit Table Columns dialog, select **Enable selection** and click **OK**.

   JDeveloper adds the `tableSelectOne` component to the **selection** facet folder (plus the needed listener and attribute that work with selection on the `table` component).
3. In the Structure window, expand the table's **selection** facet folder and select **af:tableSelectOne**.
4. In the Property Inspector for the new component, enter a value for the `text` attribute that will provide instructions for using any command buttons or links used to process the selection.
5. (Optional): Rebind the **Submit** command button to a method or operation of your choice from the Data Control Palette. For rebinding procedures, see Section 13.6, "Adding ADF Bindings to Actions". For more information about using methods to create command buttons, see Section 10.3, "Creating Command Components to Execute Methods".

> **Note:** Until you add a command component to the facet, the value for the `text` attribute will not display.

7.6.2 What Happens When You Use the TableSelectOne Component

As Example 7–5 shows, when you elect to enable selection as you first create or later edit a table, the `tableSelectOne` component is inserted into the `selection` facet with `Select and` as the value for the `text` attribute. A **Submit** command button is also included as a child.

Example 7–5 Selection Facet Code

```
<f:facet name="selection">
  <af:tableSelectOne text="Select and">
    <af:commandButton text="Submit"/>
</af:tableSelectOne>
```

As Example 7–6 shows, the table's `selectionState` attribute's value is an EL expression that evaluates to the selected row on the collection model created from the iterator. The `selectionListener` attribute's value evaluates to the `makeCurrent` method on the collection model. This value is what allows the component to set the selected row as the current object on the iterator.

Example 7–6 Table Selection Attributes

```
<af:table rows="#{bindings.findServiceRequests1.rangeSize}"
          first="#{bindings.findServiceRequests1.rangeStart}"
          var="row"
    selectionState="#{bindings.findServiceRequests1.collectionModel.selectedRow}"
  selectionListener="#{bindings.findServiceRequests1.collectionModel.makeCurrent}"
          id="table2">
```

7.6.3 What Happens at Runtime

Once the user makes a selection and clicks the associated command button, the `tableSelectOne` component updates the `RowKeySet` obtained by calling the `getSelectionState()` method on the table. Since the selection state evaluates to the selected row on the collection model, that row is marked as selected. This selection is done prior to calling the `ActionListener` associated with the command button.

For a `tableSelectOne` component, because the current row is selected before the `ActionListener` is invoked, you can bind the `ActionListener` on the command button to a method on a managed bean that provides the corresponding processing on the data in the row. Or you can simply add the logic to the declarative method. For more information, see Section 10.5, "Overriding Declarative Methods".

The `tableSelectOne` component triggers a `SelectionEvent` event when the selection state of the table is changed. The `SelectionEvent` reports which rows were selected and deselected. Because the `SelectionListener` attribute is bound to the `makeCurrent` method on the collection model, this method is invoked when the event occurs, and sets the iterator to the new current row.

7.6.4 How to Use the TableSelectMany Component in the Selection Facet

When you add the `tableSelectMany` component to a table that uses an ADF table binding, you must also add code that sets each selected row in turn to the current object so that the operation can be performed against that object.

To use the tableSelectMany component in an ADF application:

1. Create the table as shown in Section 7.2.1, "How to Create a Basic Table" but *do not* select **Enable selection**.
2. In the Structure window, expand the **Table facets** folder, right-click the **selection** facet folder, and choose **Insert inside selection** > **TableSelectMany**.
3. In the Structure window, select the **af:table** node and in the Property Inspector, delete the values for the **SelectionState** and **SelectionListener** attributes, if necessary. Doing so will keep the component from setting one of the selected rows to the current object, as you need this logic to be handled through the code you create.
4. From the Data Control Palette, drag the method that will operate on the selected object on top of the **af:tableSelectMany** node. From the ensuing context menu, choose **Method**s > **Command Button**. Doing so drops the method as a command button. You now need to set the parameter value (if needed) for the method. For those procedures, see Section 10.3.1, "How to Create a Command Component Bound to a Service Method".

   For example, if you were working in the SRDemo application and wanted the user to be able to delete the selected rows, you would drag the `removeEntity(Object)` method onto the **af:tableSelectMany** node.

   You must now add logic to the method that allows the method to operate against a set of rows, making each row current in turn. To add the logic, you need to override the declarative method created when dropping the command button. For those procedures, see Section 10.5, "Overriding Declarative Methods".

   This code allows you to override the `removeEntity(Object)` method and add the needed logic.
5. Add logic to the declarative method that does the following:
   - Accesses the table component
   - Obtains a list of all selected rows
   - Gets the objects in turn and performs the original method on each. To do this, the logic must loop through the list of selected rows as follows:
     - Get a row in the loop
     - Get the key for the row
     - Set it as the current object in the ADF binding
     - Delete the object by calling the declarative method

   Once that is done, logic should be added that refreshes the iterator, so that it displays the correct set of objects. For a code example, see Example 7–10.

7.6.5 What Happens When You Use the TableSelectMany Component

When you insert the `tableSelectMany` component into a table, and then add a command button bound to a service method, JDeveloper does the following:

- Adds the `tableSelectMany` and `commandButton` components to the `selection` facet on the `table` component
- Creates a method binding for the bound method in the page definition file, including a `NamedData` element to hold the value of the parameter needed for the method (if any), determined when you dropped the method as a button

You then need to override the method and add logic that accesses each selected row in the table and executes the method on that current row.

For example, say you create a table that shows all products using the `findAllProduct()` method. You then add a `tableSelectMany` component so that a user can select the products to delete using the `removeEntity(Object)` method. Example 7–7 shows the code on the JSF page.

Example 7–7 JSF Code for a Table That Uses the tableSelectManyComponent

```
<af:table value="#{bindings.findAllProducts1.collectionModel}"
                  var="row" rows="#{bindings.findAllProducts1.rangeSize}"
                  first="#{bindings.findAllProducts1.rangeStart}"
                  id="table1">
  <af:column>
     ...
  </af:column>
  <f:facet name="selection">
    <af:tableSelectMany text="Select items and ..."
                        id="tableSelectMany1">
      <af:commandButton text="removeEntity"
                        disabled="#{!bindings.removeEntity.enabled}"
                        id="commandButton1"
                        action="#{backing_MultiDelete.commandButton1_action}"/>
    </af:tableSelectMany>
  </f:facet>
</af:table>
```

JDeveloper adds code to the page definition that binds the parameter value for the object in the `removeEntity(Object)` method to the current row of the table, as shown in Example 7–8.

Example 7–8 Method Action Binding for a Method whose Parameter is the Current Row in a Table Binding

```
<methodAction id="removeEntity" InstanceName="SRPublicFacade.dataProvider"
                  DataControl="SRPublicFacade" MethodName="removeEntity"
                  RequiresUpdateModel="true" Action="999">
  <NamedData NDName="entity"
                NDValue="${bindings.findAllProducts1.currentRow.dataProvider}"
              NDType="java.lang.Object"/>
</methodAction>
<table id="findAllProducts1" IterBinding="findAllProductsIter">
  <AttrNames>
     ...
  </AttrNames>
</table>
```

To add logic to a declarative method, you double-click the button in the visual editor, and JDeveloper adds code to the associated backing bean that can access the method logic.

For example, if you drop the `removeEntity(Object)` method from the SRDemo application into the facet, and then double-click the **removeEntity** button in the visual editor, JDeveloper adds the code shown in Example 7–9 to the associated backing bean.

Example 7–9 Backing Bean Code for a Declarative Method

```
public String commandButton1_action() {
  BindingContainer bindings = getBindings();
  OperationBinding operationBinding =
     bindings.getOperationBinding("removeEntity");
  Object result = operationBinding.execute();
  if (!operationBinding.getErrors().isEmpty())
    return null;
  return null;
}
```

You then add code that accesses each selected row before the generated code. You use the generated code to execute the method on the object for that row. You then add code after the generated code to reexecute the query and refresh the page.

For example, say you want to allow users to delete rows of products by selecting the products and then deleting them using a command button bound to the `removeEntity(Object)` method. You would add the declarative code to a backing bean by double-clicking the button. You would then add code shown in bold in Example 7–10 to delete the objects. Code not in bold font is the code generated by JDeveloper, as shown in Example 7–9.

Example 7–10 Complete Backing Bean Code to Allow tableSelectMany

```
public String commandButton1_action() {

  //Access the tableSelectMany1 table. Note that the table name
  //is taken from the id of the table in the JSF page.
  CoreTable table = this.getTable1();

  //Obtain a list of all selected rows from the table
  Set rowSet = table.getSelectionState().getKeySet();
  Iterator rowSetIter = rowSet.iterator();

  //Use the declarative method to get the ADF bindings
  BindingContainer bindings = getBindings();

  //Get the object to delete. To do this, you must get the
  //iterator binding for the Products in the page definition file,
  //and cast it to DCIteratorBinding for further processing
  DCIteratorBinding pr_dcib = (DCIteratorBinding)
     bindings.get ("findAllProductsIter");

  //Loop through the set of selected row numbers and delete the
  //equivalent object from the Products collection.
  while (rowSetIter.hasNext()){
    //get the table row
    Key key = (Key) rowSetIter.next();
```

```
    //set the current row in the ADF binding to the same row
    pr_dcib.setCurrentRowWithKey(key.toStringFormat(true));

    //Obtain the Products object to delete
    RowImpl prRow = (RowImpl) pr_dcib.getCurrentRow();

   //Delete the object by first accessing the data and then
   //using the generated code to execute the declarative method
   Products prObjectToDelete = (Products) prRow.getDataProvider();
   OperationBinding operationBinding =
     bindings.getOperationBinding("removeEntity");

  //You don't need to set the parameter, as this was done
  //declaritively when you dropped the button on the page
  Object result = operationBinding.execute();
  if (!operationBinding.getErrors().isEmpty())
     return null;
  }

//Re-execute the query to refresh the screen
OperationBinding requery = bindings.getOperationBinding("findAllProducts");
requery.execute();

//Stay on the same page, so no returned outcome needed
return ];

}
```

7.6.6 What Happens at Runtime

When the user selects multiple rows and then clicks the command button, the application accesses the table to determine each of the selected rows, and creates a rowset for those rows. The application then accesses the binding container, and from that container, accesses the iterator used to manage the complete collection and casts it to a generic iterator binding that can manage the rowset of selected rows.

That iterator then goes through each row, and for each row:

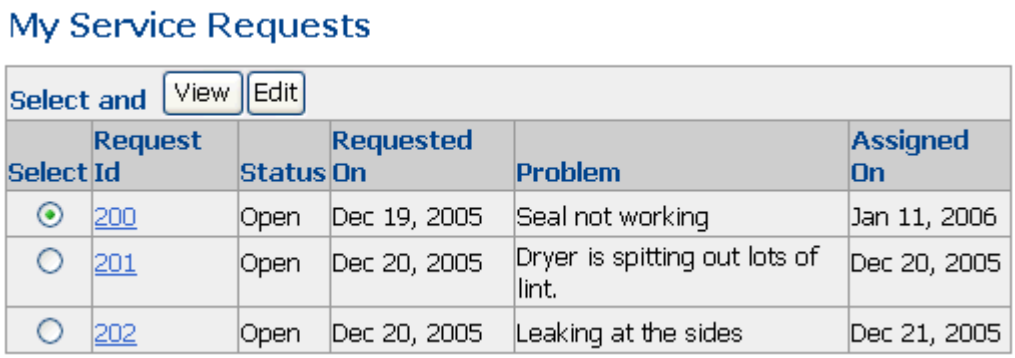
- Sets a key
- Uses that key to set the row to the current row in the iterator, using the `setCurrentRowWithKey` operation, as described in Table 6–1, " Built-in Navigation Operations"
- Uses the current row to create the object against which the method will be executed
- Accesses the associated data for the object
- Executes the method

Once that is complete, and there are no more rows in the rowset, the application accesses the iterator in the binding container and reexecutes the query to refresh the set of rows displayed in the table.

7.7 Setting the Current Object Using a Command Component

There may be cases where you need to programmatically set the current row for an object on an iterator. For example, the SRList page in the SRDemo application uses command links in the second column, as shown in Figure 7–8, which the user can click to directly edit a service request, without needing to first select the row.

Figure 7–8 Command Links Used in a Table on the SRList Page



While using command links saves a step for the user, command links do not offer the same functionality as the `selection` facet, in that they can neither determine nor set the current row on the iterator. Therefore, you must manually set the current row.

7.7.1 How to Manually Set the Current Row

You use the `setCurrentRowWithKey` or `setCurrentRowWithKeyValue` built-in operations to set the current row. These operations are built-in methods on any iterator for a collection. The `setCurrentRowWithKey` operation allows you to set the current row given "stringified" key. The `setCurrentRowWithKeyValue` operation allows you to set the current row given the a primary key's value. For more information about the current row operations, see Section 10.5.6, "Understanding the Difference Between setCurrentRowWithKey and setCurrentRowWithKeyValue".

While you can drop these operations as any type of command component, the `commandLink` component is most usually used in this situation. The following procedure explains how to use this component with the `setCurrentRowWithKey` and `setCurrentRowWithKeyValue` operations.

To set the current row:

1. From the Data Control Palette, drag the `setCurrentRowWithKey` or `setCurrentRowWithKeyValue` operation.
2. From the context menu, choose **Operations** > **ADF Command Link**.
3. In the Action Binding Editor, you need to set the value for the `rowKey` parameter. By default, it is set to `${bindings.setCurrentRowWithKey_rowKey}`. The actual value should be something that can be used to determine the current row.

   For example, the command link in Figure 7–8 needs to set the current row to the same row as the link being clicked. To access the "stringified" key of the row for the `setCurrentRowWithKey` operation, you can use the `rowKeyStr` property on the binding, or `#{row.rowKeyStr}`.

Alternatively, if you use the `setCurrentRowWithKeyValue` operation, you might set the `rowKey` to the value of the current row, or `#{row.svrId}`

For more information about the variable used to set the current row on a table (in this case, `row`), see Section 7.2.2.2, "Code on the JSF Page for an ADF Faces Table".

7.7.2 What Happens When You Set the Current Row

When you use the `setCurrentRowWithKey` operation as a command component, JDeveloper creates an action binding for that operation. Because this operation takes a parameter (`rowKey`) to determine the current row, it has a `NamedData` element used to set that value (for more information about parameters and the `NamedData` element, see Section 10.3, "Creating Command Components to Execute Methods").

Example 7–11 shows the code on the page definition file created when you drop the `setCurrentRowWithKey` operation and set `#{row.svrId}` as the value for the `rowKey` parameter.

Example 7–11 Page Definition Code for the `setCurrentRowWithKey` Operation

```
<action id="setCurrentRowWithKey" IterBinding="findServiceRequestsIter"
            InstanceName="SRPublicFacade.dataProvider"
            DataControl="SRPublicFacade" RequiresUpdateModel="false"
            Action="96">
  <NamedData NDName="rowKey" NDValue="${row.rowKeyStr}"
             NDType="java.lang.String"/>
</action>
```

7.7.3 What Happens At Runtime

When a user clicks the command link, the `setCurrentRowWithKey` operation is executed on the iterator, using the `rowKey` parameter to determine the current row. As with the `tableSelectOne` component, if you use the same iterator to display the current record, the correct data will display.

> **Tip:** For functionality similar to that in the SRDemo application, you may need your command link to pass a parameter value that represents the current row. This value might be used by the method used to create the ensuing form. For more information and procedures, see Section 10.4, "Setting Parameter Values Using a Command Component".

8

Displaying Master-Detail Data

This chapter describes how to create various types of pages that display master-detail related data.

This chapter includes the following sections:

- Section 8.1, "Introduction to Displaying Master-Detail Data"
- Section 8.2, "Identifying Master-Detail Objects on the Data Control Palette"
- Section 8.3, "Using Tables and Forms to Display Master-Detail Objects"
- Section 8.4, "Using Trees to Display Master-Detail Objects"
- Section 8.5, "Using Tree Tables to Display Master-Detail Objects"
- Section 8.6, "Using an Inline Table to Display Detail Data in a Master Table"

For information about using a selection list to populate a collection with a key value from a related master or detail collection, see Section 11.7, "Creating Databound Dropdown Lists".

8.1 Introduction to Displaying Master-Detail Data

In ADF, a master-detail relationship refers to two data objects in the data control hierarchy that are logically related in such a way that an instance of one object automatically contains a related instance of the other object. For example, in the SRDemo application, when a data control method returns a collection of service requests, each service-request object contains a list of related service-history objects. The service-history objects are returned by an accessor that is a child of the parent method in the data control hierarchy. Usually, a master-detail relationship in the data control is established by one or more unique attributes that both objects share or by an object hierarchy. For example, in the SRDemo application the `serviceRequest` collection and the `serviceHistoryCollection` have a master-detail relationship, because both collections contain the `svrId` attribute (the service request number). You can also have master-detail relationships between collections and single objects. For example, each object in a collection of service requests could contain a single user object to which that service request is assigned.

> **Tip:** In TopLink and traditional relational databases master-detail relationships are called *foreign-key* relationships.

When objects have a master-detail relationship, you can declaratively create pages that display the data from both objects simultaneously. For example, the SRDemo application has a page that displays a service request in a form at the top of the page and its related service history in a table at the bottom of the page.

This is possible because the service request and service history objects have a master-detail relationship. In this example, the service request is the master object and the service history is the detail object. The ADF iterators automatically manage the synchronization of the detail data objects displayed for a selected master data object.

Read this chapter to understand:

- Master-detail relationships in ADF
- How to identify master-detail objects on the Data Control Palette
- How to display master-detail objects in tables, forms, trees, tree tables, and inline tables
- How to display master-detail objects on different pages that are connected by a navigation component
- How ADF iterators manage the concurrency of master and detail objects
- The binding objects created when you use the Data Control Palette to create a master-detail UI component

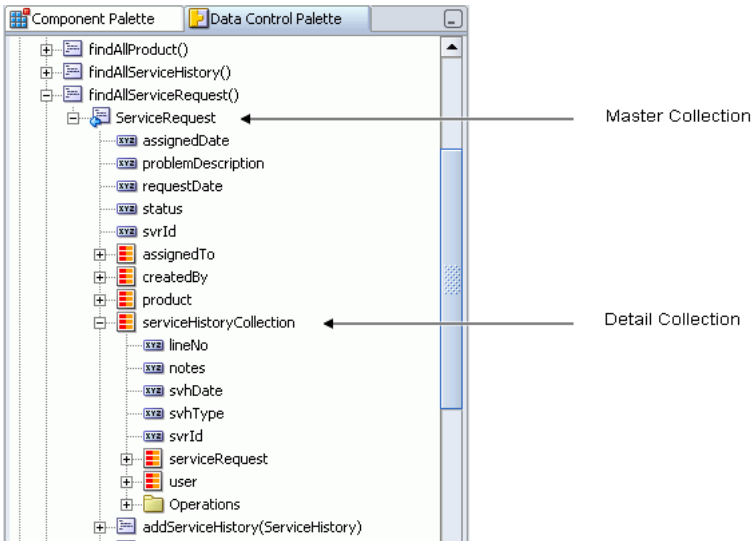
8.2 Identifying Master-Detail Objects on the Data Control Palette

JDeveloper enables you to declaratively create master-detail pages using the Data Control Palette. The Data Control Palette displays master-detail related objects in a hierarchy, where the detail object is displayed as an accessor return under the master object. In the data control, accessor returns are always detail objects in a master-detail relationship.

Figure 8–1 shows the Data Control Palette for the SRDemo application. Because the `serviceHistory` and `ServiceRequest` objects have a master-detail relationship, the accessor return `serviceHistoryCollection` appears under the `ServiceRequest` method return. In this case, the accessor return is a collection of service history objects related to a service request object. Method returns are always collections, but accessor returns can be either collections or single objects.

> **Tip:** By default, when data controls are created from TopLink POJOs (or session beans over POJOs), the names of accessor returns that are collections end in `Collection`. For example, `serviceHistoryCollection`.

Figure 8–1 Master-Detail Objects on the Data Control Palette



> **Tip:** The master-detail hierarchy displayed in the Data Control Palette does not reflect the cardinality of the relationship (for example, one-to-many, one-to-one, many-to-many). The hierarchy simply shows which collection (the master) is being use to retrieve one or more objects from another collection (the detail).

When creating a page that displays master-detail objects, be sure to correctly identify which object is the master and which is the detail for your particular purposes. Otherwise, you may not display the desired data on the page.

For example, if you want to display a user and all the related expertise areas to which the user is assigned, then `User` would be the master object. However, if you wanted to display an expertise area and all the users is assigned to it, then `expertiseArea` would be the master object. The detail objects displayed on a page depend on which object is the master.

> **Tip:** In the Data Control Palette, the attributes shared by both the master and detail objects appear under only one of the objects, not both. For example, in the SRDemo application Data Control Palette, the `srvId` attribute appears under the `ServiceRequest` master node, but not the `serviceHistoryCollection` detail node.
>
> Also, in some cases, the master collection appears as an accessor return under a detail collection. For example, in Figure 8–1, `ServiceRequest`, which is a master collection, appears as an accessor return under the `serviceHistoryCollection` node, which is a detail collection. In this case, the common attribute shared by these collections creates a recursive relationship in the data control. In most cases, you would never use the accessor return that appears as a result of such a recursion to create a UI component.

For more information about the icons displayed on the Data Control Palette, see Section 5.2.1, "How to Understand the Items on the Data Control Palette".

8.3 Using Tables and Forms to Display Master-Detail Objects

JDeveloper enables you to create a master-detail browse page in a single declarative action using the Data Control Palette—you do not need to write any extra code, even the navigation is included. The Data Control Palette provides pre-built master-detail widgets that display both the master and detail objects on the same page as any combination of read-only tables and forms. All you have to do is drop the detail collection on the page and choose the type of widget you want to use.

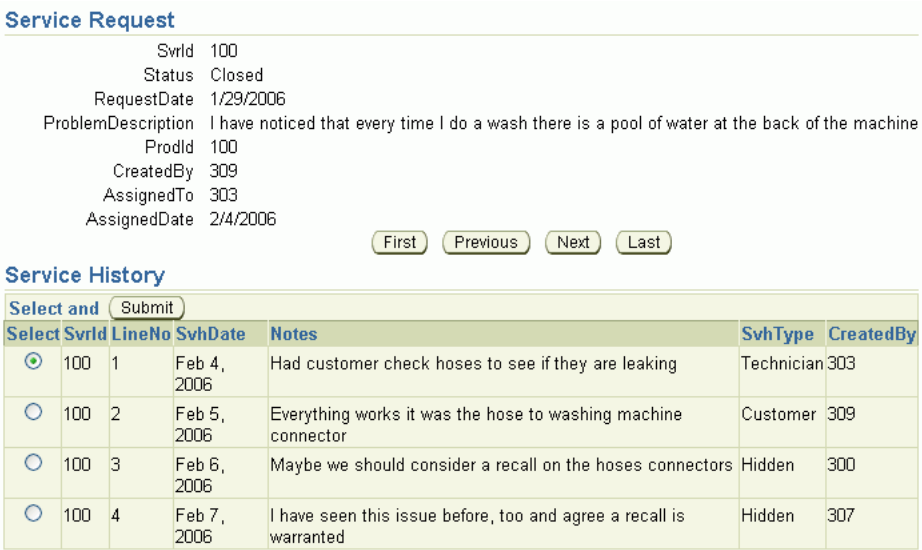
The pre-built master-detail widgets available from the Data Control Palette include range navigation that enables the user to scroll through the data objects in collections. The the table provided by the pre-built master-detail widgets includes a selection facet and **Submit** command button. By default, all attributes of the master and detail objects are included in the master-detail widgets as text fields (in forms) or columns (in tables). You can delete unwanted attributes by removing the text field or column from the page.

> **Tip:** If you do not want to use the pre-built master-detail widgets, you can drag and drop the master and detail objects individually as tables and forms on a single page or on separate pages. For more information about creating individual forms and tables, see Chapter 6, "Creating a Basic Page" or Chapter 7, "Adding Tables".

When you add master-detail components to a page, the iterator bindings are responsible for exposing data to the components on the page. The iterator bindings bind to the underlying rowset iterators. The rowset iterator for the detail object is responsible for exposing the correct detail data when a specific master object is displayed or selected on the page.

Figure 8–2 shows an example of a pre-built master-detail widget, which display a service request in a form at the top of the page and all the related service history in a table at the bottom of the page. When the user scrolls through the master data, the page automatically displays the related detail data.

Figure 8–2 Pre-Built Data Control Palette Master-Detail Widget



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

The Data Control Palette enables you to create both the master and detail widgets on one page with a single declarative action using pre-built master-detail forms and tables. For information about displaying master and detail data on separate pages, see Section 8.3.4, "What You May Need to Know About Master-Detail on Separate Pages".

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

1. From the Data Control Palette, locate the detail object, as was previously described in Section 8.2, "Identifying Master-Detail Objects on the Data Control Palette".
2. Drag and drop the detail object onto the JSF page.
3. In the context menu, choose one of the following **Master-Details** widgets:
   - **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 objects.
   - **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.

     This widget is available only when both the master and detail objects are collections.
   - **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 are displayed in the table below it.

     This widget is available only when both the master and detail objects are collections.

Note: If an object is not a collection, but rather just a single data item, JDeveloper automatically excludes the range navigation from the default widget.

Also, accessor returns can be collections or single objects. Single objects can be displayed only in forms. Consequently, the master-detail widgets available from the Data Control Palette context menu differ depending on whether the accessor return is a collection or a single object.

If you want to modify the default forms or tables, see Chapter 6, "Creating a Basic Page" or Chapter 7, "Adding Tables".

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

When you drag and drop from the Data Control Palette, JDeveloper does many things for you, including adding code to the JSF page and corresponding entries in the page definition file. For a full description of what happens and what is created when you use the Data Control Palette, see Section 5.2.3, "What Happens When You Use the Data Control Palette".

8.3.2.1 Code Generated in the JSF Page

The JSF code generated for a pre-built master-detail widget is basically the same as the JSF code generated when you use the Data Control Palette to create a basic read-only table or form. For more information, see Chapter 6, "Creating a Basic Page" and Chapter 7, "Adding Tables". If you are building your own master-detail widgets, you might want to consider including similar components that are automatically included in the pre-built master-detail tables and forms.

The tables and forms in the pre-built 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 pre-built 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` 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 would automatically omit the range navigation components.) For more information about range navigation, see Section 6.4, "Incorporating Range Navigation into Forms".

By default, tables in a pre-built master-detail widget include a `tableSelectOne` selection facet and a **Submit** button that enables the user to select a specific object in the collection. The default button is not automatically bound to a method or operation. So to get the selection facet to work, you would need to add an action binding to the button. For example, you could bind the button to a method that enables the user to edit the selected data object, as was done in the SRMain page of the SRDemo application. For more information about selection facets, see Section 10.3, "Creating Command Components to Execute Methods".

> **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, `panelForm` 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 re-rendered 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 re-rendered when the user simply makes a selection in the facet: only the detail component needs to be re-rendered to display the new data. For more information about partial rendering, see Section 11.4, "Enabling Partial Page Rendering".

8.3.2.2 Binding Objects Defined in the Page Definition File

Example 8–1 shows the page definition file created for a master-detail page that was created by dropping the `serviceHistoryCollection` accessor return, which is a detail collection under the `ServiceRequest` method return, on the page as an **ADF Master Form, Detail Table**.

The `executables` element defines a method iterator for the service requests (which is the master object) and an accessor iterator for the service history (which is the detail object). The accessor iterator contains a `MasterBinding` attribute, which references the method iterator for the master object. This reference to the master iterator enables the detail iterator to expose the correct detail data for the current master object (for more information, see Section 8.3.3, "What Happens at Runtime").

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

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

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

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

For more information about the elements and attributes of the page definition file, see Section A.7, "<pageName>PageDef.xml".

Example 8–1 Binding Objects Defined in the Page Definition for a Master-Detail Page

```
<executables>
    <methodIterator id="findAllServiceRequestIter"
                    Binds="findAllServiceRequest.result"
                    DataControl="SRPublicFacade" RangeSize="10"
                    BeanClass="oracle.srdemo.model.ServiceRequest"/>
    <accessorIterator id="serviceHistoryCollectionIterator" RangeSize="10"
                      Binds="serviceHistoryCollection"
                      DataControl="SRPublicFacade"
                      BeanClass="oracle.srdemo.model.ServiceHistory"
                      MasterBinding="findAllServiceRequestIter"/>
</executables>
<bindings>
    <methodAction id="findAllServiceRequest"
                  InstanceName="SRPublicFacade.dataProvider"
                  DataControl="SRPublicFacade"
                  MethodName="findAllServiceRequest" RequiresUpdateModel="true"
                  Action="999"
                  ReturnName="SRPublicFacade.methodResults.SRPublicFacade_
                              dataProvider_findAllServiceRequest_result"/>
    ...
```

```
    <attributeValues id="assignedDate" IterBinding="findAllServiceRequestIter">
      <AttrNames>
        <Item Value="assignedDate"/>
      </AttrNames>
    </attributeValues>
    <attributeValues id="problemDescription"
                     IterBinding="findAllServiceRequestIter">
      <AttrNames>
        <Item Value="problemDescription"/>
      </AttrNames>
    </attributeValues>
    <action id="First" RequiresUpdateModel="true" Action="12"
            IterBinding="findAllServiceRequestIter"/>
    <action id="Previous" RequiresUpdateModel="true" Action="11"
            IterBinding="findAllServiceRequestIter"/>
    <action id="Next" RequiresUpdateModel="true" Action="10"
            IterBinding="findAllServiceRequestIter"/>
    <action id="Last" RequiresUpdateModel="true" Action="13"
            IterBinding="findAllServiceRequestIter"/>
    <table id="ServiceRequestserviceHistoryCollection"
           IterBinding="serviceHistoryCollectionIterator">
      <AttrNames>
        <Item Value="lineNo"/>
        <Item Value="nextLineItem"/>
        ...
      </AttrNames>
    </table>
</bindings>
```

8.3.3 What Happens at Runtime

As was previously mentioned in Section 5.5.2.2, "Binding Objects Defined in the executables Element", ADF iterators are associated with underlying `RowSetIterator` objects, which manage which data objects, or *rows*, are currently displayed on a page. At runtime, the rowset iterators manage the data displayed in the master and detail components.

Both the master and detail rowset iterators listen to rowset navigation events, such as the user selecting a specific row or clicking the range navigation buttons, and display the appropriate rows in the UI. In the case of the default master-detail components, the rowset navigation events are the command buttons on a form (First, Previous, Next, Last) or the selection facet and **Submit** button on a table.

The rowset iterator for the detail collection manages the synchronization of the detail data with the master data. It listens to the row navigation events in both the master and detail collections. The `MasterBinding` attribute on the detail iterator definition in the page definition file tells the detail rowset iterator which master iterator to listen to. If a rowset navigation event occurs in the master collection, the detail rowset iterator automatically executes and returns the detail rows related to the current master row.

8.3.4 What You May Need to Know About Master-Detail 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 Control Palette, you can also display the collections on separate pages, and still have the binding iterators manage the synchronization of the master and detail objects.

For example, in the SRDemo application the service requests and service history are displayed on the SRMain page. However, the page could display the service request only, and instead of showing the service history, it could provide a button called **Details**. If the user clicks the **Details** button, the application would navigate to a new page that displays all the related service history in a table. A button on the service history page would enable the user to return to the service request page.

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 Control Palette. (For information about using the forms or tables, see Chapter 6, "Creating a Basic Page" or Chapter 7, "Adding Tables".) Remember that the detail object iterator manages the synchronization of the master and detail data. So, be sure to drag the appropriate detail object from the Data Control Palette when you create the page to display the detail data (see Section 8.2, "Identifying Master-Detail Objects on the Data Control Palette").

To handle the page navigation, 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 Control Palette. Each button must specify a navigation rule outcome value in the `action` attribute. In the `faces-config.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. For information about adding navigation between pages, see Chapter 9, "Adding Page Navigation".

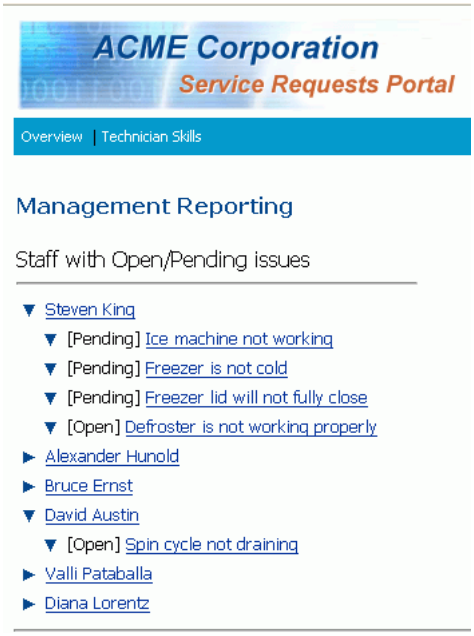
8.4 Using Trees to Display Master-Detail Objects

In addition to tables and forms, you can also display master-detail data in hierarchical trees. The ADF Faces `tree` component, available from the Data Control Palette, 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 8.5, "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 8–3 shows an example of a tree from the SRManage page of the SRDemo application. The tree displays two levels of nodes: staff members and service requests assigned to them. The root nodes display staff members. The branch nodes display open or pending service requests assigned to each staff member.

Figure 8–3 Databound ADF Faces Tree



8.4.1 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` is populated by a different data collection. In JDeveloper, you define a databound tree using the Tree Binding Editor, which enables you to define the rules for populating each node level in the tree. There must be one rule for each node level in the hierarchy. Each rule defines the following node level properties:

- The data collection that populates that node level
- The attributes from the data collection that are displayed at that node level
- An accessor method that returns a detail object to be displayed as a branch of the current node level

To display master-detail objects in a tree:

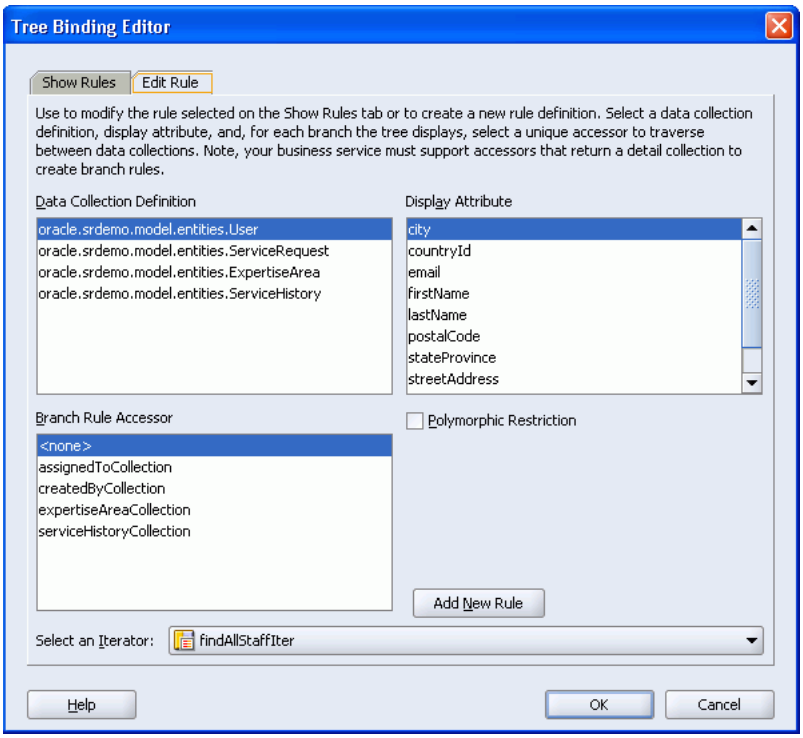
1. Drag the master object from the Data Control Palette, and drop it onto the page. This should be the master data that will represent the root level of the tree.

   Note: The root node must be a collection represented by a method return or accessor return. You cannot use a single-object accessor return as the root node of a tree.

2. In the context menu, choose **Trees** > **ADF Tree**.

   JDeveloper displays the Tree Binding Editor, as shown in Figure 8–4.

Figure 8–4 Tree Binding Editor, Edit Rule Tab



3. In the **Edit Rule** page of the Tree Binding Editor, define a rule for each node level that you want to appear in the tree. To define a rule you must select the following items:

   - **Data Collection Definition**: Select the data collection that will populate the node level you are defining.

     The first rule defines the root node level. So, for the first rule, select the same collection that you dragged from the Data Control Palette to create the tree, which was a master collection.

     To create a branch node, select the appropriate detail collection. For example, to create a root node of users, you would select the `User` collection for the first (root node) rule; to create a branch that displays services requests, you would select the `ServiceRequest` collection in the branch rule.

   - **Display Attribute**: Select one or more attributes to display at each node level. For example, for a node of users, you might select both the **FirstName** and **LastName** attributes.

   - **Branch Rule Accessor**: Select the accessor method that returns the detail collection that you want to appear as a branch under the node level you are defining. The list displays only the accessor methods that return the detail collections for the master collection you selected for the rule. If you choose **<none>**, the node will not expand to display any detail collections, thus ending the branch. For example, if you are defining the `User` node level and you want to add a branch to the service requests for each user, you would select the accessor method that returns the service request collection. Then, you must define a new rule for the `serviceRequest` node level.

- **Polymorphic Restriction**: Optionally, you can define a node-populating rule for an attribute whose value you want to make a discriminator. The rule will be polymorphic because you can define as many node-populating rules as desired for the same attribute, as long as each rule specifies a unique discriminator value. The tree will display a separate branch for each polymorphic rule, with the node equal to the discriminator value of the attribute.

  Tip: Be sure to click **Add New Rule** after you define each rule. If you click **OK** instead, the last rule you defined will not be saved. When you click **Add New Rule**, JDeveloper displays the **Show Rules** tab of the Tree Binding Editor, where you can verify the rules you have created.

4. Use the **Show Rules** page of the Tree Binding Editor, shown in Figure 8–5, to:

   - Change the order of the rules

     The order of the rules should reflect the hierarchy that you want the tree to display.

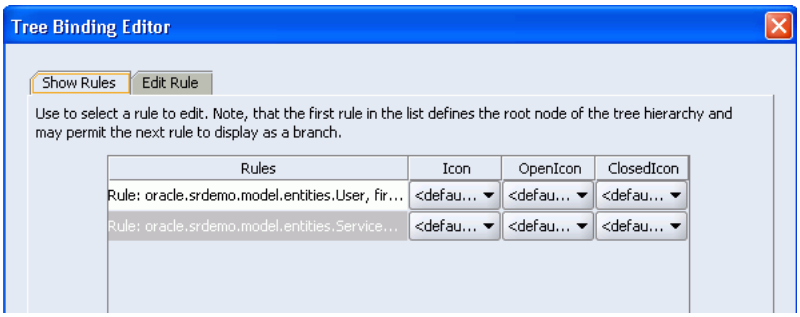
   - Delete rules

   Note: You cannot change the icon displayed in an ADF Faces or JSF tree component.

   The first rule listed in the **Show Rules** page of the Tree Binding Editor, populates the root node level of the tree. So, be sure that the first rule populates the logical root node for the tree, depending on the structure of your data model.

   For example, in the sample tree previously shown in Figure 8–3, the first rule would be the one that populates the user nodes. The order of the remaining rules should follow the hierarchy of the nodes you want to display in the tree.

Figure 8–5 Tree Binding Editor, Show Rule Tab



8.4.2 What Happens When You Create ADF Databound Trees

When you drag and drop from the Data Control Palette, JDeveloper does many things for you. For a full description of what happens and what is created when you use the Data Control Palette, see Section 5.2.3, "What Happens When You Use the Data Control Palette".

When you create a databound tree using the Data Control Palette, 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.

8.4.2.1 Code Generated in the JSF Page

Example 8–2 shows the code generated in a JSF page when you use the Data Control Palette to create a tree. This sample tree displays two levels of nodes: users and service requests. The `User` collection is the root node and is returned by the `findAllStaff` method.

Example 8–2 Code Generated in the JSF Page for a Databound Tree

```
<h:form>
   <af:tree value="#{bindings.findAllStaff1.treeModel}" var="node">
      <f:facet name="nodeStamp">
         <af:outputText value="#{node}"/>
      </f:facet>
   </af:tree>
</h:form>
```

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 `findAllStaff1` tree binding object in the page definition file. 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.

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.

8.4.2.2 Binding Objects Defined in the Page Definition File

Example 8–3 shows the binding objects defined in the page definition file for the ADF databound tree.

Example 8–3 Binding Objects Defined the Page Definition File for a Databound Tree

```
<executables>
    <methodIterator id="findAllStaffIter" Binds="findAllStaff.result"
                    DataControl="SRPublicFacade" RangeSize="10"
                    BeanClass="oracle.srdemo.model.entities.User"/>
</executables>
<bindings>
    <methodAction id="findAllStaff" InstanceName="SRPublicFacade.dataProvider"
                  DataControl="SRPublicFacade" MethodName="findAllStaff"
                  RequiresUpdateModel="true" Action="999"
                  ReturnName="SRPublicFacade.methodResults.
                                SRPublicFacade_dataProvider_findAllStaff_result"/>
    <tree id="findAllStaff1" IterBinding="findAllStaffIter">
      <AttrNames>
        <Item Value="city"/>
        <Item Value="countryId"/>
        <Item Value="email"/>
        <Item Value="firstName"/>
        <Item Value="lastName"/>
        <Item Value="postalCode"/>
        <Item Value="stateProvince"/>
        <Item Value="streetAddress"/>
        <Item Value="userId"/>
        <Item Value="userRole"/>
      </AttrNames>
      <nodeDefinition DefName="oracle.srdemo.model.entities.User" id="UserNode">
        <AttrNames>
          <Item Value="firstName"/>
          <Item Value="lastName"/>
        </AttrNames>
        <Accessors>
          <Item Value="assignedToCollection"/>
        </Accessors>
      </nodeDefinition>
      <nodeDefinition DefName="oracle.srdemo.model.entities.ServiceRequest"
                      id="ServiceRequestNode">
        <AttrNames>
          <Item Value="problemDescription"/>
        </AttrNames>
      </nodeDefinition>
    </tree>
</bindings>
```

The page definition file contains the rule information defined in the Tree Binding Editor. In the `executables` element, notice that although the tree displays two levels of nodes, only one iterator binding object is needed. This iterator iterates over the master collection, which populates the root nodes of the tree. The accessor you specified in the node rules return the detail data for each branch node.

In the example, the iterator happens to be a method iterator because a method return was dragged from the Data Control Palette and dropped on the page as an ADF tree. If an accessor return had been dragged from the Data Control Palette, this would be an accessor iterator instead of a method iterator. Because a method iterator is used in this example, a corresponding `methodAction` is defined in the `bindings` element. The `methodAction` encapsulates the details about how to invoke the method, which returns the data collection.

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 method that returns the next branch of the tree.

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

For more information about the elements and attributes of the page definition file, see Section A.7, "<pageName>PageDef.xml".

8.4.3 What Happens at Runtime

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

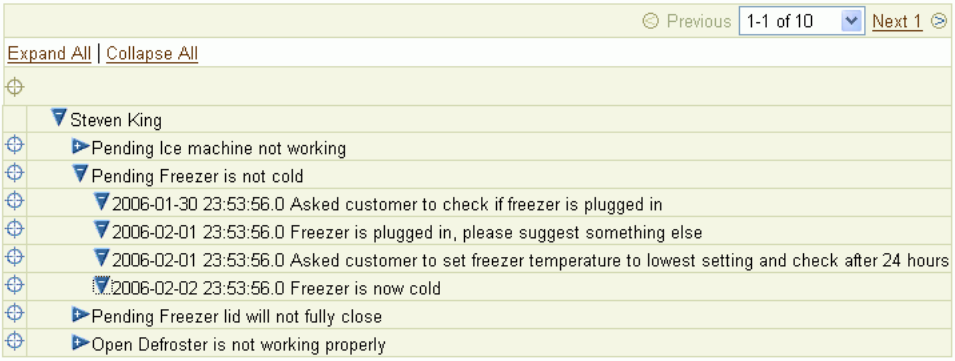
The `DisclosureListener` attribute on the tree is bound to the accessor method specified in the node rule defined in the page definition file. This accessor method is invoked in response to the `DisclosureEvent` event; in other words, whenever a user expands the node the accessor method populates the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

8.5 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.

Figure 8–6 shows an example of a tree table that displays three levels of nodes: users, service requests, and service history. Each root node represents an individual user. The branches off the root nodes display the service requests associated with that user. Each service request node branches to display the service history for each service request.

Figure 8–6 Databound ADF Faces Tree Table



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.

The ADF Faces `treeTable` component includes the following built-in functionality:

- 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 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.

8.5.1 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 data collection as an **ADF Tree Table** instead of an **ADF Tree**. For more information, see Section 8.4.1, "How to Display Master-Detail Objects in Trees".

8.5.2 What Happens When You Create a Databound Tree Table

When you drag and drop from the Data Control Palette, JDeveloper does many things for you. For a full description of what happens and what is created when you use the Data Control Palette, see Section 5.2.3, "What Happens When You Use the Data Control Palette".

When you create a databound tree table using the Data Control Palette, 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.

8.5.2.1 Code Generated in the JSF Page

Example 8–4 shows the code generated in a JSF page when you use the Data Control Palette to create a tree table. This sample tree table displays three levels of nodes: users, service requests, and service history.

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, which in the example is the `findAllStaff1` tree binding object. 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 8–4 Code Generated in the JSF Page for a Databound ADF Faces Tree Table

```
<h:form>
  <af:treeTable value="#{bindings.findAllStaff1.treeModel}" var="node">
    <f:facet name="nodeStamp">
      <af:column>
        <af:outputText value="#{node}"/>
      </af:column>
    </f:facet>
    <f:facet name="pathStamp">
      <af:outputText value="#{node}"/>
    </f:facet>
  </af:treeTable>
</h:form>
```

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.

8.5.2.2 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. For more information about tree binding objects, see Section 8.4.2.2, "Binding Objects Defined in the Page Definition File".

8.5.3 What Happens at Runtime

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

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

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

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

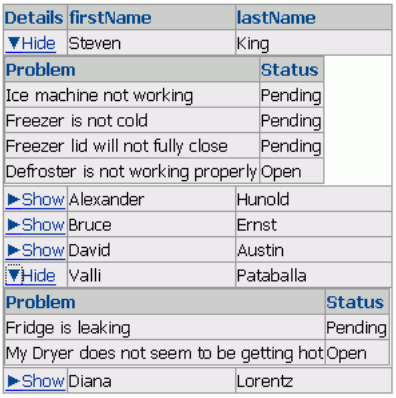
8.6 Using an Inline Table to Display Detail Data in a Master Table

As you may recall from Section 7.5, "Adding Hidden Capabilities to a Table", you can use the `detailStamp` facet in a table to hide or show additional information about a specific data object displayed in the table. When you add a component to this facet, the table displays an additional column labeled **Details**, which displays the additional information. It includes a toggle mechanism that enables the user to hide or show the information displayed in the **Details** column in a manner similar to the mechanism in an ADF Faces `tree` or `treeTable` component. In the case described in Section 7.5, "Adding Hidden Capabilities to a Table", the additional information was a single attribute from the same data collection that populates the table.

Using master-detail collections on the Data Control Palette, you can declaratively add an inline table to the `detailStamp` facet that displays additional information from a detail collection. A master collection is used to populate the main table and a detail collection is used to populate the inline table.

Figure 8–7 shows how an inline table of service requests can be embedded in a table of service request staff. If the user clicks the **Show** link in the **Details** column, which is built into the table facet, an inline table of service requests is displayed under the selected row of the table. The main table is populated by a master collection of users and displays the user's first and last name. The inline table is populated by a detail collection of service requests and displays the service request problem description and status.

Figure 8–7 Inline Table Displaying Information from a Detail Collection



8.6.1 How to Display Detail Data Using an Inline Table

Using the Data Control Palette, you can create both the main table and the inline table in a single declarative action. Since an inline table is similar to a tree table, you use the Tree Binding Editor to define the rules that populate the main table and the inline detail table. There must be one rule for the main table and one rule for the inline detail table. Each rule defines the following properties:

- The data collection that populates the table
- The attributes from the data collection that are displayed in the table

The rule for the main table must also specify an accessor method that returns the detail collection that will populate the inline table.

To create a master table with an inline detail table:

1. Drag a master data object from the Data Control Palette, and drop it on the page. This should be the master object that you want to populate the main table.

   Note: You cannot use a single-object accessor return to create a table.

2. In the context menu, choose **Tables** > **ADF Master Table, Inline Detail Table**.

   JDeveloper displays the Tree Binding Editor (previously shown in Figure 8–4).

3. In the **Edit Rule** page of the Tree Binding Editor, define a rule for populating the main table and another rule for populating the inline table. To define a rule you must select the following items:

   - **Data Collection Definition**: Select the data collection that will populate the table you are defining. The first rule defines the main table. So, for the first rule, select the same data collection that you dragged from the Data Control Palette (the master collection). When defining the rule for the inline table,

select the appropriate detail collection. For example, to create a main table of users, you would select the `User` collection for the first rule; to create an inline table that displays service requests related to a user, you would select the `ServiceRequest` collection in the branch rule.

- **Display Attribute**: Select one or more attributes to display in the table you are defining. Each attribute is a column in the table. For example, if the main table is displaying users, you might select both the **firstName** and **lastName** attributes.
- **Branch Rule Accessor**: If you are defining the rule for the main table, select the accessor method that returns the detail collection that you want to appear in the inline detail table. The list displays only the accessor methods that return the detail collections for the master collection you selected for the rule. If you are defining the rule for the inline table, select **<none>**, because you cannot embed a table inside the inline table.

Tip: Be sure to click the **Add New Rule** button after you define each rule. If you click the **OK** button instead, the last rule you defined will not be saved. When you click **Add New Rule**, JDeveloper displays the **Show Rules** tab of the Tree Binding Editor, where you can verify the rules you have created.

4. Use the **Show Rules** page of the Tree Binding Editor, shown in Figure 8–5, to:
   - Change the order of the rules

     The rule that populates the main table must be first in the list
   - Identify the icons you want displayed for the expand and collapse mechanism

     Only the main table uses the icons, so if you want to use an icon other than the default, specify it in the rule for the main table.

     The default open icon is a solid down arrow with a minus sign, while the default closed icon is a solid right arrow with a plus sign
   - Delete rules

8.6.2 What Happens When You Create an Inline Detail Table

When you drag and drop from the Data Control Palette, JDeveloper does many things for you. For a full description of what happens and what is created when you use the Data Control Palette, see Section 5.2.3, "What Happens When You Use the Data Control Palette".

8.6.2.1 Code Generated in the JSF Page

When you create a master table and an inline detail table using the Data Control Palette, JDeveloper adds binding objects to the page definition file, and it also adds the table and facet to the JSF page. The resulting UI components are fully functional and do not require any further modification.

Example 8–5 shows the code generated in the JSF page. This sample displays users in the main table and service requests in the inline detail table. The `User` collection is returned by the `findAllStaff` method. The main table is defined the same as any other ADF databound table. It is bound to the `findAllStaff1` binding object in the page definition file, which is a tree binding object. The columns in the main table display the user's first name and last name. The table includes a `detailStamp` facet in which the detail table is defined. The detail table is also bound to the

findAllStaff1 tree binding object, and the columns are set up to display the data from the service request collection. As with tree components, the page definition file defines the accessor method that returns the detail collection.

Example 8–5 JSF Code Created for the Master Table with an Inline Detail Table

```
<af:table rows="#{bindings.findAllStaff1.rangeSize}"
          emptyText="#{bindings.findAllStaff1.viewable ? \'No rows yet.\' :
                       \'Access Denied.\'}"
          var="row" value="#{bindings.findAllStaff1.treeModel}">
  <af:column headerText="#{bindings.findAllStaff1.labels.firstName}"
             sortable="false" sortProperty="firstName">
    <af:outputText value="#{row.firstName}"/>
  </af:column>
  <af:column headerText="#{bindings.findAllStaff1.labels.lastName}"
              sortable="false" sortProperty="lastName">
    <af:outputText value="#{row.lastName}"/>
  </af:column>
  <f:facet name="detailStamp">
    <af:table rows="#{bindings.findAllStaff1.rangeSize}"
              emptyText="No rows yet." var="detailRow"
              value="#{row.children}">
      <af:column headerText="#{row.children[0].labels.problemDescription}"
                 sortable="false" sortProperty="problemDescription">
        <af:outputText value="#{detailRow.problemDescription}"/>
      </af:column>
      <af:column headerText="#{row.children[0].labels.status}"
                 sortable="false" sortProperty="status">
        <af:outputText value="#{detailRow.status}"/>
      </af:column>
    </af:table>
  </f:facet>
</af:table>
```

8.6.2.2 Binding Objects Defined in the Page Definition File

Example 8–6 shows the binding objects added to the page definition file for a master table with an inline detail table. The executables element defines the findAllStaffIter iterator binding object, which iterates over the User collection that populates the main table. No iterator is needed for the detail collection, because the accessor method referenced in the tree binding object returns the detail data that is related to the currently selected master data.

In the bindings element, the methodAction binding object invokes the method that returns the User collection. The tree binding object populates the data in the master and detail tables. The nodeDefintion elements define the attributes that are displayed in the columns of the master and detail tables. The first nodeDefinition element defines the data in the master table, and the second one defines the data in the inline detail table. For more information about tree binding objects, see Section 8.4.2, "What Happens When You Create ADF Databound Trees".

Example 8–6 Binding Objects Added to the Page Definition File for a Master Table with an Inline Detail Table

```
<executables>
  <methodIterator id="findAllStaffIter" Binds="findAllStaff.result"
                  DataControl="SRPublicFacade" RangeSize="10"
                  BeanClass="oracle.srdemo.model.entities.User"/>
</executables>
<bindings>
  <methodAction id="findAllStaff" InstanceName="SRPublicFacade.dataProvider"
               DataControl="SRPublicFacade" MethodName="findAllStaff"
               RequiresUpdateModel="true" Action="999"
               ReturnName="SRPublicFacade.methodResults.
                           SRPublicFacade_dataProvider_findAllStaff_result"/>
  <tree id="findAllStaff1" IterBinding="findAllStaffIter">
    <AttrNames>
      <Item Value="city"/>
      <Item Value="countryId"/>
      <Item Value="email"/>
      <Item Value="firstName"/>
      <Item Value="lastName"/>
      <Item Value="postalCode"/>
      <Item Value="stateProvince"/>
      <Item Value="streetAddress"/>
      <Item Value="userId"/>
      <Item Value="userRole"/>
    </AttrNames>
    <nodeDefinition DefName="oracle.srdemo.model.entities.User" id="UserNode">
      <AttrNames>
        <Item Value="firstName"/>
        <Item Value="lastName"/>
      </AttrNames>
      <Accessors>
        <Item Value="assignedToCollection"/>
      </Accessors>
    </nodeDefinition>
    <nodeDefinition DefName="oracle.srdemo.model.entities.ServiceRequest"
                    id="ServiceRequestNode">
      <AttrNames>
        <Item Value="problemDescription"/>
        <Item Value="status"/>
      </AttrNames>
    </nodeDefinition>
  </tree>
</bindings>
```

8.6.3 What Happens at Runtime

When the user hides or shows the details of a row (by clicking the **Hide** or **Show** links), the table generates a `DisclosureEvent` event, which expands or collapses the inline detail table. The `isExpanded` method on this event determines whether the user is showing or hiding the detail table.

The `DisclosureEvent` event has an associated listener. The `DisclosureListener` attribute on the table is implicitly bound to the accessor method specified in the node rule defined in the page definition file. This accessor method is invoked in response to a `DisclosureEvent` event. For example, if the user clicks on the **Show** link, the accessor method is invoked to populate the data in the inline table.

9

Adding Page Navigation

This chapter describes how to create navigation rules and cases, and how to create basic navigation components, such as buttons and links, that trigger navigation rules using outcomes.

This chapter includes the following sections:

- Section 9.1, "Introduction to Page Navigation"
- Section 9.2, "Creating Navigation Rules"
- Section 9.3, "Using Static Navigation"
- Section 9.4, "Using Dynamic Navigation"

For information about how to create dynamic navigation menus, see Chapter 11, "Using Complex UI Components".

9.1 Introduction to Page Navigation

Navigation through a JSF application is defined by navigation rules. These rules determine, based on outcomes specified by UI components, which page is displayed next when the UI component is clicked.

Defining page navigation for an application is a two-step process:

- First, you create navigation rules for all the pages in your application.

  In most cases, you define one rule for each page in your application. However, you can also define pattern-based rules that affect groups of pages or global rules that affect all pages.

- Next, in each navigation component on the pages, such as a command button or link, you specify either a static or dynamic outcome value in the `action` attribute.

  Static outcome values are an explicit reference to a specific outcome defined in a navigation rule. *Dynamic* outcome values are derived from a binding on a backing bean method that returns an outcome value. In either case, the outcome value specified in the `action` attribute must match an outcome defined in the navigation rules or be handled by a default navigation rule for navigation to occur.

While you can create simple hand-coded navigation links between pages, using outcomes and navigation rules makes defining and changing application navigation much easier.

Read this chapter to understand:

- What navigation rules and cases are and how to create them
- How to create global, pattern-based, and default rules
- How to create UI components that use static outcome values
- How to bind navigation components to backing beans that return dynamic outcomes

9.2 Creating Navigation Rules

With JavaServer Faces, navigation between application pages is defined by a set of rules. Navigation rules determine the next page to display when a user clicks a navigation component, such as a button or a hyperlink.

A navigation rule defines the navigation from one page to one or more other pages. Each navigation rule can have one or more cases, which define where a user can go from that page. For example, if a page has links to several other pages in the application, you can create a single navigation rule for that page and one navigation case for each link to the different pages. The rule itself can define the navigation from:

- A specific JSF page
- All pages whose paths match a specified pattern, such as all the pages in one directory, which is called a *pattern-based* rule
- All pages in an application, which is called a *global* navigation rule

9.2.1 How to Create Page Navigation Rules

Navigation rule definitions are stored in the JSF configuration file (`faces-config.xml`). You can define the rules directly in the configuration file, or you can use the JSF Navigation Modeler and the JSF Configuration Editor in JDeveloper. Oracle recommends that you use the navigation modeler and the configuration editor, because these tools:

- Provide a GUI environment for modeling and editing the navigation between application pages
- Enable you to map out your application navigation using a visual diagram of pages and navigation links
- Update the `faces-config.xml` file for you automatically

Use the navigation modeler to initially create navigation rules from specific pages to one or more other pages in the application. Use the configuration editor to create global or pattern-based rules for multiple pages, create default navigation cases, and edit navigation rules.

9.2.1.1 About Navigation Rule Elements

Understanding the elements that define a navigation rule in the `faces-config.xml` file helps when creating rules using the navigation modeler and the configuration editor, or directly in the configuration file. The general syntax of a JSF navigation rule element in the `faces-config.xml` file is shown in Example 9–1.

Example 9–1 JSF Navigation Rule Syntax in the faces-config.xml File

```
<navigation-rule>
  <from-view-id>page-or-pattern</from-view-id>
  <navigation-case>
    <from-action>action-method</from-action>
    <from-outcome>outcome</from-outcome>
    <to-view-id>destination-page</to-view-id>
    <redirect/>
  </navigation-case>
  <navigation-case>
    ...
  </navigation-case>
</navigation-rule>
```

A navigation rule can consist of the following elements:

- `navigation-rule`: A mandatory wrapper element for navigation case elements.
- `from-view-id`: An optional element that contains either a complete page identifier (the context sensitive relative path to the page) or a page identifier prefix ending with the asterisk (*) wildcard character. If you use the wildcard character, the rule applies to all pages that match the wildcard pattern. To make a global rule that applies to all pages, leave this element blank.
- `navigation-case`: A mandatory wrapper element for each case in the navigation rule. Each case defines the different navigation paths from the same page. A navigation rule must have at least one navigation case.
- `from-action`: An optional element that limits the application of the rule only to outcomes from the specified action method. The action method is specified as an EL binding expression. For example, `#{backing_SRCreate.cancelButton_action}`.
- `from-outcome`: A mandatory element that contains an outcome value that is matched against values specified in the `action` attribute of UI components. Later you will see how the outcome value is referenced in a UI component either explicitly or dynamically through an action method return.
- `to-view-id`: A mandatory element that contains the complete page identifier of the page to which the navigation is routed when the rule is implemented.
- `redirect`: An optional element that indicates that the new view is to be requested through a redirect response instead of being rendering as the response to the current request. This element requires no value. (For more information, see Section 9.2.2, "What Happens When You Create a Navigation Rule".)

9.2.1.2 Using the Navigation Modeler to Define Navigation Rules

As a starting point for creating navigation rules, use JDeveloper's JSF Navigation Modeler. The navigation modeler is a visual modeling tool for creating application pages and navigation cases for those pages.

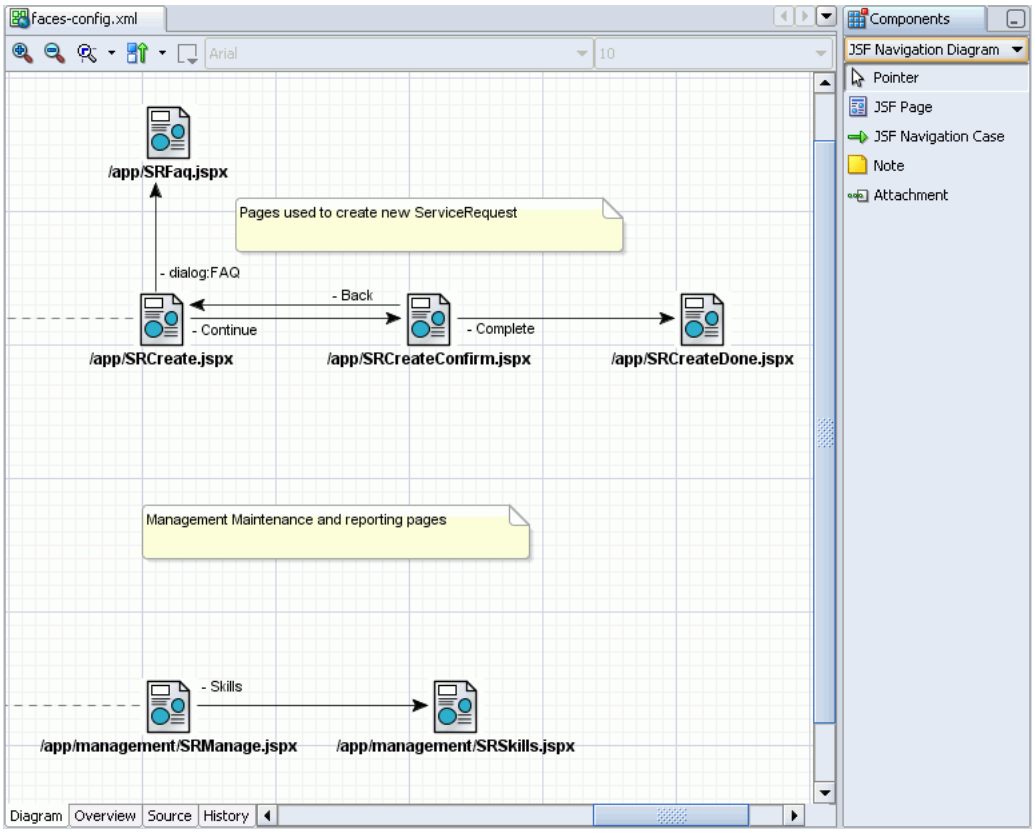
After creating the basic navigation rules using the navigation modeler, you can edit the rules in the JSF Configuration Editor or directly in the navigation modeler. There is one navigation modeler diagram for each JSF configuration file that you create.

To define a navigation rule using the JSF Navigation Modeler:

1. In the Application Navigator, double-click the `faces-config.xml` file located in the `WEB-INF` directory to display the configuration file in the visual editor.
2. In the visual editor, click the **Diagram** tab to display the navigation modeler, as shown in Figure 9–1.

   Notice that the Component Palette automatically displays the JSF Navigation Modeler components.

Figure 9–1 Navigation Modeler



3. Add application pages to the diagram using one of the following techniques:
   - To create a new page, drag **JSF Page** from the Component Palette onto the diagram. Double-click the page icon on the diagram to display the Create JSF JSP wizard where you can name and define the page characteristics.
   - To add an existing page to the diagram, drag the page from the Application Navigator onto the diagram.

     Tip: You can view a thumbnail of the entire diagram by clicking the **Thumbnail** tab in the Structure window.

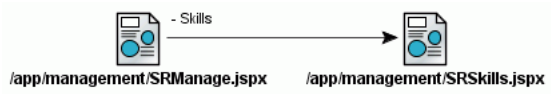
4. Create the navigation cases between the pages using the following technique:

   a. In the Component Palette, select **JSF Navigation Case** to activate it.

   b. On the diagram, click the icon for the source page, then click the icon for the destination page.

      JDeveloper draws the navigation case on the diagram as a solid line ending with an arrow between the two pages, as shown in Figure 9–2.

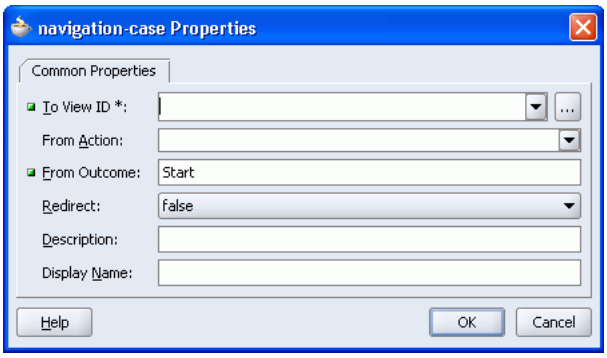
Figure 9–2 Navigation Case



The arrow indicates the direction of the navigation case. A default `from-outcome` value is shown as the label on the arrow. JDeveloper automatically creates the navigation rule for the source page and adds a default navigation case that references the destination page. If a page is the source for multiple navigation cases (for example, a page that provides links to several other pages), JDeveloper creates one rule for the source pages and adds the multiple cases to that rule.

5. In the diagram, double-click the arrow representing the navigation case to display the navigation-case Properties dialog, shown in Figure 9–3.

Figure 9–3 The navigation-case Properties Dialog



6. Use the navigation-case Properties dialog to define the elements in the navigation case. For a description of each element, see Section 9.2.1.1, "About Navigation Rule Elements".

9.2.1.3 Using the JSF Configuration Editor

Once you have defined your basic navigation between specific pages, you can use the JSF Configuration Editor to:

- Define pattern-based navigation rules for a group of pages.

  For example, if a group of pages in your application have a set of common links, such as the links from a menu bar, you can create a pattern-based rule that applies to all the pages. You identify the pages affected by the rule using a wildcard pattern, where the wildcard character (*) must be the last item in the pattern. A typical use of patterns in JSF navigation rules is to identify all the pages in a certain directory.

Example 9–2 shows a sample of a pattern-based navigation rule. Notice that the `from-view-id` element contains a pattern instead of a specific page name. This pattern would cause the rule to apply to all pages in the `management` directory whose names start with *SR*.

Example 9–2 Pattern-Based Navigation Rule

```
<navigation-rule>
  <from-view-id>/app/management/SR*</from-view-id>
  ...
</navigation-rule>
```

- Define global navigation rules that apply to all pages.

  For example, an application could define one rule that applies to all pages and returns users to the application's home page. When you create a global rule, you exclude the `from-view-id` element, which causes the rule to apply to all pages. You can optionally include a `from-outcome` element, if you want to apply the rule whenever a UI component on any page returns a specific outcome. Example 9–3 shows a sample global navigation rule. It causes the home page to be displayed when any component on any page returns the value `gohome`.

Example 9–3 Global Navigation Rule

```
<navigation-rule>
  <navigation-case>
    <to-view-id>home.jsp</to-view-id>
    <from-outcome>gohome</from-outcome>
  </navigation-case>
</navigation-rule>
```

- Define default navigation cases in which no outcome is specified.

  For example, if a navigation component is defined using a dynamic outcome (where the outcome could be one of multiple values), you may want to create a navigation case for one or two specific outcomes and a default case for all other possible outcomes. This way, if a navigation component returns an unexpected outcome, the page navigates to a specific page. Example 9–4 shows a sample default navigation rule. It displays the home page whenever any component on any page returns an outcome that is not handled by any other navigation case.

  Tip: Default navigation cases do not apply if a component specifies a `null` value in the `action` attribute. In this case, no navigation occurs; instead, the same page is redisplayed.

Example 9–4 Default Navigation Rule

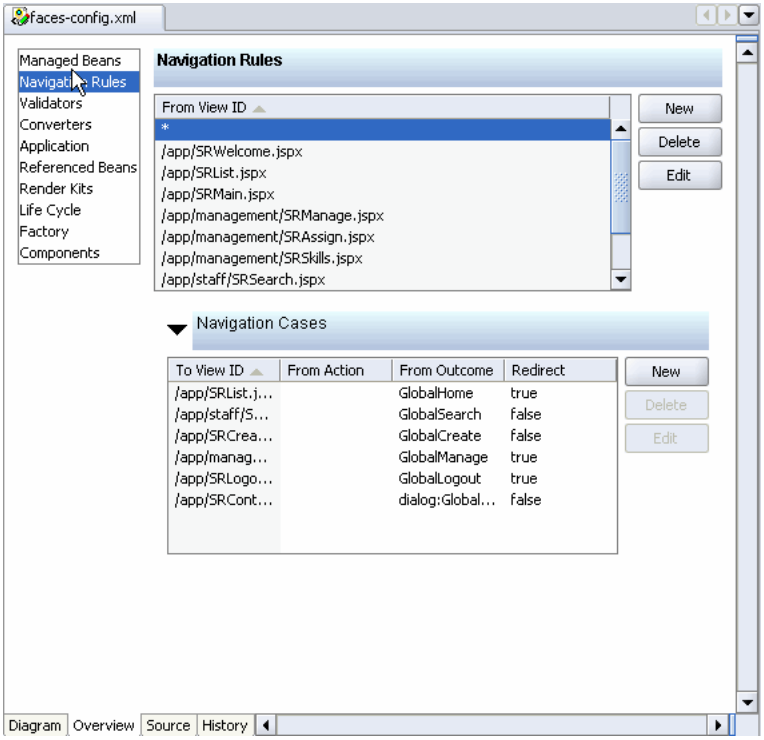
```
<navigation-rule>
  <navigation-case>
    <to-view-id>home.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
```

- Edit existing rules and cases.

To create a navigation rule using the JSF Configuration Editor:

1. In the Application Navigator, double-click the `faces-config.xml` file located in the `WEB-INF` directory to display the configuration file in the visual editor.
2. In the visual editor, click the **Overview** tab to display the configuration editor.
3. From the element list (in the left corner), select **Navigation Rules**, as shown in Figure 9–4.

Figure 9–4 Configuration Editor



4. Define the navigation rule using the following technique:
   a. Click the **New** button to the right of the Navigation Rules box to display the Create Navigation Rule dialog.
   b. Use the Create Navigation Rule dialog to specify the `from-view-id` element of the navigation rule using one of the following techniques:
      – To create a rule for a single page, enter a fully qualified page name or select a page from the dropdown list.
      – To create a pattern-based rule that applies to a group of pages whose names match the pattern, enter a pattern that uses the asterisk (*) wildcard character.

        You must use the wildcard character at the end of the pattern. For example, the pattern `/app/management/SR*` would cause the rule to apply to all pages in the management directory whose names start with *SR*. A typical use of patterns in JSF navigation rules is to identify all the pages in a certain directory.

– To create a global navigation rule that applies to all pages in the application, select **<Global Navigation Rule>** from the dropdown list.

  When you create a global navigation rule, the `from-view-id` element to be excluded from the `faces-config.xml` file.

Tip: When defining a global navigation rule, you can exclude the `from-view-id` element. However, for the sake of clarity in the `faces-config.xml` file, you may want to specify the value as either `<from-view-id>*</from-view-id>` or `<from-view-id>/*</from-view-id>`. All of these styles produce the same result—the rule is applied to all pages in the application.

When you finish, the new navigation rule appears in the navigation rules in the configuration editor.

5. Define the navigation cases using the following technique:

   a. In the list of navigation rules, select the rule to which you want to define navigation cases.

   b. Click the **New** button to the right of the **Navigation Cases** box to display the Create Navigation Case dialog.

   c. Use the Create Navigation Case dialog to specify the elements of the navigation case, which were previously described in Section 9.2.1.1, "About Navigation Rule Elements".

      You must supply a `to-view-id` value, to identify the destination of the navigation case, but can leave either or both the `from-action` and `from-outcome` elements empty. If you leave the `from-action` element empty, the case applies to the specified outcome regardless of how the outcome is returned. If you leave the `from-outcome` element empty, the case applies to all outcomes from the specified action method, thus creating a default navigation case for that method. If you leave both the `from-action` and the `from-outcome` elements empty, the case applies to all outcomes not identified in any other rules defined for the page, thus creating a default case for the entire page.

      Tip: If you have already defined the outcome values in the navigation components on the page, make sure you enter the `from-outcome` value exactly the same way, including lowercase and uppercase letters.

9.2.2 What Happens When You Create a Navigation Rule

When you create a navigation rule using the JSF Navigation Modeler or the JSF Configuration Editor, JDeveloper automatically adds the navigation rule elements to the `faces-config.xml` file for you.

When JDeveloper first creates an empty `faces-config.xml` file, it also creates a diagram file (`faces.config.oxd_faces`) to hold diagram details such as layout and annotations. JDeveloper always maintains this diagram file alongside the `faces-config.xml` file, which holds all the settings needed by your application. This means that if you are using versioning or source control, the diagram file is included as well as the `faces-config.xml` file it represents.

Example 9–5 shows a navigation rule with two cases defined in the `faces-config.xml` file for the SRCreate page in the SRDemo application. The first case navigates to the `SRCreateConfirm` page when the outcome specified in the `action` attribute of an activated navigation component is `Continue`. The second case navigates to the SRFaq page when the `action` attribute of an activated navigation component is `dialog:FAQ`. The `dialog:` outcome prefix causes the page in the `to-view-id` element to be launched as a dialog. For more information about creating dialogs, see Section 11.3, "Using Popup Dialogs".

Example 9–5 Navigation Rule for a Specific Page

```
<navigation-rule>
    <from-view-id>/app/SRCreate.jspx</from-view-id>
    <navigation-case>
      <from-outcome>Continue</from-outcome>
      <to-view-id>/app/SRCreateConfirm.jspx</to-view-id>
    </navigation-case>
    <navigation-case>
      <from-outcome>dialog:FAQ</from-outcome>
      <to-view-id>/app/SRFaq.jspx</to-view-id>
    </navigation-case>
</navigation-rule>
```

Example 9–6 shows a global navigation rule defined in the SRDemo application. The rule uses the wildcard character in the `from-view-id` element, which causes the rule to apply to all pages in the application. The cases defined in this global rule handle the navigation from the standard menu displayed on all of the pages.

Some of the cases use the `redirect` element, which causes JSF to send a redirect response that asks the browser to request the new page. When the browser requests the new page, the URL shown in the browser's address field is adjusted to show the actual URL for the new page. If a navigation case does not use the `redirect` element, the new page is rendered as a response to the current request, which means that the URL in the browser's address field does not change and that it will contain the address of the previous page. Direct rendering can be faster than redirection.

Any navigation case can be defined as a redirect. To decide whether to define a navigation case as a redirect, consider the following factors:

- If you do not use redirect rendering, when a user bookmarks a page, the bookmark will not contain the URL of the current page; instead, it will contain the the address of the previous page.
- If a user reloads a page, problems may arise if the URL is not refreshed to the new view. For example, if the page submits orders, reloading the page may submit the same order again. If any harm might result from not refreshing the URL to the new view, define the navigation case using the `redirect` element.

Example 9–6 Navigation Rule Defined with Redirect Rendering

```
<navigation-rule>
    <from-view-id>*</from-view-id>
    <navigation-case>
      <from-outcome>GlobalHome</from-outcome>
      <to-view-id>/app/SRList.jspx</to-view-id>
      <redirect/>
    </navigation-case>
    ...
    <navigation-case>
      <from-outcome>GlobalLogout</from-outcome>
      <to-view-id>/app/SRLogout.jspx</to-view-id>
      <redirect/>
    </navigation-case>
    <navigation-case>
      <from-outcome>dialog:GlobalContact</from-outcome>
      <to-view-id>/app/SRContact.jspx</to-view-id>
    </navigation-case>
</navigation-rule>
```

9.2.3 What Happens at Runtime

The Sun JSF Reference Implementation reads the navigation rules in the `faces-config.xml` file and calls the `NavigationHandler` class, which evaluates the navigation rules and determines which page to display. Knowing how the navigation rules are evaluated can help in debugging navigation issues.

When evaluating which navigation rules to execute, the navigation handler looks at three things:

- The ID of the current page
- The action method used to handle the link
- The outcome string value of the `action` attribute, or the string returned by the action method

The navigation handler evaluates navigation outcomes and rules in the following manner:

1. If the outcome returned by an action method is `null`, it returns immediately and redisplays the current page.
2. It merges all navigation rules with the same `from-view-id` value.
3. If a rule exists whose `from-view-id` value matches the view ID exactly, it uses that rule.
4. It evaluates all pattern-based navigation rules, and determines whether the prefix (the section before the wildcard character) is identical to the corresponding prefix of the ID of the current view.
5. If there are matching rules, it uses the rule whose matching prefix is longest. If there is a rule without a `from-view-id` element, it uses that rule.
6. If there is no match at all, it redisplays the current page.

Because the navigation handler merges navigation rules with matching `from-view-id` values, there may be several navigation cases from which to choose. After determining the correct navigation rule, the navigation handler evaluates which case to use based on a prioritized set of criteria.

If no case meets one criteria, the next criteria is applied until either a case is found or all criteria have been evaluated. The case evaluation criteria is as follows (shown in order of priority):

1. If both the `from-outcome` and `from-action` values of a case match the current action method and `action` value, it uses that case.
2. If a case has no from-action element, but the `from-outcome` value matches the current `action` value, it uses that case.
3. If a case has no `from-outcome` element, but the `from-action` value matches the current action method, it uses that case.
4. If there is a case with neither a `from-outcome` element nor a `from-action` element, it uses that case.
5. If no case meets any of the criteria, it redisplays the current page.

> **Tip:** When you are using Oracle ADF bindings in a page's UI components, the rowset iterators keep track of the current row. If a user clicks the browser's **Back** button instead of using the page's navigation buttons, the iterator becomes out of sync with the page displayed because the iterator has been bypassed. For more information about what happens when a user clicks the browser back button, see Section 6.4.4, "What You May Need to Know About the Browser Back Button".

9.2.4 What You May Need to Know About Navigation Rules and Cases

In addition to the basic navigation rules that have been discussed, you can define navigation rules in more than one JSF configuration file or define rules that overlap. You can also define overlapping navigation cases and cases that are split among different rules.

9.2.4.1 Defining Rules in Multiple Configuration Files

In a large application, you might want to define the navigation rules for pages in specific areas of the application in separate JSF configuration files. However, it is possible to specify rules in any of the JSF configuration files to apply to any pages in the application. In particular, each JSF configuration file may define rules for some general navigation features, such as returning to the home page or displaying help information. In such a scenario, when a navigation event arises at runtime, the rules from all the JSF configuration files are considered together. In JDeveloper, there is one navigation modeler diagram for each separate JSF configuration file.

If your application uses more than one JSF configuration file, JSF finds and loads your application's configuration settings in a predefined order. (For a description of how the configuration settings are evaluated, see Chapter 4, "Getting Started with ADF Faces".)

9.2.4.2 Overlapping Rules

Through the use of global or pattern-based rules, it is possible to define a hierarchy of overlapping rules.

Defining a hierarchy of rules ensures that particular navigation cases are directed to specific pages, and that general cases, such as clicking a Home button or a Help button, are handled in the same way across the whole application.

For example, you could create a hierarchy of rules by defining the `from-view-id` values as follows:

- `/products/select.jsp` to apply a rule to one page only
- `/product/*` to apply a rule to all pages in the product directory, including the page covered by the first rule
- `/*` to apply to all pages, including the ones covered by the previous two rules

Overlapping rules can be defined in a single rule or in multiple rules. When a user clicks a link, the more specific case is considered first, then the more general case.

9.2.4.3 Conflicting Navigation Rules

Because you can define several navigation rules for the same page, it is possible to define rules that conflict with one another. Also, because navigation rules can be defined in more than one JSF configuration file, similar rules may be defined in different files. Example 9–7 shows an example of conflicting rules in the same configuration file.

If there is a conflict in which two or more cases have the same `from-view-id`, `from-action`, and `from-outcome` values, the last case (as they are listed in the `faces-config.xml`) is used. If the conflict is among rules defined in different configuration files, the rule in the last configuration file to be loaded is used. Configuration files are loaded in the order they appear in the `web.xml` file.

Example 9–7 Conflicting Navigation Cases

```
<navigation-rule>
  <from-view-id>*</from-view-id>
  <navigation-case>
    <from-outcome>globalhelp</from-outcome>
    <to-view-id>/menu/generalHelp.html</to-view-id>
    <redirect/>
  </navigation-case>
</navigation-rule>

<navigation-rule>
  <from-view-id>*</from-view-id>
  <navigation-case>
    <from-outcome>globalhelp</from-outcome>
    <to-view-id>/menu/help.html</to-view-id>
    <redirect/>
  </navigation-case>
</navigation-rule>
```

9.2.4.4 Splitting Navigation Cases Over Multiple Rules

You can split the navigation cases for the links on one page among different navigation rules. For example, if your application provides users with a common set of controls for navigating to particular parts of the application, one rule could define the navigation cases for all the common controls, while other navigation rules would define the navigation from other controls.

To define navigation split over multiple rules, you must create separate navigation rules that would together define all the navigation cases, as shown in Example 9–8. When these rules are evaluated, the more specific navigation cases are used first, then the more general case.

Example 9–8 Navigation Cases Split Over Multiple Rules

```
<navigation-rule>
  <from-view-id>/order.jsp</from-view-id>
  <navigation-case>
    <from-action>#{backing_home.submit}</from-action>
    <from-outcome>success</from-outcome>
    <to-view-id>/summary.jsp</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-action>#{backing_home.check}</from-action>
    <from-outcome>success</from-outcome>
    <to-view-id>/check.jsp</to-view-id>
  </navigation-case></navigation-case>
</navigation-rule>
<navigation-rule>
  <from-view-id>/order.jsp</from-view-id>
  <to-view-id>/again.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
```

9.2.5 What You May Need to Know About the Navigation Modeler

When using the navigation modeler to create and maintain page navigation, be aware of the following features:

- Changes to navigation rules made directly in the `faces-config.xml` file using the XML editor or made in the configuration editor usually refresh the navigation modeler. Each JSF configuration file has its own navigation modeler diagram. If the information in a navigation diagram does not match the information in its `faces-config.xml` file, you can manually refresh the diagram by right-clicking on the diagram and choosing **Diagram > Refresh diagram from faces-config**.
- When you delete a navigation case on the diagram, the associated `navigation-case` element is removed from the `faces-config.xml` file. If you remove all the cases in a rule, the `navigation-rule` element remains in the `faces-config.xml` file. You can remove the rule directly in the `faces-config.xml` file.
- When you edit the label for the navigation case on the diagram, the associated `navigation-case` element is updated in the `faces-config.xml` file. You cannot change the destination of the navigation case in the diagram. You can, however, change the destination of a navigation case in the JSF Configuration Editor or directly in the `faces-config.xml` file itself.
- When you delete a page icon from the navigation diagram, the associated page file is not deleted the from the Web Content folder in the ViewController project in the Application Navigator.
- When you edit pages manually, JDeveloper does not automatically update the navigation diagram or the associated `faces-config.xml` file. Conversely, when you make changes to a page flow that affect the behavior of an existing page, JDeveloper does not automatically update the code in the page. To coordinate the navigation diagram with web page changes, right-click on the page in the navigation diagram and choose **Diagram > Refresh Diagram from All Pages**.

- The navigation modeler diagram is the default editor for the `faces-config.xml` file. If you have a large or complex application, loading the diagram may be slow, because the file may be large. If you do not want JSF diagram files to be created for your JSF configuration files, use the **Tools** > **Preferences** > **File Types** > **Default Editor** > **JSF Configuration File** option to change the default editor. If you change the default editor before opening the `faces-config.xml` file for the first time, no diagram file is created unless you specifically request one.

9.3 Using Static Navigation

When a component is defined using static navigation, the outcome value in the `action` attribute is a constant value that always triggers the same navigation case. When a user clicks a component that is using static navigation, a specific JSF page is displayed—there are no alternative navigation paths.

To use static navigation, you create the navigation case using a `from-outcome` value, but not a `from-action` value. In the `action` attribute of the navigation button or link you specify a constant outcome value that matches the value you entered in the `from-outcome` element of the navigation case.

For example, if you create a navigation case with a `from-outcome` value of `Confirm`, as shown in Example 9–9, you would create a button or link on the page that specifies `Confirm` as a static value of the `action` attribute, as shown in Example 9–10. In this case, when the user clicks the button, the navigation case causes the `ConfirmAction` page to be displayed.

Example 9–9 Navigation Case Defined in the faces-config.xml File

```
<navigation-case>
   <from-outcome>Confirm</from-outcome>
   <to-view-id>/app/ConfirmAction.jspx</to-view-id>
</navigation-case>
```

Example 9–10 Static Navigation Button Defined in a JSF Page

```
<af:commandButton text="Continue" action="Confirm"/>
```

9.3.1 How to Create Static Navigation

To create a navigation component that uses a static outcome, you can create the component using the Component Palette or the Data Control Palette. If you use the Data Control Palette, the `actionListener` attribute of the component will be bound to a data control operation or method. Once you have created the component, you can then specify the outcome value in the `action` attribute. When the user clicks the component, the application navigates to the page determined by the outcome value and navigation case. However, if the component is bound to a data control, first the operation or method is invoked, and then the navigation is performed.

For more information about command components that are bound to data control methods, see Section 10.3, "Creating Command Components to Execute Methods".

To create a navigation component that uses a static outcome:

1. Create a navigation component using one of the following techniques:

   - From the ADF Faces Core page of the Component Palette, drag a `CommandButton` or a `CommandLink` component onto the page.

     Tip: You can also use the JSF `commandButton` or `commandLink` components.

   - From the Data Control Palette, drag and drop an operation or a method onto the page and choose **ADF Command Button** or an **ADF Command Link** from the context menu.

     If you drag and drop a method that takes parameters, the ADF command button and command link components appear under **Method** in the context menu. JDeveloper displays the Action Binding Editor where you can define any parameter values you want to pass to the method. For more information about passing parameters to methods, see Section 10.4, "Setting Parameter Values Using a Command Component".

2. In the Structure window, select the navigation component and open the Property Inspector.

   Tip: The shortcut for opening the Property Inspector is **Ctrl+Shift-I**.

3. In the **Action** field displayed in the Property Inspector, enter the outcome value.

   The value must be a constant or an EL expression that evaluates to a string. To view a list of outcomes already defined in the page's navigation cases, click the dropdown in the **Action** field of the Property Inspector.

   Tip: If you want to trigger a specific navigation case, the outcome value you enter in the `action` attribute must exactly match the outcome value in the navigation case, including uppercase and lowercase. If the outcome specified by an action does not match any outcome in a navigation case, the navigation will be handled by a default navigation rule (if one exists), or no navigation will occur.

   Also, the `action` attribute must be either an outcome value or an EL expression that evaluates to an outcome value. You cannot enter a page URL in the `action` attribute.

9.3.2 What Happens When You Create Static Navigation

When you create a navigation component with static outcomes, JDeveloper adds the component to the JSF page. If you have not already done so, you will then need to add a navigation case to the `faces-config.xml` file to handle the navigation outcome specified in the component.

Example 9–11 shows a simple navigation component that was created using the ADF Faces `commandLink` component, which is available from the Component Palette. This command link appears on many of the SRDemo application's pages; it navigates to the SRAbout page, which displays information about the application.

Since there is only one possible navigation path, the command link is defined with a static outcome in the `action` attribute. The outcome value is `GlobalAbout`, which matches the `from-outcome` value of the navigation case shown in Example 9–12. The navigation case belongs to a global navigation rule that applies to all pages in the application.

Example 9–11 Navigation Component That Specifies a Static Outcome Value

```
<af:commandLink text="#{res['srdemo.about']}" action="GlobalAbout"
                              immediate="true"/>
```

Example 9–12 Navigation Rule Referenced by a Static Outcome Value

```
<navigation-rule>
    <from-view-id>*</from-view-id>
    ...
    <navigation-case>
      <from-outcome>GlobalAbout</from-outcome>
      <to-view-id>/app/SRAbout.jspx</to-view-id>
    </navigation-case>
    ...
</navigation-rule>
```

> **Tip:** If you enabled auto-binding by choosing the **Automatically Expose UI Components in a New Managed Bean** option when you created the page, any navigation component you create will automatically contain a binding to the managed bean (also known as a *backing bean*) defined for the page, even if the binding is not used. In a simple navigation component that has a static outcome, you may want to remove the unused binding from the component.

9.4 Using Dynamic Navigation

Instead of explicitly specifying a static outcome value in a navigation component, you can dynamically determine the outcome by binding the `action` attribute of a navigation component to an action method. An action method is a method in a backing bean (also known as a *managed bean*) that can perform an action (such as saving user input, for example) and return an outcome value. The outcome value determines the next page that should be displayed after the method performs an action. For example, an action method that verifies user input on a page might return one outcome if the input is valid and return another outcome if the input is invalid. Each of these different outcomes could trigger different navigation cases, causing the application to navigate to one of two possible target pages. As with static outcomes, a dynamic outcome triggers a navigation case that contains a matching `from-outcome` value or a default navigation case.

The method bound to a navigation component must be a public method with no parameters, and it must return a string representing the outcome of the action. An action method can return one of multiple outcomes depending on the processing it carries out. In other words, you can define conditional outcomes in the method logic. The outcome returned by the method must be defined in one of the cases in the page's navigation rules (unless you are using default rules, which handle all outcomes not specified in any navigation case).

Tip: In ADF applications, most processing of data objects is handled by the data control. Therefore, if a navigation component that uses dynamic outcomes needs to perform some processing on a data object (for example, creating, editing, deleting), it should be bound to a backing bean method that injects the ADF binding container. When a backing bean injects the ADF binding container, it calls the specified data control method to handle the processing of the data and then, based on the results, returns a navigation outcome to the UI component. For more information about injecting the binding container into a backing bean, see Section 10.5, "Overriding Declarative Methods".

9.4.1 How to Create Dynamic Navigation

If you want the outcome of a navigation component to be determined dynamically, you can bind the component to a method on a backing bean. The backing bean can execute some application logic and, depending on the results, return an outcome. The returned outcome will determine the navigation rule that is implemented. For information about creating backing beans, see Section 4.5, "Creating and Using a Backing Bean for a Web Page".

Note: If you enabled auto-binding by choosing the **Automatically Expose UI Components in a New Managed Bean** or the **Automatically Expose UI Components in an Existing Managed Bean** options when you created the page, any navigation component you create will automatically contain a binding to the managed bean (also known as a backing bean) defined for the page.

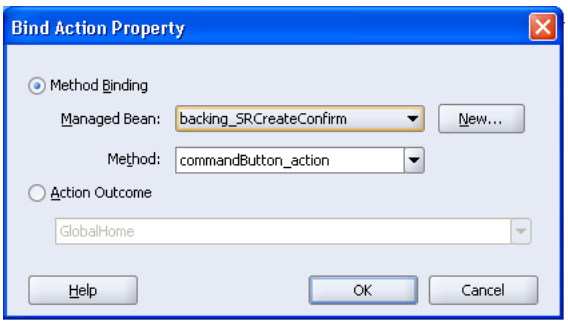
To create a navigation component that binds to a backing bean:

1. From the ADF Faces Core page of the Component Palette, drag a `CommandButton` or a `CommandLink` onto the page.

   Tip: You can also use the JSF `commandButton` and `commandLink` components.

2. In the visual editor double-click the UI component to display the Bind Action Property dialog, as shown in Figure 9–5.

Figure 9–5 Bind Action Property Dialog



The Bind Action Property dialog enables you to identify the backing bean and method to which you want to bind the component. If you enabled auto-binding when you created the page, the Bind Action Property dialog does not display the option for specifying a static outcome.

3. In the Bind Action Property dialog, identify the backing bean and the method to which you want to bind the component using one of the following techniques:
   - Click **New** to create a new backing bean. The Create Managed Bean dialog is displayed. Use this dialog to name the bean and the class.
   - Select an existing backing bean and method from the dropdown lists.
4. After identifying the backing bean and method, click **OK** on the Bind Action Property dialog.

   JDeveloper displays the source editor. If it is a new method, the source editor displays a stub method, as shown in Example 9–13. If it is an existing method, the source editor displays that method, instead of the stub method.

Example 9–13 Stub Method Created in the Backing Bean

```
public String commandButton1_action() {
    // Add event code here...
    return null;
}
```

5. Add any required processing logic to the method.
6. Change the return values of the method to the appropriate outcome strings.

   You may want to write conditional logic to return one of multiple outcomes depending on certain criteria. For example, you might want to return `null` if there is an error in the processing, or another outcome value if the processing was successful. A return value of `null` causes the navigation handler to forgo evaluating navigation cases and to immediately redisplay the current page.

   Tip: To trigger a specific navigation case, the outcome value you enter in the `action` attribute must exactly match the outcome value in the navigation rule, including uppercase and lowercase letters.

9.4.2 What Happens When You Create Dynamic Navigation

When you create a navigation component that specifies a dynamic outcome, JDeveloper adds an EL expression to the `action` attribute of the component tag. The EL expression references the backing bean method that will perform some application processing, such as saving user input, and return an outcome value.

Example 9–14 shows a button on the SRCreateConfirm page of the SRDemo application that uses a dynamic outcome value. The button was created using the ADF Faces `commandButton` component, which is available from the Data Control Palette context menu. The user clicks the button to create a new service request.

Example 9–14 Navigation Component That Uses Dynamic Outcomes

```
<af:commandButton text="#{res['srcreate.submit.button']}"
                        partialSubmit="false"
                        action="#{backing_SRCreateConfirm.createSRButton_action}"
                        id="createSRButton"/>
```

The button's `action` attribute is bound to the `createSRButton_action` method on the `SRCreateConfirm` backing bean, which is shown in Example 9–15.

Example 9–15 Backing Bean Method That Returns a Dynamic Outcome

```
    public String createSRButton_action() {
        BindingContainer bindings = getBindings();

        //Before we proceed check that the user has entered a description
        Object description =
            ADFUtils.getBoundAttributeValue(bindings, "SRCreatePageDef",
                                            "problemDescription");
        if (description == null) {
            FacesContext ctx = FacesContext.getCurrentInstance();
            ctx.addMessage(null,
                           JSFUtils.getMessageFromBundle("srcreate.
                                missingDescription",FacesMessage.SEVERITY_ERROR));

            return "Back";
        } else {
            //now find the facade method binding
            OperationBinding operationBinding =
                bindings.getOperationBinding("createServiceRequest");
            ServiceRequest result = (ServiceRequest)operationBinding.execute();

            //Put the number of the created service ID onto the request as an
            // example of passing data in that way
            Integer svrId = result.getSvrId();
            ExternalContext ectx =
                FacesContext.getCurrentInstance().getExternalContext();
            HttpServletRequest request = (HttpServletRequest)ectx.getRequest();
            request.setAttribute("SRDEMO_CREATED_SVRID", svrId);

            //Force a requery on the next visit to the SRList page
            UserSystemState.refreshNeeded();
            return "Complete";
        }
    }
```

The backing bean method starts by validating the user input. If the user did not enter a problem description for the service request, the method returns an outcome value of `Back`. If the user did enter a problem description, the method creates the service request and returns an outcome value of `Complete`. To create the service request, the backing bean method overrides the declarative method `createServiceRequest`, which was used to initially create the button. When a method overrides a declarative method, the JSF runtime injects the binding container for the current page using the managed property called `bindings`.The backing been method calls the `getBindings()` property getter, which accesses the current binding container, then it executes the method action binding for the `createServiceRequest` method in the `SRService` data control. For more information about overriding declarative methods, see Section 10.5, "Overriding Declarative Methods".)

Example 9–16 shows the navigation rule that handles the two possible outcomes returned by the backing bean.

Example 9–16 Navigation Rule Referenced by a Dynamic Outcome

```
<navigation-rule>
  <from-view-id>/app/SRCreateConfirm.jspx</from-view-id>
  <navigation-case>
    <from-outcome>Back</from-outcome>
    <to-view-id>/app/SRCreate.jspx</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>Complete</from-outcome>
    <to-view-id>/app/SRCreateDone.jspx</to-view-id>
  </navigation-case>
</navigation-rule>
```

9.4.3 What Happens at Runtime

When a user clicks a navigation component that has a dynamic outcome, the action method on the backing bean is executed. The method usually processes some user input and then returns an outcome value to the page. The JSF navigation handler evaluates the outcome returned by the action method and matches it to a navigation case that has the same value defined in the `from-outcome` element. The matching rule is then implemented and the page defined in the rule's `to-view-id` element is displayed. If the method does not return an outcome or if the outcome does not match any of the navigation cases, the user remains on the current page.

When using an action method to handle navigation in an application, you don't need to implement an action listener interface to invoke the method because JSF uses a default action listener to invoke action methods for page navigation: the method's logical outcome value is used to tell the JSF navigation handler what page to use for the render response.

9.4.4 What You May Need to Know About Using Default Cases

If an action method returns different outcomes depending on the processing logic, you may want to define a default navigation case to prevent having the method return an outcome that is not covered by any specific navigation case.

Default navigation cases catch all the outcomes not specifically covered in other navigation cases. To define a default navigation case, you can exclude the `from-outcome` element, which tells the navigation handler that the case should apply to any outcome not handled by another case.

For example, suppose you are using an action method to handle a **Submit** command button. You can handle the success case by displaying a particular page for that outcome. For all other outcomes, you can display a page explaining that the user cannot continue. Example 9–17 shows the navigation cases for this scenario.

Example 9–17 Navigation Rule with a Default Navigation Case

```
<navigation-rule>
  <from-view-id>/order.jsp</from-view-id>
  <navigation-case>
    <from-action>#{backing_home.submit}</from-action>
    <from-outcome>success</from-outcome>
    <to-view-id>/summary.jsp</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-action>#{backing_home.submit}</from-action>
    <to-view-id>/again.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
```

In the example, the first navigation case is a dynamic navigation case, where an action method is determining the outcome. If the outcome is `success`, the user navigates to the `/summary.jsp` page.

The second navigation case is a default navigation case that catches all other outcomes returned by the action method and displays the `/again.jsp` for all outcomes. Notice that the default case does not specify a `from-outcome` value, which causes this case to be implemented if the outcome returned by the action method does not match any of the other cases.

9.4.5 What You May Need to Know About Action Listener Methods

You can use action listener methods in a navigation component when you have an action that needs information about the user interface. Suppose you have a button that uses an image of the state of California, and you want a user to be able to select a county and display information about that county. You could implement an action listener method that determines which county is selected by storing an outcome for each county, and an action method that uses the outcome value to navigate to the correct county page.

To use an action method and action listener method on a component, you would reference them as shown in Example 9–18.

Example 9–18 Navigation Button with Action Listener and Action Methods

```
<h:commandButton image="californiastate.jpg"
                 actionListener="#{someBean.someListenmethod}"
                 action="#{someBean.someActionmethod}"/>
```

9.4.6 What You May Need to Know About Data Control Method Outcome Returns

Instead of binding an `action` attribute to a backing bean, you can bind it to a data control method that returns a navigation outcome. To bind the `action` attribute to a data control method you must enter the ADF binding expression manually and use the `outcome` binding property, as shown in Example 9–19.

Example 9–19 Navigation Component Bound to a Data Control Method

```
<af:commandButton
                text="Delete Service History Notes"
                action="#{bindings.deleteServiceHistoryNotes.outcome}"/>
```

The `outcome` property invokes the `outcome()` method in the `FacesCtrlActionBinding` class, which executes the data control method by calling the `execute` method of that same class. When the data control method returns a value, the `outcome()` method converts it to a string (if necessary) and returns it to the `action` attribute.

10

Creating More Complex Pages

This chapter describes how to add more complex bindings to your pages, such as using methods that take parameters to create forms and command components.

This chapter includes the following sections:

- Section 10.1, "Introduction to More Complex Pages"
- Section 10.2, "Using a Managed Bean to Store Information"
- Section 10.3, "Creating Command Components to Execute Methods"
- Section 10.4, "Setting Parameter Values Using a Command Component"
- Section 10.5, "Overriding Declarative Methods"
- Section 10.6, "Creating a Form or Table Using a Method that Takes Parameters"
- Section 10.7, "Creating an Input Form for a New Record"
- Section 10.8, "Creating Search Pages"
- Section 10.9, "Conditionally Displaying the Results Table on a Search Page"

10.1 Introduction to More Complex Pages

Once you create a basic page and add navigation capabilities, you may want to add more complex features to your pages, such as the ability to pass parameter values from one page to another, or the ability to search for and return certain records. ADF provides many features that allow you to add this complex functionality using very little actual code.

For example, you can create command components, such as buttons, that directly execute methods on your service bean. You only need to drag the method from the data control and drop it as a command button.You can also use command components to pass parameter values to another page.

In addition to creating basic forms that display information from a data store, you can also create forms that display only certain records (determined by parameter values), forms that allow you to create new records, and forms that search through and return objects based on a user's input.

Read this chapter to understand:

- How to create an use a managed bean to store parameter values or flags
- How to create command components bound to methods
- How to set parameter values
- How to add logic to a method bound to a command component
- How to create forms and tables that display only certain records
- How to create forms that allow users to create new records
- How to create search forms with result tables

10.2 Using a Managed Bean to Store Information

Often, pages require information from other pages in order to display correct information. Instead of setting this information directly on a page (for example, by setting the parameter value on the page's page definition file), which essentially hardcodes the information, you can store this information on a managed bean. As long as the bean is stored in a scope that is accessible, any value for an attribute on the bean can be accessed using an EL expression.

For example, the SREdit page requires the value of the `svrId` attribute for the row selected by the user on the previous page. This value provides the parameter value for the `findServiceRequestById(Integer)` method used to display the form. Additionally, the method bound to the **Cancel** button needs to return an outcome allowing the user to return to the correct page (the SREdit page can be accessed from three different pages). The SRDemo application has a managed bean that holds this information, allowing the sending page to set the information, and the SREdit page to use the information in order to determine where to navigate for the Cancel action. This information is stored as a hash map in the bean.

Managed beans are Java classes that you register with the application using the `faces-config.xml` file. When the JSF application starts up, it parses this configuration file and the beans are made available and can be referenced in an EL expression, allowing access to the beans' properties and methods. Whenever a managed bean is referenced for the first time and it does not already exist, the Managed Bean Creation Facility instantiates the bean by calling the default constructor method on the bean. If any properties are also declared, they are populated with the declared default values.

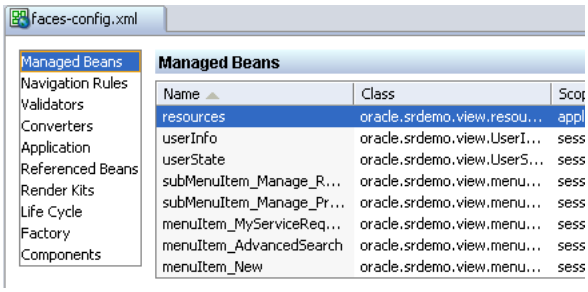
10.2.1 How to Use a Managed Bean to Store Information

Using the JSF Configuration Editor in JDeveloper, you can create a managed bean and register it with the JSF application at the same time.

To create a managed bean:

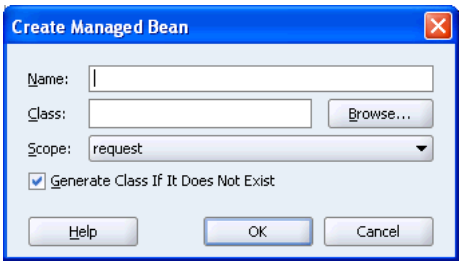
1. Open the `faces-config.xml` file. This file is stored in the `<project_name>/WEB-INF` directory.
2. At the bottom of the window, select the **Overview** tab.
3. In the element list on the left, select **Managed Beans**. Figure 10–1 shows the JSF Configuration Editor for the faces-config.xml file.

Figure 10–1 The JSF Configuration Editor



4. Click the **New** button to open the Create Managed Bean dialog, as shown in Figure 10–2. Enter the name and fully qualified class path for the bean. Select a scope, select the **Generate Java File** checkbox, and click **OK**.

Figure 10–2 The Create Managed Bean Dialog



> **Tip:** If the managed bean will be used by multiple pages in the application, you should set the scope to `Session`. However, then the bean cannot contain any reference to the binding container, as the data on the binding object is on the `request` scope, and therefore cannot "live" beyond a request. For examples of when you may need to reference the binding container, see Section 10.5, "Overriding Declarative Methods".

5. You can optionally use the arrow to the left of the **Managed Properties** bar to display the properties for the bean. Click **New** to create any properties. Press F1 for additional help with the configuration editor.

> **Note:** While you can declare managed properties using the configuration editor, the corresponding code is not generated on the Java class. You will need to add that code by creating private member fields of the appropriate type and then using the **Generate Accessors...** menu item on the context menu of the code editor to generate the corresponding getter and setter methods for these bean properties.

10.2.2 What Happens When You Create a Managed Bean

When you use the configuration editor to create a managed bean, and elect to generate the Java file, JDeveloper creates a stub class with the given name and a default constructor. Example 10–1 shows the code added to the `MyBean` class stored in the view package.

Example 10–1 Generated Code for a Managed Bean

```
package view;

public class MyBean {
    public MyBean() {
    }
}
```

JDeveloper also adds a `managed-bean` element to the `faces-config.xml` file. This declaration allows you to easily access any logic on the bean using an EL expression that refers to the given name. Example 10–2 shows the `managed-bean` element created for the `MyBean` class.

Example 10–2 Managed Bean Configuration on the faces-config.xml File

```
<managed-bean>
  <managed-bean-name>my_bean</managed-bean-name>
  <managed-bean-class>view.MyBean</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

You now need to add the logic required by your pages and methods in your application. You can then refer to that logic using an EL expression that refers to the `managed-bean-name` given to the managed bean. For example, to access the `myInfo` property on the bean, the EL expression would be:

```
#{my_bean.myInfo}
```

The following sections of this chapter provide examples of using the SRDemo application's `userState` managed bean (`view.UserSystemState.java`) to hold or get information. Please see those sections for more detailed examples of using a managed bean to hold information.

- Section 10.4, "Setting Parameter Values Using a Command Component"
- Section 10.5, "Overriding Declarative Methods"
- Section 10.6, "Creating a Form or Table Using a Method that Takes Parameters"
- Section 10.9, "Conditionally Displaying the Results Table on a Search Page"

10.3 Creating Command Components to Execute Methods

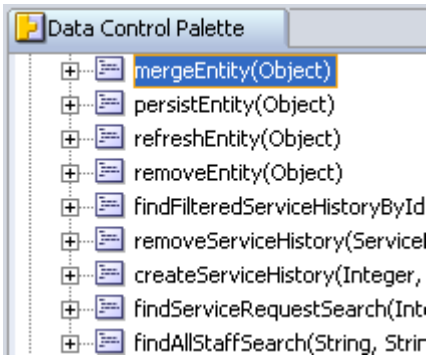
When you create a UI component by dragging and dropping a collection that is a return of a method, that method is executed when the page is rendered, so that it can return the collection. However, you can also drag the method itself (or any other type of method) and drop it as a command component to directly invoke the method.

> **Tip:** You can also drop the method as a parameter form. For more information, see Section 10.7.3, "How to Use a Custom Method to Create an Input Form" and Section 10.8, "Creating Search Pages".

In addition to custom methods, your data control may contain built-in methods that perform some standard business logic, such as updating or deleting objects. You can use these methods in conjunction with a collection. For example, the SRDemo application contains the `mergeEntity(Object)` method on the `SRPublicFacade` bean that can be used to update any object and merge it in the data source.

When you drag this method from the Data Control Palette and drop it as command button, the button is automatically bound to the method, and so that method will execute whenever the button is clicked. Figure 10–3 shows some of the methods in the Data Control Palette for the SRDemo application.

Figure 10–3 Methods in the Data Control Palette



Whether using a custom method or a built-in method, you can create a command button that executes the associated business logic on your service bean by binding the button to that method. When you use the Data Control Palette to create the button, JDeveloper creates the binding for you. You need only to set the values for any parameters needed by the method.

10.3.1 How to Create a Command Component Bound to a Service Method

In order to perform the required business logic, many methods require a value for the method's parameter or parameters. That means when you create a button bound to the method, you need to specify from where the value for the parameter(s) can be retrieved.

For example, if you use the `mergeEntity(Object)` method, you need to specify the object to be updated.

To add a button bound to a method:

1. From the Data Control Palette, drag the method onto the page.

   Tip: If you are dropping a button for a method that needs to work with data in a table or form, that button must be dropped inside the table or form.

2. From the context menu, choose **Methods** > **ADF Command Button**.
3. If the method takes parameters, the Action Binding dialog opens. In the Action Binding Editor, click the ellipses (**...**) button in the **Value** column of **Parameters** to launch the EL Expression Builder. You use the builder to set the value of the method's parameter.

   For example, to set the value for the `Object` parameter of the `mergeEntity(Object)` method used to update a collection, you would:

   1. In the EL Expression Builder, expand the **ADF Bindings** node and then expand the **bindings** node.

      All the bindings in the JSF page's page definition file are displayed.

   2. Expand the node for the iterator that works with the object you want to merge.

3. Expand the **currentRow** node. This node represents the current row in the iterator.
4. Select the **dataProvider** property and click the right-arrow button. Doing so creates an EL expression that evaluates to the data for the object in the current row of the iterator. Click **OK** to close the EL Expression Builder and populate the value with the EL expression. Click **OK** again to bind the button to the method.

   Tip: Consider criteria such as the following when determining what to select for the parameter value:

   - If you want the method to operate on a row in a table, you would set the parameter to be the current row in the table binding, and not the current object in the iterator.
   - If you want to be able to update multiple rows that represent detail objects in a master-detail relationship, you can set the parameter to be the master list object.
   - If the value is stored in a scope or on a managed bean, you would select the corresponding attribute in that scope or on that managed bean. However, this value must be available when this method is executed. For more information, see Section 10.4, "Setting Parameter Values Using a Command Component".

10.3.2 What Happens When You Create Command Components Using a Method

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

- Defines a method action binding for the method. If the method takes any parameters, JDeveloper creates `NamedData` elements that hold the parameter values.
- Inserts code in the JSF page for the ADF Faces command component. This code is the same as code for any other command button, as described in Section 6.4.2.3, "Using EL Expressions to Bind to Navigation Operations". However, instead of being bound to the `execute` method of the action binding for a built-in operation, the button is bound to the `execute` method of the method action binding for the method that was dropped.

10.3.2.1 Using Parameters in a Method

As when you drop a collection that is a return of a method, when you drop a method that takes parameters onto a JSF page, JDeveloper creates a method action binding (for details, see Section 6.2.2.1, "Creating and Using Iterator Bindings"). However, when the method requires parameters to run, JDeveloper also creates `NamedData` elements for each parameter. These elements represent the parameters of the method.

For example, the `mergeEntity(Object)` method action binding contains a `NamedData` element for the `Object` parameter. This element is bound to the value specified when you created the action binding. Example 10–3 shows the method action binding created when you dropped the `mergeEntity(Object)` method, and bound the `Object` parameter (named `entity`) to the data for the current row in the `findServiceRequestByIdIter` iterator.

Example 10–3 Method Action Binding for a Parameter Method

```
<methodAction id="mergeEntity" InstanceName="SRPublicFacade.dataProvider"
                  DataControl="SRPublicFacade" MethodName="mergeEntity"
                  RequiresUpdateModel="true" Action="999"
                  ReturnName="SRPublicFacade.methodResults.SRPublicFacade
                        _dataProvider_mergeEntity_result">
  <NamedData NDName="entity"
             NDValue="${bindings.findServiceRequestByIdIter.
                       currentRow.dataProvider}"
             NDType="java.lang.Object"/>
</methodAction>
```

10.3.2.2 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 command button's `actionListener` attribute, see Section 6.4.3, "What Happens at Runtime: About Action Events and Action Listeners".

> **Tip:** Instead of binding a button to the `execute` method on the action binding, you can bind the button to 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 10.5, "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 10–4 shows the EL expression used to bind the command button to the `mergeEntity(Object)` method.

Example 10–4 JSF Code to Bind a Command Button to a Method

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

> **Tip:** When you drop a UI component onto the page, JDeveloper automatically gives it an ID based on the number of that component previously dropped, for example, `commandButton1`, `commandButton2`. 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. Note that if you do change the ID, you must manually update any references to it in EL expressions in the page.

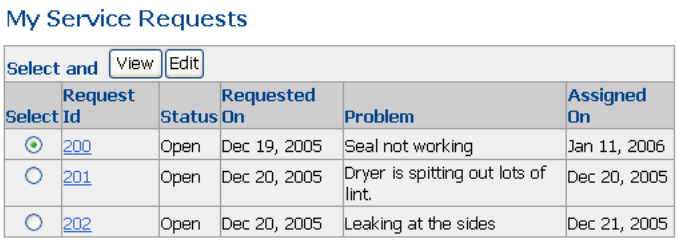
10.3.3 What Happens at Runtime

When the user clicks the button, the method binding causes the associated method to be invoked, passing in the value bound to the `NamedData` element as the parameter. For example, if a user clicks a button bound to the `mergeEntity(Object)` method, the method takes the value of the current record and updates the data source accordingly.

10.4 Setting Parameter Values Using a Command Component

There may be cases where an action on one page needs to set the parameters, for example, for a method used to display data on another page. As Figure 10–4 shows, the SRList page in the SRDemo application uses command links, which the user can click in order to directly edit a service request.

Figure 10–4 Command Links Used in a Table



The `commandLink` component is used to both navigate to the SREdit page and to set the needed parameter for the `findServiceRequestById(Integer)` method used to create the form that displays the data on the SREdit page. You can use the ADF Faces `setActionListener` component to set parameters.

> **Tip:** The SRDemo application does not use the `setActionListener` component for this page. Instead, this same functionality is provided by methods on a managed bean, as more than one page needs the same logic to set this parameter for the SREdit page. When logic needed for one page is also needed by other pages, it might be beneficial to place that logic on a managed bean. For more information about using managed beans, see Section 10.2, "Using a Managed Bean to Store Information".

10.4.1 How to Set Parameters Using Command Components

You can use the `setActionListener` component to set values on other objects. This component must be a child to a command component.

To use the setActionListener component:

1. Create a command component using either the Data Control Palette or the Component Palette.
2. From the Component Palette, drag a `setActionListener` component and drop it as a child to the command component.
3. In the Insert ActionListener dialog, set **From** to be the parameter value.
4. Set **To** to be where you want to set the parameter value.

Tip: Consider storing the parameter value on a managed bean or in scope instead of setting it directly on the resulting page's page definition file. By setting it directly on the next page, you lose the ability to easily change navigation in the future. For more information, see Section 10.2, "Using a Managed Bean to Store Information". Additionally, the data in a binding container is valid only during the request in which the container was prepared. Therefore, the data may change between the time you set it and the time the next page is rendered

5. If the parameter value is required by a method used to create a parameterized form on another page, when you create the form, set the parameter to the value of the **To** attribute in step 4. For more information, see Section 10.6, "Creating a Form or Table Using a Method that Takes Parameters".

10.4.2 What Happens When You Set Parameters

The `setActionListener` component lets the command component set a value before navigating. When you set the `From` attribute to the source of the value you need to pass, the component will be able to access that value. When you set the `To` attribute to a target, the command component is able to set the value on the target. Example 10–5 shows the code on the JSF page for a command link that accesses the data for the current row as the `from` value and sets that as the value of an attribute on a managed bean using the `To` attribute.

Example 10–5 JSF Page Code for a Command Link Using a setActionListener Component

```
<af:commandLink actionListener="#{bindings.setCurrentRowWithKey.execute}"
                action="edit"
                text="#{row.svrId}"
                disabled="#{!bindings.setCurrentRowWithKey.enabled}"
                id="commandLink1">
  <af:setActionListener from="#{row.svrId}"
                        to="#{userState.currentSvrId}"/>
</af:commandLink>
```

10.4.3 What Happens at Runtime

When a user clicks the command component, before navigation occurs, the `setActionListener` component sets the parameter value. In Example 10–5, the `setActionListener` gets the current row's `svrId` attribute value and sets it as the value for the `currentSvrId` attribute on the `userState` managed bean. Now, any method that needs this page's current row's `svrId` can access it using the EL expression `#{userState.currentSvrId}`.

For example, when dropping the `findServiceRequestById(Integer)` method to create the form for the SREdit page, you would enter `#{userState.currentSvrId}` as the value for the `Integer` parameter. For more information, see Section 10.6, "Creating a Form or Table Using a Method that Takes Parameters".

10.5 Overriding Declarative Methods

When you drop a method as a command button, JDeveloper binds the button to the `execute` method on the associated binding object. This binding allows you to create the JSF page declaratively, without needing to write the associated binding code. However, there may be occasions when you need to add logic before or after the method executes. For example, in order to delete multiple selected rows in a table, you must add code before the delete method executes that accesses each row and makes it current. For more information, see Section 7.6.4, "How to Use the TableSelectMany Component in the Selection Facet".

JDeveloper allows you to add logic to a declarative method by creating a new method and property on a managed bean that provide access to the associated action binding. By default, this generated code executes the method of the corresponding binding. You can then add logic before or after this code. JDeveloper automatically binds the command component to this new method instead of the `execute` property on the binding. Now when the user clicks the button, the new method is executed.

Following are some of the instances in the SRDemo application where backing beans contain methods that inject the binding container and then add logic before or after the declarative method is executed:

- `SRCreateConfirm.java`: The `createSR_action` method overrides the `createServiceRequest` method to add validation logic before the method is executed, and sets a parameter value after the method is executed.
- `SRMain.java`: The `deleteSR_action` method overrides the `removeServiceRequest` method to check whether a service request has associated histories before deleting. The `srDelete_action` method overrides the `removeServiceHistory` method in order to delete multiple rows in the Service History table.

10.5.1 How to Override a Declarative Method

In order to override a declarative method, you must have a managed bean to hold the new method to which the command component will be bound. If your page has a backing bean associated with it, JDeveloper adds the code needed to access the binding object to this backing bean. If your page does not have a backing bean, JDeveloper asks you to create one.

Note: You cannot use the following procedure if the command component currently has an EL expression as its value for the `Action` attribute, as JDeveloper will not overwrite an EL expression. You must remove this value before continuing.

To override a declarative method:

1. Drag the method to be overridden onto the JSF page and drop it as a UI command component.

   Doing so creates the component and binds it to the associated binding object in the ADF Model layer using the `ActionListener` attribute on the component.

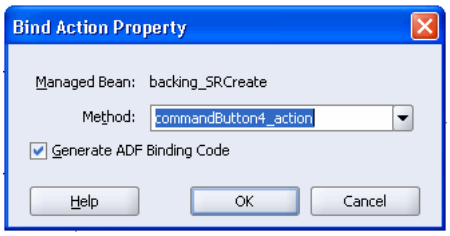
   For more information about creating command components using methods on the Data Control Palette, see Section 10.3, "Creating Command Components to Execute Methods".

2. On the JSF page, double-click on the component.

   In the Bind Action Property dialog, identify the backing bean and the method to which you want to bind the component using one of the following techniques:

   - If auto-binding has been enabled on the page, the backing bean is already selected for you, as shown in Figure 10–5.

Figure 10–5 Bind Action Property Dialog for a Page with Auto-Binding Enabled



     – To create a new method, enter a name for the method in the **Method** field, which initially displays a default name.

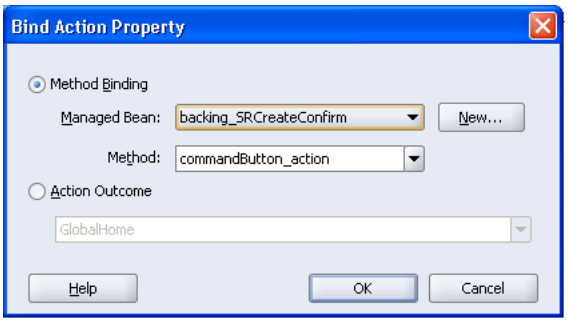
       OR

     – To use an existing method, select a method from the dropdown list in the **Method** field.

     – Select **Generate ADF Binding Code**.

   - If the page is not using auto-binding, you can select from an existing backing bean or create a new one, as shown in Figure 10–6. For more information about auto-binding, see Section 4.5.4, "How to Use the Automatic Component Binding Feature".

Figure 10–6 Bind Action Property Dialog for a Page with Auto-Binding Disabled



     – Click **New** to create a new backing bean. The Create Managed Bean dialog is displayed. Use this dialog to name the bean and the class, and set the bean's scope.

       OR

     – Select an existing backing bean and method from the dropdown lists.

Note: If you are creating a new managed bean, then you must set the scope of the bean to `request`. Setting the scope to `request` is required because the data in the binding container object that will be referenced by the generated code is on the `request` scope, and therefore cannot "live" beyond a request.

Additionally, JDeveloper understands that the button is bound to the `execute` property of a binding whenever there is a value for the `ActionListener` attribute on the command component. Therefore, if you have removed that binding, you will not be given the choice to generate the ADF binding code. You need to either inject the code manually, or to set a dummy value for the `ActionListener` before double-clicking on the command component.

3. After identifying the backing bean and method, click **OK** in the Bind Action Property dialog

   JDeveloper opens the managed bean in the source editor. Example 10–6 shows the code inserted into the bean. In this example, a command button is bound to the `mergeEntity` method.

Example 10–6 Generated Code in a Backing Bean to Access the Binding Object

```
public BindingContainer getBindings() {

    return this.bindings;
}

public void setBindings(BindingContainer bindings) {
    this.bindings = bindings;
}

public String commandButton_action1() {
    BindingContainer bindings = getBindings();
    OperationBinding operationBinding =
        bindings.getOperationBinding("mergeEntity");
    Object result = operationBinding.execute();
    if (!operationBinding.getErrors().isEmpty()) {
        return null;
    }
    return null;
}
```

4. You can now add logic either before or after the binding object is accessed.

   Tip: To get the result of an EL expression, you need to use the `ValueBinding` class, as shown in Example 10–7

Example 10–7 Accessing the Result of an EL Expression in a Managed Bean

```
FacesContext fc = FacesContext.getCurrentInstance();
  ValueBinding expr =
    fc.getApplication().
      createValueBinding("#{bindings.SomeAttrBinding.inputValue}");
DCIteratorBinding ib = (DCIteratorBinding)
  expr.getValue(fc);
```

In addition to any processing logic, you may also want to write conditional logic to return one of multiple outcomes depending on certain criteria. For example, you might want to return `null` if there is an error in the processing, or another outcome value if the processing was successful. A return value of `null` causes the navigation handler to forgo evaluating navigation cases and to immediately redisplay the current page.

> **Tip:** To trigger a specific navigation case, the outcome value returned by the method must exactly match the outcome value in the navigation rule in a case-sensitive way.

The command button is now bound to this new method using the `Action` attribute instead of the `ActionListener` attribute. If a value had previously existed for the `Action` attribute (such as an outcome string), that value is added as the return for the new method. If there was no value, the return is kept as `null`.

10.5.2 What Happens When You Override a Declarative Method

When you ovrerride a declarative method, JDeveloper adds a managed property to your backing bean with the managed property value of `#{bindings}` (the reference to the binding container), and it adds a strongly-typed bean property to your class of the `BindingContainer` type which the JSF runtime will then set with the value of the managed property expression `#{bindings}`. JDeveloper also adds logic to the UI command action method. This logic includes the strongly-typed `getBindings()` method used to access the current binding container.

Example 10–8 shows the code that JDeveloper adds to the chosen managed bean. Notice that the return `String "back"` was automatically added to the method.

Example 10–8 Generated Code in a Backing Bean to Access the Binding Object

```
private BindingContainer bindings;
...
public String commandButton1_action() {
    BindingContainer bindings = getBindings();
    OperationBinding operationBinding =
            bindings.getOperationBinding("mergeEntity");
    Object result = operationBinding.execute();
    if (!operationBinding.getErrors().isEmpty()) {
        return null;
    }

    return "back";
}
```

JDeveloper automatically rebinds the UI command component to the new method using the `Action` attribute, instead of the `ActionListener` attribute. For example, Example 10–9 shows the code on a JSF page for a command button created by dropping the `mergeEntity` method. Notice that the `actionListener` attribute is bound to the `mergeEntity` method, and the `action` attribute has a `String` outcome of `back`. If the user were to click the button, the method would simply execute, and navigate to the page defined as the `toViewId` for this navigation case.

Example 10–9 JSF Page Code for a Command Button Bound to a Declarative Method

```
<af:commandButton actionListener="#{bindings.mergeEntity.execute}"
                  text="persistEntity"
                  disabled="#{!bindings.persistEntity.enabled}"
                  id="commandButton1"
                  action="back"/>
```

Example 10–10 shows the code after overriding the method on the page's backing bean. Note that the `action` attribute is now bound to the backing bean's method.

Example 10–10 JSF Page Code for a Command Button Bound to an Overridden Method

```
<af:commandButton text="persistEntity"
                  disabled="#{!bindings.mergeEntity.enabled}"
                  binding="#{backing_create.commandButton1}"
                  id="commandButton1"
                  action="#{backing_create.commandButton1_action}"/>
```

This code does the following:

- Accesses the binding container
- Finds the binding for the associated method, and executes it
- Adds a return for the method that can be used for navigation. By default the return is `null`, or if an outcome string had previously existed for the button's `Action` attribute, that attribute is used as the return value. You can change this code as needed. For more information about using return outcomes, see Section 9.4, "Using Dynamic Navigation".

> **Tip:** If when you click the button that uses the overridden method, you receive this error:
>
> ```
> SEVERE: Managed bean main_bean could not be created
> The scope of the referenced object: '#{bindings}' is
> shorter than the referring object
> ```
>
> it is because the managed bean that contains the overriding method has a scope that is greater than `request`, (that is, either `session` or `application`). Because the data in the binding container referenced in the method has a scope of `request`, the scope of this managed bean must be set to `request` scope or a lesser scope.

10.6 Creating a Form or Table Using a Method that Takes Parameters

There may be cases where a form or table needs information before it can display content. For these types of pages, you create the form or table using a returned collection from a method that takes parameters. The parameter values will have been set by another page or method.

For example, the form on the SREdit page is created using the returned collection from the `findServiceRequestsById(Integer)` method. Instead of returning all service requests, it returns only the service request the user selected on the previous page. The command link on the previous page sets the parameter (`Integer`), which provides the service request's ID. For more information about using a command component to set a parameter value, see Section 10.4, "Setting Parameter Values Using a Command Component".

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

To create forms or tables that require parameters, you must be able to access the values for the parameters in order to determine the record(s) to return. For example, before creating the your form or table, you may need to add logic to a command button on another page that will set the parameter value on some object that the method can then access. For more information, see Section 10.4, "Setting Parameter Values Using a Command Component". Once that is done, you can set the parameter value for the form or table.

To create a form or table that uses parameters:

1. From the Data Control Palette, drag a collection that is a return of a method that takes a parameter or parameters and drop it as any type of form or table.
2. In the Edit Form Fields dialog or Edit Table Columns dialog, configure the form or table as needed and click **OK**.

   For help in using the dialogs, click **Help**.

   Because the method takes parameters, the Action Binding Editor 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 ellipses (**...**) button in the **Value** field to open the EL Expression Builder. Select the node that represents the value for the parameter.

   For example, Example 10–5, "JSF Page Code for a Command Link Using a setActionListener Component" shows a `setActionListenerComponent` setting the `svrId` parameter value on the `userState` bean as the `currentSvrId` attribute. To access that value, you would use `#{userState.currentSvrId}` as 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.

10.6.2 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 10–11 shows the action method binding created when you dropped the `findServiceRequestsById(Integer)` method, where the value for the `findSvrId` was set to the `currentSvrId` attribute of the `UserState` managed bean.

Example 10–11 Method Action Binding for a Method Return

```
<bindings>
  <methodAction id="findServiceHistoryById"
                InstanceName="SRPublicFacade.dataProvider"
                DataControl="SRPublicFacade"
                MethodName="findServiceHistoryById" RequiresUpdateModel="true"
                Action="999"
                ReturnName="SRPublicFacade.methodResults.SRPublicFacade_
                            dataProvider_findServiceHistoryById_result">
    <NamedData NDName="svrIdParam" NDValue="${userState.currentSvrId}"
               NDType="java.lang.Integer"/>
  </methodAction>
...
</bindings>
```

Note that the `NamedData` element will evaluate to the current service request ID on the `userState` bean, as set by any requesting page.

10.6.3 What Happens at Runtime

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 SREdit page loads, it takes the value of the `currentSvrId` field on the `userState` managed bean, and sets it as the value of the parameter needed by the `findServiceRequestsById(Integer)` 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 service request that is displayed.

10.7 Creating an Input Form for a New Record

You can create a form that allows a user to enter information for a new record and then commit that record into the data source. When the session bean for your data control is created, by default, constructor methods are created for objects on the data control. For example, the SRPublicFacade session bean in the SRDemo application has constructor methods for products, expertise areas, service histories, users, and service requests. The constructors provide an easy way to declaratively create an object that can be passed to a method. For example, by default, session beans have a `persistEntity(Object)` CRUD method that can be used to persist new records to the database (for more information about the default session bean methods, see Section 3.2.1.2, "Generating Session Facade Methods"). When you use the constructor method to create an input form, that method is called to create the object and initialize the object's variables to the entered values. You can then easily pass the constructor's results to the `persistEntity` method, which will create the new record in the data source.

There may be instances, however, when you need more control over how the new object is created. For example, you may want certain attributes to be populated programmatically. In this case, you might create a custom method to handle the creation of objects.

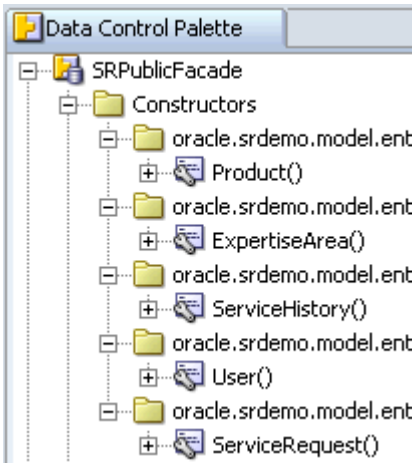
To use a custom method to create an input form, instead of dropping the collection returned from a method, you drop the method itself as a parameter form. JDeveloper automatically adds a command button bound to the method, so that the custom create method will execute when the user clicks the button.

For example, the SRDemo application's `SRPublicFacade` data control contains the `createServiceRequest(String, Integer, Integer)` method. `String` represents the value for the problem description, the first `Integer` represents the product ID, and the second `Integer` represents the ID of the user who created the request. This method creates a new instance of a `ServiceRequest` object using the values of the parameters to populate the attributes on the object; however, the product ID and the ID of the user are set by the method instead of by the user.

10.7.1 How to Use Constructors to Create an Input Form

Constructors for a data control are located in its **Constructor** folder. These access the default methods on the data control used to create an object (for more information, see Section 3.2.1.2, "Generating Session Facade Methods"). Figure 10–7 shows the constructors for the `SRPublicFacade` data control.

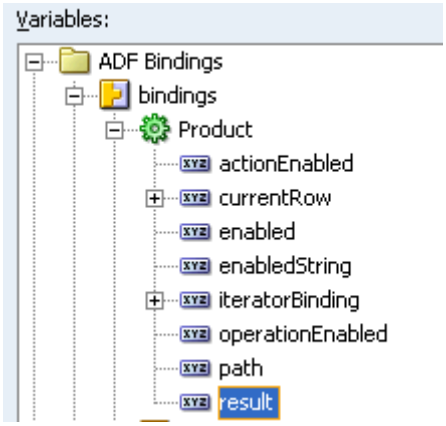
Figure 10–7 Constructors in the Data Control Palette



To create an input form using a constructor:

1. From the Data Control Palette, drag the appropriate constructor onto the JSF page. Constructors can only be dropped as a form, so no context menu displays.
2. In the Edit Form Fields dialog, set the labels, bindings, and UI components. Do not select **Include Submit Button**.
3. From the Data Control Palette, drag the `persistEntity(Object)` method. This method persists the object created by the constructor to the database.
4. In the context menu, choose **Methods** > **ADF Command Button**.
5. In the Action Binding Editor, enter the value for the `entity` parameter by clicking the ellipses (**...**) button in the **Value** field to launch the EL Expression Builder. Since you want the entity parameter to be the result of the constructor, select the **result** under the action binding for the constructor, as shown in Figure 10–8.

Figure 10–8 Result from the Product Constructor



Tip: If you will be navigating to this page from another page that should display the newly created object when you return to it, you must set the `cacheResults` attribute on the iterator for the first page to `false`.

For example, say you have a page that lists all products, and a user can navigate from that page to another page to create a product. A button on this page both creates the product and navigates back to the list page. In order for the user to see the product just created, you must set the iterator binding for the product list to `cacheResults=false`. Doing so forces the iterator to reexecute when returning to the page and display the newly created product.

10.7.2 What Happens When You Use a Constructor

When you use a constructor to create an input form, JDeveloper:

- Creates an action binding for the constructor method, a method iterator binding for the result of constructor method, and attribute bindings for each of the attributes of the object. It also creates an invoke action in the executables that causes the constructor method to execute during the Render Model phase.
- Inserts code in the JSF page for the form using ADF Faces `inputText` components.

For example, to create a simple input form for products in the SRDemo application, you might drop the product constructor method from the Data Control Palette as a parameter form, and then drop the `persistEntity` method as a button below the form, as shown in Figure 10–9.

Note: This page is an example only and does not exist in the SRDemo application.

Figure 10–9 A Simple Create Product Form

Example 10–12 shows the page definition file for this input form. When the invoke action binding executes, the constructor method is called using the action binding. The result of that constructor method is then bound to the iterator. When you drop the `persistEntity(Object)` method as a command button, that method can access the result of the constructer through the iterator.

Example 10–12 Page Definition Code for a Constructor

```
<executables>
   <invokeAction Binds="Product" id="invokeProduct" Refresh="renderModel"
                 RefreshCondition="${!adfFacesContext.postback and empty
                                              bindings.exceptionsList}"/>
   <methodIterator DataControl="SRPublicFacade"
                   BeanClass="oracle.srdemo.model.entities.Product"
                   Binds="Product.result" id="ProductIter"
                   Refresh="renderModel"
                   RefreshCondition="${!adfFacesContext.postback and empty
                                              bindings.exceptionsList}"/>
</executables>
<bindings>
  <methodAction DataControl="SRPublicFacade" id="Product" MethodName="Product"
                RequiresUpdateModel="true" Action="999"
                ReturnName="SRPublicFacade.methodResults.SRPublicFacade_
                                                    Product_result"
                ClassName="oracle.srdemo.model.entities.Product"/>
  <attributeValues id="description" IterBinding="ProductIter">
    <AttrNames>
      <Item Value="description"/>
    </AttrNames>
  </attributeValues>
  <attributeValues id="image" IterBinding="ProductIter">
    <AttrNames>
      <Item Value="image"/>
    </AttrNames>
  </attributeValues>
  <attributeValues id="name" IterBinding="ProductIter">
    <AttrNames>
      <Item Value="name"/>
    </AttrNames>
  </attributeValues>
  <attributeValues id="prodId" IterBinding="ProductIter">
    <AttrNames>
      <Item Value="prodId"/>
    </AttrNames>
  </attributeValues>
```

```
  <methodAction id="persistEntity" InstanceName="SRPublicFacade.dataProvider"
                DataControl="SRPublicFacade" MethodName="persistEntity"
                RequiresUpdateModel="true" Action="999"
                ReturnName="SRPublicFacade.methodResults.SRPublicFacade_
                                      dataProvider_persistEntity_result">
    <NamedData NDName="entity" NDValue="${bindings.Product.result}"
               NDType="java.lang.Object"/>
  </methodAction>
</bindings>
```

Anytime the constructor method is invoked, an object is created. However, since data is cached, (which allows the method to do a postback to the server), the constructor method will create the same object again when the user revisits the page, perhaps to create another object. Additionally, if errors occur, when the page is rerendered with the error message, the object would again be created.

To prevent duplicates, the invoke action's `refreshCondition` property is set so that the constructor will only be invoked whenever there has been no postback to the server and as long as there are no error messages. See Example 10–12 for the EL expression.

The iterator has the same refresh condition. This setting prevents the iterator from displaying the cached data used in the postback, and instead allows the form to display without any data when the user revisits the page.

10.7.3 How to Use a Custom Method to Create an Input Form

When you use a custom method to create an input form, you drag the method that can take the data populated by the user and drop it as a parameter form. In this case, since you need to create an object, you cannot drop a return. You drop the method itself.

To create an input form using a custom method:

1. From the Data Control Palette, drag the appropriate method onto the JSF page.
2. From the context menu select **Parameters** > **ADF Parameter Form**.

   The Edit Form Fields dialog opens, which allows you to customize the labels, bindings, and UI components before creating the form. JDeveloper automatically adds a command button bound to the method.

10.7.4 What Happens When You Use Methods to Create a Parameter Form

When you drop a method as a parameter form, JDeveloper:

- Defines variables to hold the data values, a method binding for the method, and the attribute bindings for the associated attributes in the page definition file.
- Inserts code in the JSF page for the form using ADF Faces `inputText` components and an ADF Faces command button component. This code is the same as code for any other input form or command button.

10.7.4.1 Using Variables and Parameters

Just as when you drop a collection that is a return of a method that takes parameters, when you drop the method itself onto a JSF page, JDeveloper creates `NamedData` elements for each parameter. However, since the user will provide the parameter values (instead of another page providing the values, as described in Section 10.6, "Creating a Form or Table Using a Method that Takes Parameters"), each `NamedData` element is bound to the attribute binding for the corresponding attribute.

This binding allows the method to access the correct attribute's value for the parameter on execution.

For example, the `createServiceRequest` method action binding contains a `NamedData` element for each of the parameters it takes. The `NamedData` elements are then bound to a corresponding attribute binding using an EL expression. Example 10–13 shows the method action binding and some of the attribute bindings created when you drop the `createServiceRequest` method.

Example 10–13 Method Action Binding in the Page Definition File

```
<bindings>
  <methodAction id="createServiceRequest" MethodName="createServiceRequest"
                RequiresUpdateModel="true" Action="999"
                DataControl="SRPublicFacade"
                InstanceName="SRPublicFacade.dataProvider"
                ReturnName="SRPublicFacade.methodResults.SRPublicFacade_
                         dataProvider_createServiceRequest_result">
    <NamedData NDName="problemDescription" NDType="java.lang.String"
               NDValue="${bindings.createServiceRequest_problemDescription}"/>
    <NamedData NDName="productId" NDType="java.lang.Integer"
               NDValue="${bindings.createServiceRequest_productId}"/>
    <NamedData NDName="createdBy" NDType="java.lang.Integer"
               NDValue="${bindings.createServiceRequest_createdBy}"/>
  </methodAction>
  <attributeValues id="problemDescription" IterBinding="variables">
    <AttrNames>
      <Item Value="createServiceRequest_problemDescription"/>
    </AttrNames>
    </attributeValues>
    <attributeValues id="productId" IterBinding="variables">
      <AttrNames>
        <Item Value="createServiceRequest_productId"/>
    </AttrNames>
      ...
  </attributeValues>
</bindings>
```

Note that like the attributes for a collection, attributes for a method also reference an iterator. However, instead of referencing a method iterator that accesses and iterates over the collection that the associated method returns, attributes for a creation-type method access and iterate over variables. Because this type of method has not returned an object, there is nothing to hold the values entered on the page. Variables act as the data holders.

JDeveloper creates a variable for each parameter the method takes. The variables are declared as children to the variable iterator, and are local, meaning they "live" only as long as the associated binding context. Example 10–14 shows the variable iterator and variables created when you use the `createServiceRequest(String, Integer, Integer)` method.

Example 10–14 Variable Iterator and Variables in the Page Definition File

```
<executables>
  <variableIterator id="variables">
    <variable Type="java.lang.String"
              Name="createServiceRequest_problemDescription"
              IsQueriable="false"/>
    <variable Type="java.lang.Integer" Name="createServiceRequest_productId"
              IsQueriable="false"/>
    <variable Type="java.lang.Integer" Name="createServiceRequest_createdBy"
              IsQueriable="false"/>
  </variableIterator>
</executables>
```

10.7.5 What Happens at Runtime

When the user enters data and submits the form, the variables are populated and the attribute binding can then provide the value for the method's parameters using the EL expression for the value of the `NamedData` element.

The service request creation process in the SRDemo application uses a process train, which causes the actual creation of the request to be spread out over three steps (for more information, see Section 11.5, "Creating a Multipage Process"). For the purposes of this explanation, assume the process is contained on one page. When the user enters a description in the corresponding `inputText` component and clicks the **Create Product** button, the following happens:

- The `problemDescriptionVar` variable is populated with the value the user entered.
- Because the `problemDescription` attribute binding refers to the variable iterator, and to the `problemDescriptionVar` variable specifically as its `Item` value, the attribute binding can get the description:

  ```
  <attributeValues id="problemDescription" IterBinding="variables">
    <AttrNames>
      <Item Value="createServiceRequest_problemDescription"/>
    </AttrNames>
  </attributeValues>
  ```

- Because the `Name` `NamedData` element has an EL expression that evaluates to the item value of the `problemDescription` attribute binding, it can also access the value:

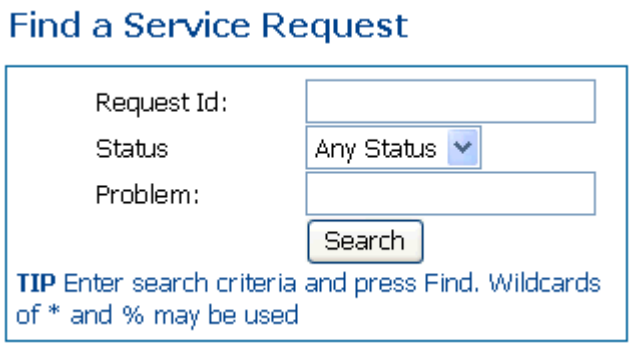
  ```
  <NamedData NDName="problemDescription" NDType="java.lang.String"
             NDValue="${bindings.createServiceRequest_problemDescription}"/>
  ```

- The `createServiceRequest` method is executed with each parameter taking its value from the corresponding `NamedData` element.

10.8 Creating Search Pages

You can create a search form using a method that finds records by taking parameters. The results can then be displayed in a table. In this type of search form, users must enter information for each parameter. Figure 10–10 shows the SRSearch search form used to find all service requests, given an ID, status, and a problem description.

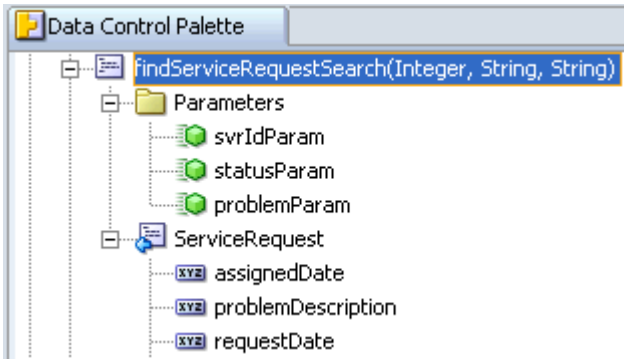
Figure 10–10 The SRSearch Form



10.8.1 How to Create a Search Form

You create search form by dropping an existing method that contains the logic to find and return the records based on parameters. This method must already exist on the data control. Figure 10–11 shows the `findServiceRequestSearch(Integer, String, String)` method used to create the search form shown in Figure 10–10. This method finds and returns all service requests given an ID, status, and description.

Figure 10–11 A Search Method That Takes Parameters in the Data Control Palette



To create the search form, you drop the method as a parameter form. You then drop the returned collection as a table to display the results. The SRSearch page hides the results table if it is the first time in a session that the user visits the page. For procedures for conditionally hiding the results table, see Section 10.9, "Conditionally Displaying the Results Table on a Search Page".

To create a search form and results table:

1. From the Data Control Palette, drag a find method that takes parameters.
2. From the context menu, choose **Parameters** > **ADF Parameter Form.**
3. From the Data Control Palette, drag the return of that method and drop it as any type of table.

10.8.2 What Happens When You Use Parameter Methods

When you drop a method as a parameter form, JDeveloper:

- Defines the following in the page definition file: variables to hold the data values, a method binding for the method, and the attribute bindings for the associated attributes.
- Inserts code in the JSF page for the form using ADF Faces `inputText` components and an ADF Faces `commandButton` component. This code is the same as code for any other input form or command button.

Just as when you drop a collection that is a return of a method, when you drop a method that takes parameters onto a JSF page, JDeveloper creates a method action binding. However, because the method requires parameters to run, JDeveloper also creates `NamedData` elements for each parameter. These represent the parameters of the method. Each is bound to a value binding for the corresponding attribute. These bindings allow the method to access the correct attribute's value for the parameter on execution.

For example, the `findServiceRequestSearch` method action binding contains a `NamedData` element for each of the parameters it takes. The `statusParam` `NamedData` element is bound to the `findServiceRequestSearch_statusParam` attribute binding using an EL expression. Example 10–13 shows the method action binding and some of the attribute bindings created when you drop the `findServiceRequestSearch` method as a parameter form.

Example 10–15 Method Action Binding in the Page Definition File

```
<bindings>
  <methodAction id="findServiceRequestSearch"
                    MethodName="findServiceRequestSearch"
                    RequiresUpdateModel="true" Action="999"
                    DataControl="SRPublicFacade"
                    InstanceName="SRPublicFacade.dataProvider"
                    ReturnName="SRPublicFacade.methodResults.SRPublicFacade_
                              dataProvider_findServiceRequestSearch_result">
    <NamedData NDName="svrIdParam" NDType="java.lang.Integer"
               NDValue="${bindings.findServiceRequestSearch_svrIdParam}"/>
    <NamedData NDName="statusParam" NDType="java.lang.String"
               NDValue="${bindings.findServiceRequestSearch_statusParam}"/>
    <NamedData NDName="problemParam" NDType="java.lang.String"
               NDValue="${bindings.findServiceRequestSearch_problemParam}"/>
  </methodAction>
...
  <attributeValues id="svrIdParam" IterBinding="variables">
    <AttrNames>
      <Item Value="findServiceRequestSearch_svrIdParam"/>
    </AttrNames>
  </attributeValues>
  <attributeValues id="problemParam" IterBinding="variables">
    <AttrNames>
      <Item Value="findServiceRequestSearch_problemParam"/>
    </AttrNames>
  </attributeValues>
...
</bindings>
```

Because you dropped the method and not the return, the attributes reference a variable iterator that accesses and iterates over variables instead of a method iterator that accesses and iterates over a collection. This is because the method (unlike the returned collection) does not need to access an instance of an object; therefore, there is nothing to hold the values entered on the page. Variables act as these data holders.

JDeveloper creates a variable for each method parameter. The variables are declared as children to the variable iterator, and are local, meaning they "live" only as long as the associated binding context. Example 10–16 shows the variable iterator and variables created when using the `findServiceRequestSearch` method. The variable iterator is used both by the form and by the button.

Example 10–16 Variable Iterator and Variables in the Page Definition File

```
<executables>
  <variableIterator id="variables">
    <variable Type="java.lang.Integer"
              Name="findServiceRequestSearch_svrIdParam" IsQueriable="false"/>
    <variable Type="java.lang.String"
              Name="findServiceRequestSearch_statusParam"
              IsQueriable="false"/>
    <variable Type="java.lang.String"
              Name="findServiceRequestSearch_problemParam"
              IsQueriable="false"/>
  </variableIterator>
...
</executables>
```

When you then drop the returned collection for the results table, JDeveloper adds a method iterator that iterates over the returned collection. Since the results are in a table, a table binding is also created. Example 10–17 shows the code generated for the method iterator and table binding.

Example 10–17 Page Definition Code for a Returned Collection

```
<executables>
  <variableIterator id="variables">
...
  </variableIterator>
  <methodIterator id="findServiceRequestSearchIter"
                  Binds="findServiceRequestSearch.result"
                  DataControl="SRPublicFacade" RangeSize="10"
                  BeanClass="oracle.srdemo.model.entities.ServiceRequest"/>
</executables>
<bindings>
...
  <table id="findAllServiceRequest1" IterBinding="resultsIterator">
    <AttrNames>
      <Item Value="assignedDate"/>
      <Item Value="problemDescription"/>
      <Item Value="requestDate"/>
      <Item Value="status"/>
      <Item Value="svrId"/>
    </AttrNames>
  </table>
...
</bindings>
```

Note that because the same method is used, when you drop the table, a new method binding is not created. For more information, see Section 7.2.2, "What Happens When You Use the Data Control Palette to Create a Table".

10.8.3 What Happens at Runtime

When the user enters data and submits the form, the variables are populated and the attribute binding can then provide the value for the method's parameters using the EL expression for the value of the `NamedDataElement`.

> **Tip:** When the search form and results table are on the same page, the first time a user accesses the page, the table displays all records from the iterator. You can make it so that the results table does not display until the user actually executes the search. For procedures, see Section 10.9, "Conditionally Displaying the Results Table on a Search Page".

When the user enters `Closed` as the status in the corresponding `inputText` component, and clicks the command button, the following happens:

- The `findServiceRequestSearch_status` variable is populated with the value `Closed`.
- Because the attribute binding refers to the variable iterator, the attribute binding can get the value for `status`:

```
<attributeValues id="status" IterBinding="variables">
  <AttrNames>
    <Item Value="findServiceRequestSearch_statusParam"/>
  </AttrNames>
</attributeValues>
```

- Because the `NamedData` element has an EL expression that evaluates to the item value of the attribute binding, the parameter can also access the value:

```
<NamedData NDName="status" NDType="java.lang.String"
    NDValue="${bindings.findServiceRequests_statusParam}"/>
```

- The `findServiceRequestSearch` method is executed with the parameters taking their values from the `NamedData` elements.
- The `findServiceRequestSearch` method returns a collection of records that match the parameter values.
- The `findServiceRequestSearchIter` iterator iterates over the collection, allowing the table to display the results. For more information about tables at runtime, see Section 7.2.2, "What Happens When You Use the Data Control Palette to Create a Table".

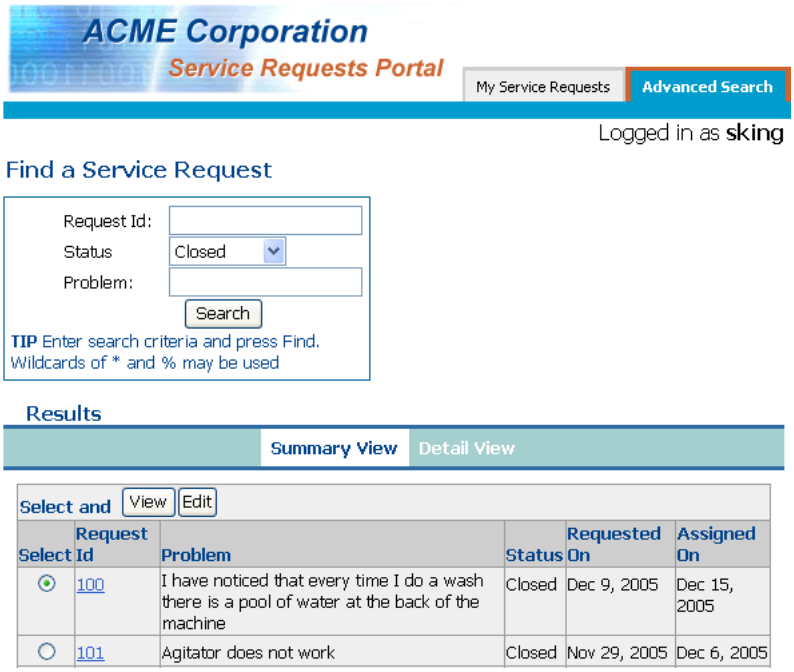
10.9 Conditionally Displaying the Results Table on a Search Page

When the search form and results table are on the same page, the first time a user accesses the page, the table displays all records from the iterator. You can make it so that the results table does not display until the user actually executes the search. Figure 10–12 shows the SRSearch page as it displays the first time a user accesses it.

Figure 10–12 Hidden Results Table for a Search Page

Once the user executes a search, the results table displays, as shown in Figure 10–13.

Figure 10–13 Results Table Displayed for a Search Page



10.9.1 How to Add Conditional Display Capabilities

To conditionally display the results table, you must enter an EL expression on the UI component (either the table itself or another component that holds the table component), that evaluates to whether this is the first time the user has accessed the search page. A field on a managed bean holds the value used in the expression.

To conditionally display the results table:

1. Create a search form and results table on the same page. For procedures, see Section 10.8, "Creating Search Pages".
2. Create a flag on a managed bean that will be set when the user accesses the page for the first time. For example, the `userState` managed bean in the SRDemo

application contains the `SEARCH_FIRSTTIME_FLAG` parameter. An EL expression on the page needs to know the value of this parameter to determine whether or not to render the page (see step 4). When the bean is instantiated for the EL expression, the `isSearchFirstTime` method then checks that field. If it is `null`, it sets the value to `True`. For information about creating managed beans, see Section 10.2, "Using a Managed Bean to Store Information"

3. On the JSF page, insert a `setActionListener` component into the command component used to execute this search. Set the `from` attribute to `#{false}`. Set the `to` attribute to the field on the managed bean created in step two. This will set that field to `false` whenever the button is clicked. For more information about using the `setActionListener` component, see Section 10.4, "Setting Parameter Values Using a Command Component".

   Example 10–18 shows the code for the **Search** button on the SRSearch page.

Example 10–18 Using a setActionListener Component to Set a Value

```
<af:commandButton actionListener="#{bindings.findServiceRequestSearch.execute}"
                  text="#{res['srsearch.searchLabel']}">
  <af:setActionListener from="#{false}"
                        to="#{userState.searchFirstTime}"/>
</af:commandButton>
```

4. On the JSF page, use an EL expression as the value of the `Rendered` attribute so that the UI component (the table or the UI component holding the table) only renders when the variable is a certain value.

   Example 10–19 shows the EL expression used for the value for the `Rendered` attribute of the `panelGroup` component on the SRSearch page.

Example 10–19 JSF Code to Conditionally Display the Search Results Table

```
<af:panelGroup rendered="#{!userState.searchFirstTime}">
```

   This EL expression causes the `panelGroup` component to render only if the `searchFirstTime` flag has a value of `False`.

10.9.2 What Happens When you Conditionally Display the Results Table

When you use a managed bean to hold a value, other objects can both set the value and access the value. For example, similar to passing parameter values, you can use the `setActionListener` component to set values on a managed bean that can then be accessed by an EL expression on the `rendered` attribute of a component.

For example, when a user accesses the SRSearch page for the first time, the following happens:

- Because the `panelGroup` component that holds the table contains an EL expression for it's `rendered` attribute, and the EL expression references the `userState` bean, that bean is instantiated.
- Because the user has not accessed page, the SEARCH_FIRSTTIME_FLAG field on the `userState` bean has not yet been set, and therefore has a value of `null`
- Because the value is `null`, the `isSearchFirstTime` method on that bean sets the value to `true`.
- The EL expression is evaluated, and because he SEARCH_FIRSTTIME_FLAG field is `true`, the SRSearch page displays without rendering the panel group, including the nested table.
- When the user enters search criteria and clicks the **Search** button, the associated `setActionListener` component sets the `searchFirstTime` value on the `userState` bean to `false`.
- Because there is no outcome defined for the command button, the user stays on the same page.
- Because the `searchFirstTime` value is now set to `false`, when the page rerenders with the results, the `panelGroup` component displays the table with the result.

11

Using Complex UI Components

This chapter describes how to use ADF Faces components to create some of the functionality in the SRDemo application.

This chapter includes the following sections:

- Section 11.1, "Introduction to Complex UI Components"
- Section 11.2, "Using Dynamic Menus for Navigation"
- Section 11.3, "Using Popup Dialogs"
- Section 11.4, "Enabling Partial Page Rendering"
- Section 11.5, "Creating a Multipage Process"
- Section 11.6, "Providing File Upload Capability"
- Section 11.7, "Creating Databound Dropdown Lists"
- Section 11.8, "Creating a Databound Shuttle"

11.1 Introduction to Complex UI Components

ADF Faces components simplify user interaction. For example, `inputFile` enables file uploading, and `selectInputText` has built-in dialog support for navigating to a popup window and returning to the initial page with the selected value. While most of the ADF Faces components can be used out-of-the-box with minimal Java coding, some of them require extra coding in backing beans and configuring in `faces-config.xml`.

While the SRDemo pages use a custom skin, the descriptions of the rendered UI components and the illustrations in this chapter follow the default Oracle skin.

Read this chapter to understand:

- How to create dynamic navigation menus using a menu model
- How to create popup dialogs using command components
- How to enable partial page rendering explicitly using partial triggers and events
- How to create a multipage process using a process train model
- How to provide file upload support
- How to create lists with static and dynamic list of values, and navigation list binding
- How to create a shuttle for displaying and moving list items

11.2 Using Dynamic Menus for Navigation

The SRDemo pages use a `panelPage` component to lay out the page with a hierarchical menu system for page navigation. Figure 11–1 shows the Management page with the available menu choices from the SRDemo application's menu hierarchy. Typically, a menu hierarchy consists of global buttons, menu tabs, and a menu bar beneath the menu tabs.

Figure 11–1 Dynamic Navigation Menus in the SRDemo Application

There are two ways to create a menu hierarchy, namely:

- Manually by inserting individual menu item components into each menu component, and marking the current menu items as "selected" on each page
- Declaratively by binding each menu component to a menu model object and using the menu model display the appropriate menu items, including setting the current items as "selected"

For most of the pages you see in the SRDemo application, the declarative technique is employed—using a menu model and managed beans—to dynamically generate the menu hierarchy.

The `panelPage` component supports `menu1` and `menu2` facets for creating the hierarchical, navigation menus that enable a user to go quickly to related pages in the application.

The `menu1` facet takes a `menuTabs` component, which lays out a series of menu items rendered as menu tabs. Similarly, the `menu2` facet takes a `menuBar` component that renders menu items in a bar beneath the menu tabs.

Global buttons are buttons that are always available from any page in the application, such as a Help button. The `menuGlobal` facet on `panelPage` takes a `menuButtons` component that lays out a series of buttons.

Note: The global buttons in the SRDemo application are not generated dynamically, instead they are hard-coded into each page. In some pages, *cacheable fragments* are used to contain the `menuTabs` and `menuBar` components. For purposes of explaining how to create dynamic menus in this chapter, global buttons are included and caching is excluded in the descriptions and code samples. For information about caching, see Chapter 15, "Optimizing Application Performance with Caching".

11.2.1 How to Create Dynamic Navigation Menus

To display hierarchical menus dynamically, you build a menu model and bind the menu components (such as `menuTabs` and `menuBar`) to the menu model. At runtime, the menu model generates the hierarchical menu choices for the pages.

To create dynamic navigation menus:

1. Create a menu model. (See Section 11.2.1.1, "Creating a Menu Model")
2. Create a JSF page for each menu choice or item in the menu hierarchy. (See Section 11.2.1.2, "Creating the JSF Page for Each Menu Item")
3. Create one global navigation rule that has navigation cases for each menu item. (See Section 11.2.1.3, "Creating the JSF Navigation Rules")

11.2.1.1 Creating a Menu Model

Use the `oracle.adf.view.faces.model.MenuModel`, `oracle.adf.view.faces.model.ChildPropertyTreeModel`, and `oracle.adf.view.faces.model.ViewIdPropertyMenuModel` classes to create a menu model that dynamically generates a menu hierarchy.

To create a menu model:

1. Create a class that can get and set the properties for each item in the menu hierarchy or tree.

   For example, each item in the tree needs to have a `label`, a `viewId`, and an `outcome` property. If items have children (for example, a menu tab item can have children menu bar items), you need to define a property to represent the list of children (for example, `children` property). To determine whether items are shown or not shown on a page depending on security roles, define a boolean property (for example, `shown` property). Example 11–1 shows the `MenuItem` class used in the SRDemo application.

Example 11–1 MenuItem.java for All Menu Items

```
package oracle.srdemo.view.menu;
import java.util.List;
import oracle.adf.view.faces.component.core.nav.CoreCommandMenuItem;
public class MenuItem {
    private String _label          = null;
    private String _outcome        = null;
    private String _viewId         = null;
    private String _destination    = null;
    private String _icon           = null;
    private String _type           = CoreCommandMenuItem.TYPE_DEFAULT;
    private List   _children       = null;
    //extended security attributes
    private boolean _readOnly = false;
    private boolean _shown = true;
    public void setLabel(String label) {
        this._label = label;
    }
    public String getLabel() {
        return _label;
    }
 // getter and setter methods for remaining attributes omitted
}
```

> **Note:** The `type` property defines a menu item as global or nonglobal. Global items can be accessed from any page in the application. For example, a Help button on a page is a global item.

2. Configure a managed bean for each menu item or page in the hierarchy, with values for the properties that require setting at instantiation.

   Each bean should be an instance of the menu item class you create in step 1. Example 11–2 shows the managed bean code for all the menu items in the SRDemo application. If an item has children items, the list entries are the children managed beans listed in the order you desire. For example, the **Management** menu tab item has two children.

   Typically each bean should have `none` as its bean scope. The SRDemo application, however, uses `session` scoped managed beans for the menu items because security attributes are assigned to the menu items when they are created dynamically, and the SRDemo application uses a `session` scoped `UserInfo` bean to hold the user role information for the user currently logged in. The user role information is used to determine which menu items a user sees when logged in. For example, only users with the user role of 'manager' see the **Management** menu tab. JSF doesn't let you reference a `session` scoped managed bean from a `none` scoped bean; therefore, the SRDemo application uses all `session` scoped managed beans for the menu system.

Example 11–2 Managed Beans for Menu Items in the faces-config.xml File

```
<!-- If you were to use dynamically generated global buttons -->
<!-- Root pages: Two global button menu items -->
<managed-bean>
  <managed-bean-name>menuItem_GlobalLogout</managed-bean-name>
  <managed-bean-class>oracle.srdemo.view.menu.MenuItem</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
  <managed-property>
    <property-name>label</property-name>
    <value>#{resources['srdemo.menu.logout']}</value>
  </managed-property>
  <managed-property>
    <property-name>icon</property-name>
    <value>/images/logout.gif</value>
  </managed-property>
  <managed-property>
    <property-name>type</property-name>
    <value>global</value>
  </managed-property>
  <managed-property>
    <property-name>viewId</property-name>
    <value>/app/SRLogout.jsp</value>
  </managed-property>
  <managed-property>
    <property-name>outcome</property-name>
    <value>GlobalLogout</value>
  </managed-property>
</managed-bean>
```

```
<managed-bean>
  <managed-bean-name>menuItem_GlobalHelp</managed-bean-name>
  <managed-bean-class>oracle.srdemo.view.menu.MenuItem</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
  <managed-property>
    <property-name>label</property-name>
    <value>#{resources['srdemo.menu.help']}</value>
  </managed-property>
  <managed-property>
    <property-name>icon</property-name>
    <value>/images/help.gif</value>
  </managed-property>
  <managed-property>
    <property-name>type</property-name>
    <value>global</value>
  </managed-property>
  <managed-property>
    <property-name>viewId</property-name>
    <value>/app/SRHelp.jspx</value>
  </managed-property>
  <managed-property>
    <property-name>outcome</property-name>
    <value>GlobalHelp</value>
  </managed-property>
</managed-bean>

<!-- Root pages: Four menu tabs -->
<!-- 1. My Service Requests menu tab item -->
<managed-bean>
  <managed-bean-name>menuItem_MyServiceRequests</managed-bean-name>
  <managed-bean-class>oracle.srdemo.view.menu.MenuItem</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
  <managed-property>
    <property-name>label</property-name>
    <value>#{resources['srdemo.menu.my']}</value>
  </managed-property>
  <managed-property>
    <property-name>viewId</property-name>
    <value>/app/SRList.jspx</value>
  </managed-property>
  <managed-property>
    <property-name>outcome</property-name>
    <value>GlobalHome</value>
  </managed-property>
</managed-bean>

<!-- 2. Advanced Search menu tab item -->
<managed-bean>
  <managed-bean-name>menuItem_AdvancedSearch</managed-bean-name>
  <managed-bean-class>oracle.srdemo.view.menu.MenuItem</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
  <managed-property>
    <property-name>label</property-name>
    <value>#{resources['srdemo.menu.advanced']}</value>
  </managed-property>
  <managed-property>
    <property-name>shown</property-name>
    <value>#{userInfo.staff}</value>
  </managed-property>
```

```
  <managed-property>
    <property-name>viewId</property-name>
    <value>/app/staff/SRSearch.jspx</value>
  </managed-property>
  <managed-property>
    <property-name>outcome</property-name>
    <value>GlobalSearch</value>
  </managed-property>
</managed-bean>

<!-- 3. New Service Request menu tab item -->
<managed-bean>
  <managed-bean-name>menuItem_New</managed-bean-name>
  <managed-bean-class>oracle.srdemo.view.menu.MenuItem</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
  <managed-property>
    <property-name>label</property-name>
    <value>#{resources['srdemo.menu.new']}</value>
  </managed-property>
  <managed-property>
    <property-name>viewId</property-name>
    <value>/app/SRCreate.jspx</value>
  </managed-property>
  <managed-property>
    <property-name>outcome</property-name>
    <value>GlobalCreate</value>
  </managed-property>
</managed-bean>

<!-- 4. Management menu tab item -->
<!-- This managed bean uses managed bean chaining for children menu items -->
<managed-bean>
  <managed-bean-name>menuItem_Manage</managed-bean-name>
  <managed-bean-class>oracle.srdemo.view.menu.MenuItem</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
  <managed-property>
    <property-name>label</property-name>
    <value>#{resources['srdemo.menu.manage']}</value>
  </managed-property>
  <managed-property>
    <property-name>shown</property-name>
    <value>#{userInfo.manager}</value>
  </managed-property>
  <managed-property>
    <property-name>viewId</property-name>
    <value>/app/management/SRManage.jspx</value>
  </managed-property>
  <managed-property>
    <property-name>outcome</property-name>
    <value>GlobalManage</value>
  </managed-property>
  <managed-property>
    <property-name>children</property-name>
    <list-entries>
      <value-class>oracle.srdemo.view.menu.MenuItem</value-class>
      <value>#{subMenuItem_Manage_Reporting}</value>
      <value>#{subMenuItem_Manage_ProdEx}</value>
    </list-entries>
  </managed-property>
</managed-bean>
```

```
<!-- Children menu bar items for Management tab -->
<managed-bean>
  <managed-bean-name>subMenuItem_Manage_Reporting</managed-bean-name>
  <managed-bean-class>oracle.srdemo.view.menu.MenuItem</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
  <managed-property>
    <property-name>label</property-name>
    <value>#{resources['srdemo.menu.manage.reporting']}</value>
  </managed-property>
  <managed-property>
    <property-name>shown</property-name>
    <value>#{userInfo.manager}</value>
  </managed-property>
  <managed-property>
    <property-name>viewId</property-name>
    <value>/app/management/SRManage.jspx</value>
  </managed-property>
  <managed-property>
    <property-name>outcome</property-name>
    <value>GlobalManage</value>
  </managed-property>
</managed-bean>
<managed-bean>
  <managed-bean-name>subMenuItem_Manage_ProdEx</managed-bean-name>
  <managed-bean-class>oracle.srdemo.view.menu.MenuItem</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
  <managed-property>
    <property-name>label</property-name>
    <value>#{resources['srdemo.menu.manage.prodEx']}</value>
  </managed-property>
  <managed-property>
    <property-name>shown</property-name>
    <value>#{userInfo.manager}</value>
  </managed-property>
  <managed-property>
    <property-name>viewId</property-name>
    <value>/app/management/SRSkills.jspx</value>
  </managed-property>
  <managed-property>
    <property-name>outcome</property-name>
    <value>Skills</value>
  </managed-property>
</managed-bean>
```

Note: As you see in Figure 11–1, the **Management** menu tab has a menu bar with two items: **Overview** and **Technician Skills**. As each menu item has its own page or managed bean, so the two items are represented by these managed beans, respectively: `subMenuItem_Manage_Reporting` and `subMenuItem_Manage_ProdEx`. The **Management** menu tab is represented by the `menuItem_Manage` managed bean, which uses value binding expressions (such as `#{subMenuItem_Manage_ProdEx}`) inside the list `value` elements to reference the children managed beans.

3. Create a class that constructs a `ChildPropertyTreeModel` instance. The instance represents the entire tree hierarchy of the menu system, which is later injected into a menu model. Example 11–3 shows the `MenuTreeModelAdapter` class used in the SRDemo application.

Example 11–3 MenuTreeModelAdapter.java for Holding the Menu Tree Hierarchy

```
package oracle.srdemo.view.menu;
import java.beans.IntrospectionException;
import java.util.List;
import oracle.adf.view.faces.model.ChildPropertyTreeModel;
import oracle.adf.view.faces.model.TreeModel;

public class MenuTreeModelAdapter {
    private String _propertyName = null;
    private Object _instance = null;
    private transient TreeModel _model = null;

    public TreeModel getModel() throws IntrospectionException
    {
      if (_model == null)
      {
        _model = new ChildPropertyTreeModel(getInstance(), getChildProperty());
      }
      return _model;
    }

    public String getChildProperty()
    {
      return _propertyName;
    }
    /**
     * Sets the property to use to get at child lists
     * @param propertyName
     */
    public void setChildProperty(String propertyName)
    {
      _propertyName = propertyName;
      _model = null;
    }

    public Object getInstance()
    {
      return _instance;
    }
    /**
     * Sets the root list for this tree.
     * @param instance must be something that can be converted into a List
     */
    public void setInstance(Object instance)
    {
      _instance = instance;
      _model = null;
    }
```

```
    /**
     * Sets the root list for this tree.
     * This is needed for passing a List when using the managed bean list
     * creation facility, which requires the parameter type of List.
     * @param instance the list of root nodes
     */
    public void setListInstance(List instance)
    {
      setInstance(instance);
    }
}
```

4. Configure a managed bean to reference the menu tree model class in step 3. The bean should be instantiated with a `childProperty` value that is the same as the property value that represents the list of children as created on the bean in step 1.

   The bean should also be instantiated with a list of root pages (listed in the order you desire) as the value for the `listInstance` property. The root pages are the global button menu items and the first-level menu tab items, as shown in Example 11–2. Example 11–4 shows the managed bean for creating the menu tree model.

Example 11–4 Managed Bean for Menu Tree Model in the faces-config.xml File

```
<managed-bean>
  <managed-bean-name>menuTreeModel</managed-bean-name>
  <managed-bean-class>
    oracle.srdemo.view.menu.MenuTreeModelAdapter
  </managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
  <managed-property>
    <property-name>childProperty</property-name>
    <value>children</value>
  </managed-property>
  <managed-property>
    <property-name>listInstance</property-name>
    <list-entries>
      <value-class>oracle.srdemo.view.menu.MenuItem</value-class>
      <value>#{menuItem_GlobalLogout}</value>
      <value>#{menuItem_GlobalHelp}</value>
      <value>#{menuItem_MyServiceRequests}</value>
      <value>#{menuItem_AdvancedSearch}</value>
      <value>#{menuItem_New}</value>
      <value>#{menuItem_Manage}</value>
    </list-entries>
  </managed-property>
</managed-bean>
```

5. Create a class that constructs a `ViewIdPropertyMenuModel` instance. The instance creates a menu model from the menu tree model. Example 11–5 shows the `MenuModelAdapter` class used in the SRDemo application.

Example 11–5 MenuModelAdapter.java

```
package oracle.srdemo.view.menu;
import java.beans.IntrospectionException;
import java.io.Serializable;
import java.util.List;
import oracle.adf.view.faces.model.MenuModel;
import oracle.adf.view.faces.model.ViewIdPropertyMenuModel;
```

```
public class MenuModelAdapter implements Serializable {
    private           String    _propertyName = null;
    private           Object    _instance = null;
    private transient MenuModel _model = null;
    private           List      _aliasList = null;

    public MenuModel getModel() throws IntrospectionException
    {
      if (_model == null)
      {
        ViewIdPropertyMenuModel model =
                              new ViewIdPropertyMenuModel(getInstance(),
                                                          getViewIdProperty());

        if(_aliasList != null && !_aliasList.isEmpty())
        {
          int size = _aliasList.size();
          if (size % 2 == 1)
            size = size - 1;

          for ( int i = 0; i < size; i=i+2)
          {
            model.addViewId(_aliasList.get(i).toString(),
                            _aliasList.get(i+1).toString());
          }
        }

        _model = model;
      }
      return _model;
    }

    public String getViewIdProperty()
    {
      return _propertyName;
    }
    /**
     * Sets the property to use to get at view id
     * @param propertyName
     */
    public void setViewIdProperty(String propertyName)
    {
      _propertyName = propertyName;
      _model = null;
    }

    public Object getInstance()
    {
      return _instance;
    }
    /**
     * Sets the treeModel
     * @param instance must be something that can be converted into a TreeModel
     */
    public void setInstance(Object instance)
    {
      _instance = instance;
      _model = null;
    }
```

```
    public List getAliasList()
    {
      return _aliasList;
    }
    public void setAliasList(List aliasList)
    {
      _aliasList = aliasList;
    }
}
```

6. Configure a managed bean to reference the menu model class in step 5. This is the bean to which all the menu components on a page are bound.

   The bean should be instantiated with the `instance` property value set to the `model` property of the menu tree model bean configured in step 4. The instantiated bean should also have the `viewIdProperty` value set to the `viewId` property on the bean created in step 1. Example 11–6 shows the managed bean code for creating the menu model.

Example 11–6 Managed Bean for Menu Model in the faces-config.xml File

```
<!-- create the main menu menuModel -->
<managed-bean>
  <managed-bean-name>menuModel</managed-bean-name>
  <managed-bean-class>
    oracle.srdemo.view.menu.MenuModelAdapter</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
  <managed-property>
    <property-name>viewIdProperty</property-name>
    <value>viewId</value>
  </managed-property>
  <managed-property>
    <property-name>instance</property-name>
    <value>#{menuTreeModel.model}</value>
  </managed-property>
</managed-bean>
```

11.2.1.1.1 What You May Need to Know About Chaining Managed Beans

By using value binding expressions to chain managed bean definitions, you can create a tree-like menu system instead of a flat structure. The order of the individual managed bean definitions in `faces-config.xml` does not matter, but the order of the children `list-entries` in a parent bean should be in the order you want the menu choices to appear.

When you chain managed bean definitions together, the bean scopes must be compatible. Table 11–1 lists the compatible bean scopes.

Table 11–1 Combinations of Managed Bean Scopes Allowed

A bean of this scope...	Can chain with beans of these scopes
`none`	`none`
`application`	`none`, `application`
`session`	`none`, `application`, `session`
`request`	`none`, `application`, `session`, `request`

11.2.1.1.2 What You May Need to Know About Accessing Resource Bundle Strings

The `String` resources for all labels in the SRDemo application are contained in a resource bundle. This resource bundle is configured in `faces-config.xml`. As described earlier, each menu item is defined as a `session` scoped managed bean, and the various attributes of a menu item (such as its type and label) are defined through managed bean properties. For the menu item managed bean to access the label to use from the resource bundle, you need to configure a managed bean that provides the access to the bundle.

In the SRDemo application, the `ResourceAdapter` class exposes the resource bundle within EL expressions via the `resources` managed bean. Example 11–7 shows the `ResourceAdapter` class, and the `JSFUtils.getStringFromBundle()` method that retrieves a `String` from the bundle.

Example 11–7 Part of ResourceAdapter.java and Part of JSFUtils.java

```
package oracle.srdemo.view.resources;
import oracle.srdemo.view.util.JSFUtils;
/**
 * Utility class that allows us to expose the specified resource bundle within
 * general EL
 */
public class ResourceAdapter implements Map {

    public Object get(Object resourceKey) {
        return JSFUtils.getStringFromBundle((String)resourceKey);
    }
 // Rest of file omitted from here
}
...
/** From JSFUtils.java */
package oracle.srdemo.view.util;
import java.util.MissingResourceException;
import java.util.ResourceBundle;
...
public class JSFUtils {
    private static final String NO_RESOURCE_FOUND = "Missing resource: ";
    /**
       * Pulls a String resource from the property bundle that
       * is defined under the application's message-bundle element in
       * faces-config.xml. Respects Locale.
       * @param key
       * @return Resource value or placeholder error String
       */
    public static String getStringFromBundle(String key) {
        ResourceBundle bundle = getBundle();
        return getStringSafely(bundle, key, null);
    }
```

```
    /*
     * Internal method to proxy for resource keys that don't exist
     */
    private static String getStringSafely(ResourceBundle bundle, String key,
                                          String defaultValue) {
      String resource = null;
      try {
        resource = bundle.getString(key);
      } catch (MissingResourceException mrex) {
        if (defaultValue != null) {
          resource = defaultValue;
        } else {
          resource = NO_RESOURCE_FOUND + key;
        }
      }
      return resource;
    }
//Rest of file omitted from here
}
```

Example 11–8 shows the `resources` managed bean code that provides the access for other managed beans to the `String` resources.

Example 11–8 Managed Bean for Accessing the Resource Bundle Strings

```
<!-- Resource bundle -->
<application>
  <message-bundle>oracle.srdemo.view.resources.UIResources</message-bundle>
  ...
</application>

<!-- Managed bean for ResourceAdapater class -->
<managed-bean>
  <managed-bean-name>resources</managed-bean-name>
  <managed-bean-class>
    oracle.srdemo.view.resources.ResourceAdapter</managed-bean-class>
  <managed-bean-scope>application</managed-bean-scope>
</managed-bean>
```

The `resources` managed bean defines a `Map` interface onto the resource bundle that is defined in `faces-config.xml`. The menu item labels automatically pick up the correct language strings.

> **Tip:** The menu model is built when it is first referenced. This means it is not rebuilt if the browser language is changed within a single session.

11.2.1.2 Creating the JSF Page for Each Menu Item

Each menu item (whether it is a menu tab item, menu bar item, or global button) has its own page. To display the available menu choices on a page, bind the menu components (such as `menuTabs`, `menuBar`, or `menuButtons`) to the menu model. Example 11–9 shows the `menuTabs` component code that binds the component to a menu model.

Example 11–9 MenuTabs Component Bound to a Menu Model

```
<af:panelPage title="#{res['srmanage.pageTitle']}"
              binding="#{backing_SRManage.panelPage1}"
              id="panelPage1">
  <f:facet name="menu1">
    <af:menuTabs value="#{menuModel.model}"...>
      ...
    </af:menuTabs>
  </f:facet>
  ...
</af:panelPage>
```

Each menu component has a `nodeStamp` facet, which takes one `commandMenuItem` component, as shown in Example 11–10. By using a variable and binding the menu component to the model, you need only one `commandMenuItem` component to display all items in a menu, which is accomplished by using an EL expression similar to `#{var.label}` for the `text` value, and `#{var.getOutcome}` for the `action` value on the `commandMenuItem` component. It is the `commandMenuItem` component that provides the actual label you see on a menu item, and the navigation outcome when the menu item is activated.

Example 11–10 NodeStamp Facet and CommandMenuItem Component

```
<af:panelPage title="#{res['srmanage.pageTitle']}"
              binding="#{backing_SRManage.panelPage1}"
              id="panelPage1">
  <f:facet name="menu1">
    <af:menuTabs var="menuTab"
                 value="#{menuModel.model}">
      <f:facet name="nodeStamp">
        <af:commandMenuItem text="#{menuTab.label}"
                            action="#{menuTab.getOutcome}"
                            .../>
      </f:facet>
    </af:menuTabs>
  </f:facet>
  ...
</af:panelPage>
```

Whether a menu item renders on a page is determined by the security role of the current user logged in. For example, only users with the manager role see the **Management** menu tab. The `rendered` and `disabled` attributes on a `commandMenuItem` component determine whether a menu item should be rendered or disabled.

Following along with the `MenuItem` class in Example 11–1: For global items, bind the `rendered` attribute to the variable's `type` property and set it to `global`. For nonglobal items, bind the `rendered` attribute to the variable's `shown` property and the `type` property, and set the `type` property to `default`. For nonglobal items, bind also the `disabled` attribute to the variable's `readOnly` property. Example 11–11 shows how this is done for `menuTabs` (a nonglobal component) and `menuButtons` (a global component).

Example 11–11 Rendered and Disabled Menu Item Components

```
<af:menuTabs var="menuTab" value="#{menuModel.model}">
  <f:facet name="nodeStamp">
    <af:commandMenuItem text="#{menuTab.label}"
                        action="#{menuTab.getOutcome}"
                        rendered="#{menuTab.shown and
                                    menuTab.type=='default'}"
                        disabled="#{menuTab.readOnly}"/>
  </f:facet>
</af:menuTabs>
...
<af:menuButtons var="menuOption" value="#{menuModel.model}">
  <f:facet name="nodeStamp">
    <af:commandMenuItem text="#{menuOption.label}"
                        action="#{menuOption.getOutcome}"
                        rendered="#{menuOption.type=='global'}"
                        icon="#{menuOption.icon}"/>
  </f:facet>
</af:menuButtons>
```

You can use any combination of menus you desire in an application. For example, you could use only menu bars, without any menu tabs. To let ADF Faces know the start level of your menu hierarchy, you set the `startDepth` attribute on the menu component. Based on a zero-based index, the possible values of `startDepth` are 0, 1, and 2, assuming three levels of menus are used. If `startDepth` is not specified, it defaults to zero (0).

If an application uses global menu buttons, menu tabs, and menu bars: A global `menuButtons` component always has a `startDepth` of zero. Since menu tabs are the first level, the `startDepth` for `menuTabs` is zero as well. The `menuBar` component then has a `startDepth` value of 1. Example 11–12 shows part of the menu code for a `panelPage` component.

Example 11–12 PanelPage Component with Menu Facets

```
<af:panelPage title="#{res['srmanage.pageTitle']}">
  <f:facet name="menu1">
    <af:menuTabs var="menuTab" value="#{menuModel.model}">
      <f:facet name="nodeStamp">
        <af:commandMenuItem text="#{menuTab.label}"
                            action="#{menuTab.getOutcome}"
                            rendered="#{menuTab.shown and
                                        menuTab.type=='default'}"
                            disabled="#{menuTab.readOnly}"/>
          </f:facet>
    </af:menuTabs>
  </f:facet>
  <f:facet name="menu2">
    <af:menuBar var="menuSubTab" startDepth="1"
                value="#{menuModel.model}">
      <f:facet name="nodeStamp">
        <af:commandMenuItem text="#{menuSubTab.label}"
                            action="#{menuSubTab.getOutcome}"
                            rendered="#{menuSubTab.shown and
                                        menuSubTab.type=='default'}"
                            disabled="#{menuSubTab.readOnly}"/>
      </f:facet>
    </af:menuBar>
  </f:facet>
```

```
  <f:facet name="menuGlobal">
    <af:menuButtons var="menuOption" value="#{menuModel.model}">
      <f:facet name="nodeStamp">
        <af:commandMenuItem text="#{menuOption.label}"
                            action="#{menuOption.getOutcome}"
                            rendered="#{menuOption.type=='global'}"
                            icon="#{menuOption.icon}"/>
      </f:facet>
    </af:menuButtons>
  </f:facet>
  ...
</af:panelPage>
```

Tip: If your menu system uses menu bars as the first level, then the `startDepth` on `menuBar` should be set to zero, and so on.

11.2.1.2.1 What You May Need to Know About the PanelPage and Page Components

Instead of using a `panelPage` component and binding each menu component on the page to a menu model object, you can use the `page` component with a menu model. By value binding the `page` component to a menu model, as shown in the following code snippet, you can take advantage of the more flexible rendering capabilities of the page component. For example, you can easily change the look and feel of menu components by creating a new renderer for the `page` component. If you use the `panelPage` component, you need to change the renderer for each of the menu components.

```
<af:page title="Title 1" var="node" value="#{menuModel.model}">
  <f:facet name="nodeStamp">
    <af:commandMenuItem text="#{node.label}"
                        action="#{node.getOutcome}"
                        type="#{node.type}"/>
  </f:facet>
</af:page>
```

Because a menu model dynamically determines the hierarchy (that is, the links that appear in each menu component) and also sets the current items in the focus path as "selected," you can use practically the same code on each page.

11.2.1.3 Creating the JSF Navigation Rules

Create one global navigation rule that has navigation cases for each first-level and global menu item. Children menu items are not included in the global navigation rule. For menu items that have children menu items (for example, the **Management** menu tab has children menu bar items), create a navigation rule with all the navigation cases that are possible from the parent item, as shown in Example 11–13.

Example 11–13 Navigation Rules for a Menu System in the faces-config.xml File

```
<navigation-rule>
  <from-view-id>*</from-view-id>
  <navigation-case>
    <from-outcome>GlobalHome</from-outcome>
    <to-view-id>/app/SRList.jspx</to-view-id>
    <redirect/>
  </navigation-case>
  <navigation-case>
    <from-outcome>GlobalSearch</from-outcome>
    <to-view-id>/app/staff/SRSearch.jspx</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>GlobalCreate</from-outcome>
    <to-view-id>/app/SRCreate.jspx</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>GlobalManage</from-outcome>
    <to-view-id>/app/management/SRManage.jspx</to-view-id>
    <redirect/>
  </navigation-case>
  <navigation-case>
    <from-outcome>GlobalLogout</from-outcome>
    <to-view-id>/app/SRLogout.jspx</to-view-id>
    <redirect/>
  </navigation-case>
  <navigation-case>
    <from-outcome>GlobalAbout</from-outcome>
    <to-view-id>/app/SRAbout.jspx</to-view-id>
  </navigation-case>
</navigation-rule>
<!-- Navigation rule for Management menu tab with children items -->
<navigation-rule>
  <from-view-id>/app/management/SRManage.jspx</from-view-id>
  <navigation-case>
    <from-outcome>Skills</from-outcome>
    <to-view-id>/app/management/SRSkills.jspx</to-view-id>
  </navigation-case>
</navigation-rule>
```

11.2.2 What Happens at Runtime

`MenuModelAdapter` constructs the menu model, which is a `ViewIdPropertyMenuModel` instance, via the `menuModel` managed bean. When the `menuTreeModel` bean is requested, this automatically triggers the creation of the chained beans `menuItem_GlobalLogout`, `menuItem_GlobalHelp`, `menuItem_MyServiceRequests`, and so on. The tree of menu items is injected into the menu model. The menu model provides the model that correctly highlights and enables the items on the menus as you navigate through the menu system.

The individual menu item managed beans (for example, `menuItem_MyServiceRequests`) are instantiated with values for `label`, `viewId`, and `outcome` that are used by the menu model to dynamically generate the menu items. The default JSF `actionListener` mechanism uses the `outcome` values to handle the page navigation.

Each menu component has a `nodeStamp` facet, which is used to stamp the different menu items in the menu model. The `commandMenuItem` component housed within the `nodeStamp` facet provides the text and action for each menu item.

Each time nodeStamp is stamped, the data for the current menu item is copied into an EL reachable property. The name of this property is defined by the var attribute on the menu component that houses the nodeStamp facet. Once the menu has completed rendering, this property is removed (or reverted back to its previous value). In Example 11–14, the data for each menu bar item is placed under the EL property menuSubTab. The nodeStamp displays the data for each item by getting further properties from the menuSubTab property.

Example 11–14 MenuBar Component Bound to a Menu Model

```
<af:menuBar var="menuSubTab" startDepth="1"
            value="#{menuModel.model}">
  <f:facet name="nodeStamp">
    <af:commandMenuItem text="#{menuSubTab.label}"
                        action="#{menuSubTab.getOutcome}"
                        rendered="#{menuSubTab.shown and
                                    menuSubTab.type=='default'}"
                        disabled="#{menuSubTab.readOnly}"/>
  </f:facet>
</af:menuBar>
```

By binding a menu component to a menu model and using a variable to represent a menu item, you need only one commandMenuItem component to display all menu items at that hierarchy level, allowing for more code reuse between pages, and is much less error prone than manually inserting a commandMenuItem component for each item. For example, if menu is the variable, then EL expressions such as #{menu.label} and #{menu.getOutcome} specify the text and action values for a commandMenuItem component.

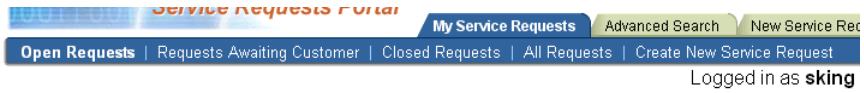
The menu model in conjunction with nodeStamp controls whether a menu item is rendered as selected. As described earlier, a menu model is created from a tree model, which contains viewId information for each node. ViewIdPropertyMenuModel, which is an instance of MenuModel, uses the viewId of a node to determine the focus rowKey. Each item in the menu model is stamped based on the current rowKey. As the user navigates and the current viewId changes, the focus path of the model also changes and a new set of items is accessed. MenuModel has a method getFocusRowKey() that determines which page has focus, and automatically renders a node as selected if the node is on the focus path.

11.2.3 What You May Need to Know About Menus

Sometimes you might want to create menus manually instead of using a menu model.

The first-level menu tab **My Service Requests** has one second-level menu bar with several items, as illustrated in Figure 11–2. From **My Service Requests**, you can view open, pending, closed, or all service requests, represented by the first, second, third, and fourth menu bar item from the left, respectively. Each view is actually generated from the SRList.jspx page.

Figure 11–2 Menu Bar Items on My Service Requests Page (SRList.jspx)



In the SRList.jspx page, instead of binding the menuBar component to a menu model and using a nodeStamp to generate the menu items, you use individual children commandMenuItem components to display the menu items because the command components require a value to determine the type of requests to navigate to (for example, open, pending, closed, or all service requests). Example 11–15 shows part of the code for the menuBar component used in the SRList.jspx page.

Example 11–15 MenuBar Component with Children CommandMenuItem Components

```
<af:menuBar>
  <af:commandMenuItem text="#{res['srlist.menubar.openLink']}"
                      disabled="#{!bindings.findServiceRequests.enabled}"
                      selected="#{userState.listModeOpen}"
                      actionListener="#{bindings.findServiceRequests.execute}">
    <af:setActionListener from="#{'Open'}"
                          to="#{userState.listMode}"/>
  </af:commandMenuItem>
  <af:commandMenuItem text="#{res['srlist.menubar.pendingLink']}"
                      disabled="#{!bindings.findServiceRequests.enabled}"
                      selected="#{userState.listModePending}"
                      actionListener="#{bindings.findServiceRequests.execute}"
    <af:setActionListener from="#{'Pending'}"
                          to="#{userState.listMode}"/>
  </af:commandMenuItem>
  ...
  <af:commandMenuItem text="#{res['srlist.menubar.allRequests']}"
                      selected="#{userState.listModeAll}"
                      disabled="#{!bindings.findServiceRequests.enabled}"
                      actionListener="#{bindings.findServiceRequests.execute}">
    <af:setActionListener from="#{'%'}"
                          to="#{userState.listMode}"/>
  </af:commandMenuItem>
  ...
</af:menuBar>
```

The af:setActionListener tag, which declaratively sets a value on an ActionSource component before navigation, passes the correct list mode value to the userState managed bean. The session scoped userState managed bean stores the current list mode of the page.

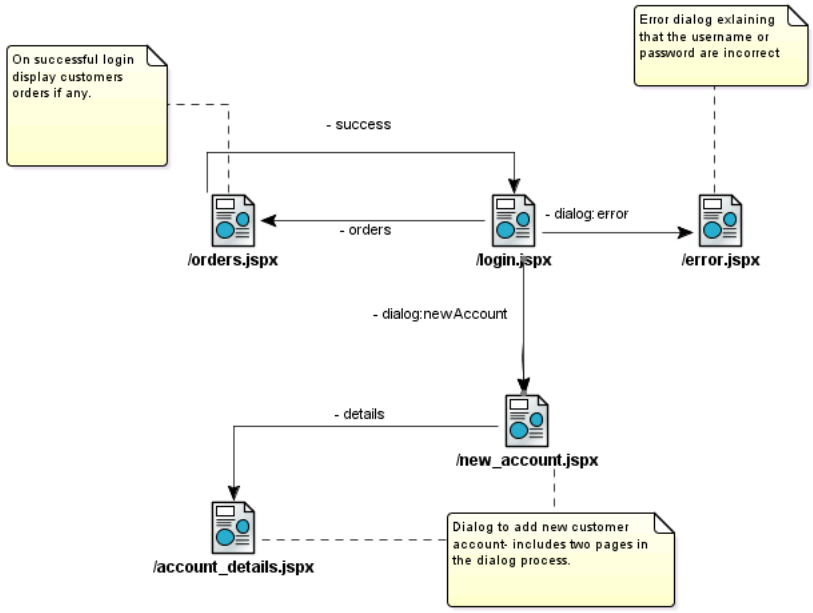
When the commandMenuItem component is activated, the findServiceRequests method executes with the list mode value, and returns a collection that matches the value. The commandMenuItem components also use convenience functions in the userSystemState bean to evaluate whether the menu item should be marked as selected or not.

11.3 Using Popup Dialogs

Sometimes you might want to display a new page in a separate popup dialog instead of displaying it in the same window containing the current page. In the popup dialog, you might let the user enter or select information, and then return to the original page to use that information. Ordinarily, you would need to use JavaScript to launch the popup dialog and manage the process, and create code for managing cases where popup dialogs are not supported on certain client devices such as a PDA. With the dialog framework, ADF Faces has made it easy to launch and manage popup dialogs and processes without using JavaScript.

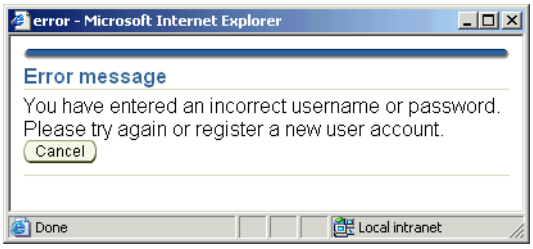
Consider a simple application that requires users to log in to see their orders. Figure 11–3 shows the page flow for the application, which consists of five pages—`login.jspx`, `orders.jspx`, `new_account.jspx`, `account_details.jspx`, and `error.jspx`.

Figure 11–3 Page Flow of a Dialog Sample Application



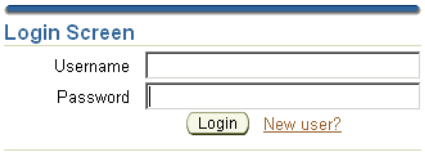
When an existing user logs in successfully, the application displays the Orders page, which shows the user's orders, if there are any. When a user does not log in successfully, the Error page displays in a popup dialog, as shown in Figure 11–4.

Figure 11–4 Error Page in a Popup Dialog



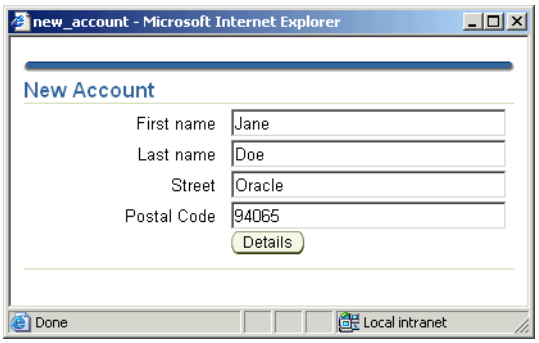
On the Error page there is a **Cancel** button. When the user clicks **Cancel**, the popup dialog closes and the application returns to the Login page, as shown in Figure 11–5.

Figure 11–5 Login Page



When a new user clicks the **New User** link on the Login page, the New Account page displays in a popup dialog, as shown in Figure 11–6.

Figure 11–6 New Account Page in a Popup Dialog

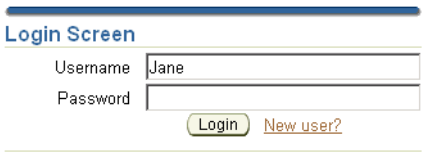


After entering information such as first name and last name, the user then clicks the **Details** button to display the Account Details page in the same popup dialog, as shown in Figure 11–7. In the Account Details page, the user enters other information and confirms a password for the new login account. There are two buttons on the Account Details page—**Cancel** and **Done**.

Figure 11–7 Account Details Page in a Popup Dialog

If the new user decides not to proceed with creating a new login account and clicks **Cancel**, the popup dialog closes and the application returns to the Login page. If the new user clicks **Done**, the popup dialog closes and the application returns to the Login page where the **Username** field is now populated with the user's first name, as shown in Figure 11–8. The new user can then proceed to enter the new password and log in successfully.

Figure 11–8 Login Page With the Username Field Populated



11.3.1 How to Create Popup Dialogs

To make it easy to support popup dialogs in your application, ADF Faces has built in the dialog functionality to components that implement `ActionSource` (such as `commandButton` and `commandLink`). For ADF Faces to know whether to launch a page in a popup dialog from an `ActionSource` component, four conditions must exist:

- There must be a JSF navigation rule with an outcome that begins with "dialog:".
- The command component's action outcome must begin with "dialog:".
- The `useWindow` attribute on the command component must be "true".
- The client device must support popup dialogs.

> **Note:** If `useWindow` is `false` or if the client device doesn't support popup dialogs, ADF Faces automatically shows the page in the current window instead of using a popup—code changes are not needed to facilitate this.

The page that displays in a popup dialog is an ordinary JSF page. But for purposes of explaining how to implement popup dialogs in this chapter, a page that displays in a popup dialog is called the *dialog page*, and a page from which the popup dialog is launched is called the *originating page*. A *dialog process* starts when the originating page launches a dialog (which can contain one dialog page or a series of dialog pages), and ends when the user dismisses the dialog and is returned to the originating page.

The tasks for supporting popup dialogs in an application are:

1. Define a JSF navigation rule for launching a dialog.
2. Create the JSF page from which a dialog is launched.
3. Create the dialog page and return a dialog value.
4. Handle the return value.
5. Pass a value into a dialog.

The tasks can be performed in any order.

11.3.1.1 Defining a JSF Navigation Rule for Launching a Dialog

You manage the navigation into a popup dialog by defining a standard JSF navigation rule with a special `dialog:` outcome. Using the dialog sample application shown in Figure 11–3, three navigation outcomes are possible from the Login page:

- Show the Orders page in the same window (successful login)
- Show the Error dialog page in a popup dialog (login failure)
- Show the New Account dialog page in a popup dialog (new user)

Example 11–16 shows the navigation rule for the three navigation cases from the Login page (`login.jspx`).

Example 11–16 Dialog Navigation Rules in the faces-config.xml File

```
<navigation-rule>

  <!-- Originating JSF page -->
  <from-view-id>/login.jspx</from-view-id>

  <!-- Navigation case for the New Account dialog page (new user)-->
  <navigation-case>
    <from-outcome>dialog:newAccount</from-outcome>
    <to-view-id>/new_account.jspx</to-view-id>
  </navigation-case>

  <!-- Navigation case for the Error dialog page (upon login failure) -->
  </navigation-case>
    <from-outcome>dialog:error</from-outcome>
    <to-view-id>/error.jspx</to-view-id>
  </navigation-case>

  <!-- Navigation case for the Orders page (upon login success) -->
  </navigation-case>
    <from-outcome>orders</from-outcome>
    <to-view-id>/orders.jspx</to-view-id>
  </navigation-case>

</navigation-rule>
```

11.3.1.1.1 What Happens at Runtime

The dialog navigation rules on their own simply show the specified pages in the main window. But when used with command components with `dialog:` action outcomes and with `useWindow` attributes set to `true`, ADF Faces knows to launch the pages in popup dialogs. This is described in the next step.

11.3.1.2 Creating the JSF Page That Launches a Dialog

In the originating page from which a popup dialog is launched, you can use either an action method or a static action outcome on the `ActionSource` component. Whether you specify a static action outcome or use an action method that returns an action outcome, this action outcome must begin with `dialog:`.

The sample application uses an action method binding on the `commandButton` component to determine programmatically whether to navigate to the Orders page or the Error dialog page, and a static action outcome on the `commandLink` component to navigate directly to the New Account dialog page. Both command components are on the Login page. Example 11–17 shows the code for the Login `commandButton` component.

Example 11–17 Login Button on the Login Page

```
af:commandButton id="cmdBtn"
                 text="Login"
                 action="#{backing_login.commandButton_action}"
                 useWindow="true"
                 windowHeight="200"
                 windowWidth="500"
                 partialSubmit="true"/>
```

The attributes `useWindow`, `windowHeight`, and `windowWidth` are used in launching pages in popup dialogs. These attributes are ignored if the client device doesn't support popup dialogs.

When `useWindow="true"` ADF Faces knows to launch the dialog page in a new popup dialog. The `windowHeight` and `windowWidth` attributes specify the size of the popup dialog.

> **Tip:** Set the `partialSubmit` attribute on the `commandButton` component to `true`. This prevents the originating page from refreshing (and hence flashing momentarily) when the popup dialog displays.

The `action` attribute on `commandButton` specifies a reference to an action method in the page's backing bean, `Login.java`. The action method must return an outcome string, which JSF uses to determine the next page to display by comparing the outcome string to the outcomes in the navigation cases defined in `faces-config.xml`. The code for this action method is shown in Example 11–18.

Example 11–18 Action Method Code for the Login Button

```
public String commandButton_action()
{
  String retValue;
  retValue = "orders";
  _cust = getListCustomer();
  if (_cust == null || !password.equals(_cust.getPassword()))
  {
    retValue = "dialog:error";
  }

  return retValue;
}
```

Example 11–19 shows the code for the **New User** `commandLink` component that uses a static action outcome.

Example 11–19 New User Command Link on the Login Page

```
<af:commandLink id="cmdLink"
                text="New User?"
                action="dialog:newAccount"
                useWindow="true"
                partialSubmit="true"
                windowHeight="200"
                windowWidth="500" />
```

Instead of referencing an action method, the `action` attribute value is simply a static outcome string that begins with `dialog:`.

11.3.1.2.1 What Happens at Runtime

ADF Faces uses the attribute `useWindow="true"` in conjunction with an action outcome that begins with `dialog:` to determine whether to start a dialog process and launch a page in a popup dialog (assuming `dialog:` navigation rules have been defined in `faces-config.xml`).

If the action outcome does not begin with `dialog:`, ADF Faces does not start a process or launch a popup dialog even when `useWindow="true"`. Conversely, if the action outcome begins with `dialog:`, ADF Faces does not launch a popup dialog if `useWindow="false"` or if `useWindow` is not set, but ADF Faces does start a new process.

If the client device does not support popup dialogs, ADF Faces shows the dialog page in the current window after preserving all the state of the current page—you don't have to write any code to facilitate this.

When a command component is about to launch a dialog, it delivers a launch event (`LaunchEvent`). The launch event stores information about the component that is responsible for launching a popup dialog, and the root of the component tree to display when the dialog process starts. A launch event can also pass a map of parameters into the dialog. For more information, see Section 11.3.1.5, "Passing a Value into a Dialog".

11.3.1.3 Creating the Dialog Page and Returning a Dialog Value

The dialog pages in our sample application are the Error page, the New Account page, and the Account Details page. The dialog process for a new user actually contains two pages: the New Account page and the Account Details page. The dialog process for a user login failure contains just the Error page.

A dialog page is just like any other JSF page, with one exception. In a dialog page you must provide a way to tell ADF Faces when the dialog process finishes, that is, when the user dismisses the dialog. Generally, you do this programmatically or declaratively via a command component. Example 11–20 shows how to accomplish this programmatically via a **Cancel** button on the Error page.

Example 11–20 Cancel Button on the Error Page

```
<af:commandButton text="Cancel"
                  actionListener="#{backing_error.cancel}" />
```

The `actionListener` attribute on `commandButton` specifies a reference to an action listener method in the page's backing bean, `Error.java`. The action listener method processes the action event that is generated when the **Cancel** button is clicked. You call the `AdfFacesContext.returnFromDialog()` method in this action listener method, as shown in Example 11–21.

Example 11–21 Action Listener Method for the Cancel Button in a Backing Bean

```
public void cancel(ActionEvent actionEvent)
{
  AdfFacesContext.getCurrentInstance().returnFromDialog(null, null);
}
```

> **Note:** The `AdfFacesContext.returnFromDialog()` method returns `null`. This is all that is needed in the backing bean to handle the Cancel action event.

To accomplish the same declaratively on the Account Details dialog page, attach a `af:returnActionListener` tag to the **Cancel** button component, as shown in Example 11–22. The `af:returnActionListener` tag calls the `returnFromDialog` method on the `AdfFacesContext`—no backing bean code is needed.

Example 11–22 Cancel Button on the Account Details Page

```
<af_commandButton text="Cancel" immediate="true">
  <af:returnActionListener/>
</af:commandButton>
```

No attributes are used with the `af:returnActionListener` tag. The `immediate` attribute on `commandButton` is set to `true`: if the user clicks **Cancel** without entering values in the required **Password** and **Confirm Password** fields, the default JSF `ActionListener` can execute during the Apply Request Values phase instead of the Invoke Application phase, thus bypassing input validation.

The New Account page and Account Details page belong in the same dialog process. A dialog process can have as many pages as you desire, but you only need to call `AdfFacesContext.returnFromDialog()` once.

The same `af:returnActionListener` tag or `AdfFacesContext.returnFromDialog()` method can also be used to end a process and return a value from the dialog. For example, when the user clicks **Done** on the Account Details page, the process ends and returns the user input values. Example 11–23 shows the code for the **Done** button.

Example 11–23 Done Button on the Account Details Page

```
<af:commandButton text="Done"
                  actionListener="#{backing_new_account.done}" />
```

The `actionListener` attribute on `commandButton` specifies a reference to an action listener method in the page's backing bean, `New_account.java`. The action listener method processes the action event that is generated when the **Done** button is clicked. Example 11–24 shows the code for the action listener method, where the return value is retrieved, and then returned via the `AdfFacesContext.returnFromDialog()` method.

Example 11–24 Action Listener Method for the Done Button in a Backing Bean

```
public void done(ActionEvent e)
{
  AdfFacesContext afContext = AdfFacesContext.getCurrentInstance();
  String firstname = afContext.getProcessScope().get("firstname").toString();
  String lastname = afContext.getProcessScope().get("lastname").toString();
  String street = afContext.getProcessScope().get("street").toString();
  String zipCode = afContext.getProcessScope().get("zipCode").toString();
  String country = afContext.getProcessScope().get("country").toString();
  String password = afContext.getProcessScope().get("password").toString();
  String confirmPassword =
   afContext.getProcessScope().get("confirmPassword").toString();
  if (!password.equals(confirmPassword))
  {
    FacesMessage fm = new FacesMessage();
    fm.setSummary("Confirm Password");
    fm.setDetail("You've entered an incorrect password. Please verify that you've
     entered a correct password!");
    FacesContext.getCurrentInstance().addMessage(null, fm);
  }
  else
  {
```

```
    //Get the return value
    Customer cst = new Customer();
    cst.setFirstName(firstname);
    cst.setLastName(lastname);
    cst.setStreet(street);
    cst.setPostalCode(zipCode);
    cst.setCountry(country);
    cst.setPassword(password);
    // And return it
    afContext.getCurrentInstance().returnFromDialog(cst, null);
    afContext.getProcessScope().clear();
  }
}
```

The `AdfFacesContext.returnFromDialog()` method lets you send back a return value in the form of a `java.lang.Object` or a `java.util.Map` of parameters. You don't have to know where you're returning the value to—ADF Faces automatically takes care of it.

11.3.1.3.1 What Happens at Runtime

The `AdfFacesContext.returnFromDialog()` method tells ADF Faces when the user dismisses the dialog. This method can be called whether the dialog page is shown in a popup dialog or in the main window. If a popup dialog is used, ADF Faces automatically closes it.

In the sample application, when the user clicks the **Cancel** button on the Error page or Account Details page, ADF Faces calls `AdfFacesContext.returnFromDialog()`, (which returns `null`), closes the popup dialog, and returns to the originating page.

The first page in the new user dialog process is the New Account page. When the **Details** button on the New Account page is clicked, the application shows the Account Details dialog page in the same popup dialog (because `useWindow="false"`), after preserving the state of the New Account page.

When the **Done** button on the Account Details page is clicked, ADF Faces closes the popup dialog and `AdfFacesContext.returnFromDialog()` returns `cst` to the originating page.

When the dialog is dismissed, ADF Faces generates a return event (`ReturnEvent`). The `AdfFacesContext.returnFromDialog()` method sends a return value as a property of the return event. The return event is delivered to the return listener (`ReturnListener`) that is registered on the command component that launched the dialog (which would be the **New User** `commandLink` on the Login page). How you would handle the return value is described in Section 11.3.1.4, "Handling the Return Value".

11.3.1.4 Handling the Return Value

To handle a return value, you register a return listener on the command component that launched the dialog, which would be the **New User** link component on the Login page in the sample application. Example 11–25 shows the code for the **New User** link component.

Example 11–25 New User Command Link on the Login Page

```
<af:commandLink id="cmdLink" text="New User?"
                action="dialog:newAccount"
                useWindow="true" partialSubmit="true"
                returnListener="#{backing_login.handleReturn}"
                windowHeight="200" windowWidth="500" />
```

The `returnListener` attribute on `commandLink` specifies a reference to a return listener method in the page's backing bean, `Login.java`. The return listener method processes the return event that is generated when the dialog is dismissed. Example 11–26 shows the code for the return listener method that handles the return value.

Example 11–26 Return Listener Method for the New User Link in a Backing Bean

```
public void handleReturn(ReturnEvent event)
{
  if (event.getReturnValue() != null)
  {
    Customer cst;
    String name;
    String psw;
    cst = (Customer)event.getReturnValue();
    name = cst.getFirstName();
    psw = cst.getPassword();
    CustomerList.getCustomers().add(cst);
    inputText1.setSubmittedValue(null);
    inputText1.setValue(name);
    inputText2.setSubmittedValue(null);
    inputText2.setValue(psw);
  }
}
```

You use the `getReturnValue()` method to retrieve the return value, because the return value is automatically added as a property of the `ReturnEvent`.

11.3.1.4.1 What Happens at Runtime

In the sample application, when ADF Faces delivers a return event to the return listener registered on the `commandLink` component, the `handleReturn()` method is called and the return value is processed accordingly. The new user is added to a customer list, and as a convenience to the user any previously submitted values in the Login page are cleared and the input fields are populated with the new information.

11.3.1.5 Passing a Value into a Dialog

The `AdfFacesContext.returnFromDialog()` method lets you send a return value back from a dialog. Sometimes you might want to pass a value into a dialog. To pass a value into a dialog, you use a launch listener (`LaunchListener`).

In the sample application, a new user can enter a name in the **Username** field on the Login page, and then click the **New User** link. When the New Account dialog page displays in a popup dialog, the **First Name** input field is automatically populated with the name that was entered in the Login page. To accomplish this, you register a launch listener on the command component that launched the dialog (which would be `commandLink`). Example 11–27 shows the code for the `commandLink` component.

Example 11–27 Input Field and New User Command Link on the Login Page

```
<af:inputText label="Username" value="#{backing_login.username}"/>
<af:commandLink id="cmdLink" text="New User?"
                action="dialog:newAccount"
                useWindow="true" partialSubmit="true"
                launchListener="#{backing_login.handleLaunch}"
                returnListener="#{backing_login.handleReturn}"
                windowHeight="200" windowWidth="500" />
```

The `LaunchListener` attribute on `commandLink` specifies a reference to a launch listener method in the page's backing bean, `Login.java`. In the launch listener method you use the `getDialogParameters()` method to add a parameter to a `Map` using a key-value pair. Example 11–28 shows the code for the launch listener method.

Example 11–28 Launch Listener Method for the New User Command Link in a Backing Bean

```
public void handleLaunch(LaunchEvent event)
{
  //Pass the current value of the field into the dialog
  Object usr = username;
  event.getDialogParameters().put("firstname", usr);
}
// Use by inputText value binding
public String username;
public String getUsername()
{
  return username;
}
public void setUsername(String username)
{
  this.username = username;
}
```

To show the parameter value in the New Account dialog page, use the ADF Faces `processScope` to retrieve the key and value via a special EL expression in the format `#{processScope.someKey}`, as shown in Example 11–29.

Example 11–29 Input Field on the New Account Page

```
<af:inputText label="First name" value="#{processScope.firstname}"/>
```

> **Note:** You can use `processScope` with all JSF components, not only with ADF Faces components.

11.3.1.5.1 What Happens at Runtime

When a command component is about to launch a dialog (assuming all conditions have been met), ADF Faces queues a launch event. This event stores information about the component that is responsible for launching a dialog, and the root of the component tree to display when the dialog process starts. Associated with a launch event is a launch listener, which takes the launch event as a single argument and processes the event as needed.

In the sample application, when ADF Faces delivers the launch event to the launch listener registered on the `commandLink` component, the `handleLaunch()` method is called and the event processed accordingly.

In ADF Faces, a process always gets a copy of all the values that are in the `processScope` of the page from which a dialog is launched. When the `getDialogParameters()` method has added parameters to a `Map`, those parameters also become available in `processScope`, and any page in the dialog process can get the values out of `processScope` by referring to the `processScope` objects via EL expressions.

Unlike `sessionScope`, `processScope` values are visible only in the current "page flow" or process. If the user opens a new window and starts navigating, that series of windows has its own process; values stored in each window remain independent. Clicking on the browser's Back button automatically resets `processScope` to its original state. When you return from a process the `processScope` is back to the way it was before the process started. To pass values out of a process you would use `AdfFacesContext.returnFromDialog()`, `sessionScope` or `applicationScope`.

11.3.2 How the SRDemo Popup Dialogs Are Created

The SRDemo application uses a popup dialog to:

- Display a list of frequently asked questions (FAQ).
- Select and assign a technician to an open service request.

In the Create New Service Request page (see Figure 11–13), when the user clicks the **Frequently Asked Questions** link, the application displays a popup dialog showing the FAQ list.

In the Edit Service Request page, when the user clicks the flashlight icon next to the **Assigned to** label (see Figure 11–12), the application displays the **Search for Staff** popup dialog. In the dialog (as shown in Figure 11–9), the user first makes a search based on user role. Then in the results section, the user clicks the radio button next to a name and clicks **Select**.

Figure 11–9 Search for Staff Popup Dialog (SRStaffSearch.jspx)

After making a selection, the popup dialog closes and the application returns to the Edit Service Request page where the **Assigned to** display-only fields are now updated with the selected technician's first name and last name, as shown in Figure 11–10.

Figure 11–10 Edit Service Request Page (SREdit.jspx) With an Assigned Request

To reiterate, the tasks for supporting a popup dialog are (not listed in any particular order):

1. Create the JSF navigation rules with `dialog:` outcomes.
2. Create the page that launches the dialog via a `dialog:` action outcome.
3. Create the dialog page and return a value.

4. Handle the return value.

Firstly, the JSF navigation rules for launching dialogs are shown in Example 11–30. The navigation case for showing the dialog page `SRStaffSearch.jspx` is defined by the `dialog:StaffSearch` outcome; the navigation case for showing the `SRFaq.jspx` dialog page is defined by the `dialog:FAQ` outcome.

Example 11–30 Dialog Navigation Rules in the faces-config.xml File

```
<navigation-rule>
  <from-view-id>/app/staff/SREdit.jspx</from-view-id>
  ...
  <navigation-case>
    <from-outcome>dialog:StaffSearch</from-outcome>
    <to-view-id>/app/staff/SRStaffSearch.jspx</to-view-id>
  </navigation-case>
</navigation-rule>
<navigation-rule>
  <from-view-id>/app/SRCreate.jspx</from-view-id>
  <navigation-case>
    <from-outcome>dialog:FAQ</from-outcome>
    <to-view-id>/app/SRFaq.jspx</to-view-id>
  </navigation-case>
  ...
</navigation-rule>
```

Secondly, the pages that launch popup dialogs are `SREdit.jspx` and `SRCreate.jspx`. In both pages the `useWindow` attribute on the `commandLink` component is set to `true`, which is a precondition for ADF Faces to know that it has to launch a popup dialog.

Example 11–31 shows the `commandLink` component on the page that launches the `SRStaffSearch.jspx` dialog page. The `commandLink` component has the static action outcome `dialog:StaffSearch`.

Example 11–31 CommandLink Component for Launching the SRStaffSearch Dialog Page

```
<af:commandLink id="staffLOVLink" action="dialog:StaffSearch"
                useWindow="true" immediate="true"
                partialSubmit="true"
                returnListener="#{backing_SREdit.handleStaffLOVReturn}"..>
  <af:objectImage height="24" width="24"
                  source="/images/searchicon_enabled.gif"/>
</af:commandLink>
```

Example 11–32 shows the `commandLink` component on the page that launches the `SRFaq.jspx` dialog page. The `commandLink` component has the static action outcome `dialog:SRFaq`.

Example 11–32 CommandLink Component for Launching the SRFaq Dialog Page

```
<af:commandLink action="dialog:FAQ"
                text="#{res['srcreate.faqLink']}"
                useWindow="true"
                immediate="true"
                partialSubmit="true"/>
```

Thirdly, the dialog pages `SRStaffSearch.jspx` and `SRFaq.jspx` have to call the `AdfFacesContext.returnFromDialog()` method to let ADF Faces know when the user dismisses the dialogs. In `SRStaffSearch.jspx`, which uses a `table` component with a `tableSelectOne` component to display the names for selection, the `AdfFacesContext.returnFromDialog()` method is called when the user clicks the **Select** `commandButton` component after selecting the radio button for a technician in the table. The `action` attribute on `commandButton` is bound to the `selectButton_action` action method in the page's backing bean (`SRStaffSearch.java`); the action method retrieves the selected row data from the table, extracts the User object, and then returns the object via the `AdfFacesContext.returnFromDialog()` method. Example 11–33 shows the code snippets for the **Select** button component and its action method.

Example 11–33 Action Method for the Select Command Button

```
<af:tableSelectOne>
  <af:commandButton text="#{res['srstaffsearch.button.select']}"
                    action="#{backing_SRStaffSearch.selectButton_action}"/>
</af:tableSelectOne>

...
...
public String selectButton_action() {

  //get row data from table
  JUCtrlValueBindingRef selectedRowData =
   (JUCtrlValueBindingRef)this.getResultsTable().getSelectedRowData();
  RowImpl row = (RowImpl)selectedRowData.getRow();
  User staffMember = (User)row.getDataProvider();

  // And return it
  AdfFacesContext.getCurrentInstance().returnFromDialog(staffMember, null);
  // no navigation to another page and thus null is returned
  return null;
}
```

Similarly in `SRFaq.jspx`, a `commandLink` component is used to close the dialog and call the `AdfFacesContext.returnFromDialog()` method. The `af:returnActionListener` tag calls the `returnFromDialog` method on the `AdfFacesContext`—backing bean code is not needed. Example 11–34 shows the code snippet for the `commandLink`. When the user dismisses the `SRFaq.jspx` popup dialog, ADF Faces simply closes the dialog. No dialog return value is sent, so there's no need to handle a return value.

Example 11–34 CommandLink Component for Closing the SRFaq Popup Dialog

```
<af:commandLink text="#{res['srdemo.close']}">
  <af:returnActionListener/>
</af:commandLink>
```

When the SRStaffSearch.jspx popup dialog is dismissed, a dialog return value (that is, the selected row data) is sent as a property of the return event (ReturnEvent). The return event is delivered to the return listener registered on the commandLink component of the originating page SREdit.jspx, as shown in Example 11–35. The returnListener attribute on commandLink is bound to the handleStaffLOVReturn listener method in the page's backing bean (SREdit.java). The return listener method handles the return value from the dismissed dialog. Example 11–35 also shows the code snippet for the handleStaffLOVReturn listener method.

Example 11–35 Return Listener Method for Handling the Return Value

```
<af:commandLink id="staffLOVLink" action="dialog:StaffSearch"
                useWindow="true" immediate="true"
                partialSubmit="true"
                returnListener="#{backing_SREdit.handleStaffLOVReturn}"..>
  <af:objectImage height="24" width="24"
                  source="/images/searchicon_enabled.gif"/>
</af:commandLink>
...
...
  public void handleStaffLOVReturn(ReturnEvent event) {
    //Get the return value from the pop up
    User returnedStaffMember = (User)event.getReturnValue();

    if (returnedStaffMember != null) {
      DCBindingContainer bc = (DCBindingContainer)getBindings();

      // Get the handle to the Service Request we are editing
      DCControlBinding thisSRId =
        (DCControlBinding)bc.getControlBinding("svrId");
      RowImpl srRowImpl = (RowImpl)thisSRId.getCurrentRow();
      ServiceRequest thisSR = (ServiceRequest)srRowImpl.getDataProvider();

      //See if a different user has been selected?
      User oldUser = thisSR.getAssignedTo();
      if ((oldUser == null) || (!oldUser.equals(returnedStaffMember))) {

        //Set the returned Staff member from the LOV
        thisSR.setAssignedTo(returnedStaffMember);

        //now re-execute the iterator to refresh the screen
        DCControlBinding accessorData =
          (DCControlBinding)bc.getControlBinding("assignedToFirstName");
        accessorData.getDCIteratorBinding().executeQuery();

        //Now reset the Assigned date
        ADFUtils.setPageBoundAttributeValue(getBindings(), "assignedDate",
                                      new Timestamp(System.currentTimeMillis()));

        //And get the data field to update with the new bound value
       this.getAssignedDate().resetValue();

      }
    }
  }
```

11.3.3 What You May Need to Know About ADF Faces Dialogs

The ADF Faces dialog framework has these known limitations:

- Does not support the use of `</redirect>` in navigation rules that may launch dialog pages in new popup dialogs. You can, however, use `</redirect>` in navigation rules that launch dialog pages within the same window.
- Cannot detect popup blockers. If you use popup dialogs in your web application, tell your users to disable popup blocking for your site.

11.3.4 Other Information

The ADF Faces select input components (such as `selectInputText` and `selectInputDate`) also have built-in dialog support. These components automatically handle launching a page in a popup dialog, and receiving the return event. For example, when you use `selectInputText` to launch a dialog, all you have to do is to set the `action` attribute to a `dialog:` outcome, and specify the width and height of the dialog. When the user dismisses the dialog, the return value from the dialog is automatically used as the new value of the input component. You would still need to define a JSF navigation rule with the `dialog:` outcome, create the dialog page, and create the dialog page's backing bean to handle the action events.

Besides being able to launch popup dialogs from action events, you can also launch popup dialogs from value change events and poll events. For example, you can programmatically launch a dialog (without a JSF navigation rule) by using the `AdfFacesContext.launchDialog()` method in a value change listener method or poll listener method.

If you're a framework or component developer you can enable a custom renderer to launch a dialog and handle a return value, or add `LaunchEvent` and `ReturnEvent` events support to your custom `ActionSource` components. For details about the `DialogService` API that you can use to implement dialogs, see the ADF Faces Javadoc for `oracle.adf.view.faces.context.DialogService`. See also the *ADF Faces Developer's Guide* for further information about supporting dialogs in custom components and renderers.

11.4 Enabling Partial Page Rendering

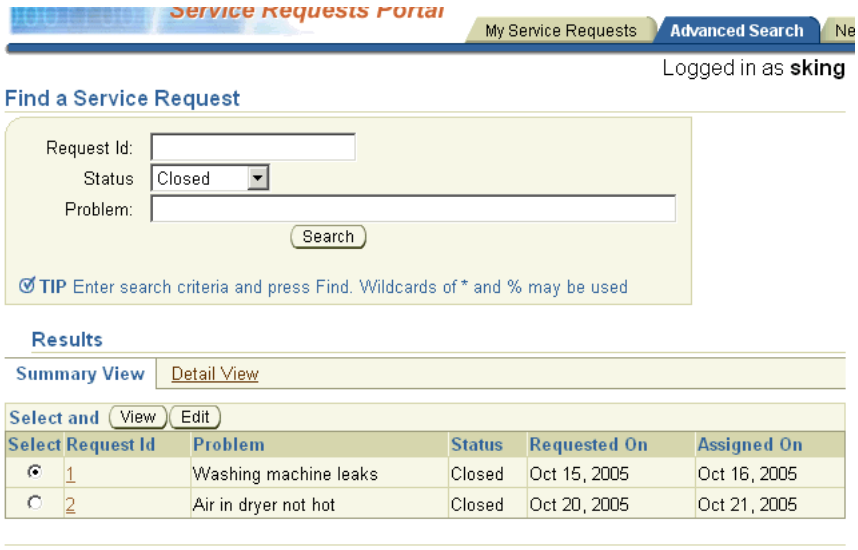
ADF Faces components use partial page rendering (PPR), which allows small areas of a page to be refreshed without the need to redraw the entire page. PPR is the same as AJAX-style browser user interfaces that update just parts of the page for a more interactive experience. PPR is currently supported on the following browsers:

- Internet Explorer 5.5 and above (Windows)
- Mozilla 1.0/Netscape 7.0

On all other platforms, ADF Faces automatically uses full page rendering You don't need to disable PPR or write code to support both cases.

Most of the time you don't have to do anything to enable PPR because ADF Faces components have built-in support for PPR. For example, in the `SRSearch.jspx` page, the **Results** section of the page uses a `showOneTab` component with two `showDetailItem` components to let the user display either a summary view or detail view of the search results. Figure 11–11 shows the **Results** section with the **Summary View** selected. When the user clicks **Detail View**, only the portion of the page that is below the **Results** title will refresh.

Figure 11–11 Search Page (SRSearch.jspx) with the Summary Result View Selected



At times you want to explicitly refresh parts of a page yourself. For example, you may want an output component to display what a user has chosen or entered in an input component, or you may want a command link or button to update another component. Three main component attributes can be used to enable partial page rendering:

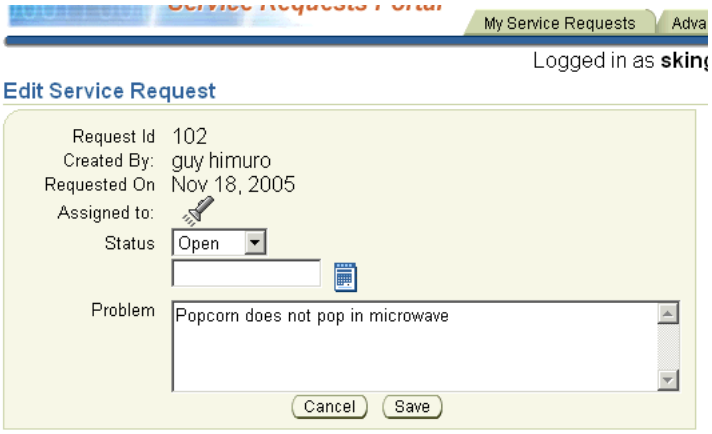
- `autoSubmit`: When the `autoSubmit` attribute of an input component (such as `inputText` and `selectOneChoice`) or a table select component (such as `tableSelectOne`) is set to `true`, and an appropriate action takes place (such as a value change), the component automatically submits the form it is enclosed in. For PPR, you might use this in conjunction with a listener attribute bound to a method that performs some logic when an event based on the submit is launched.
- `partialSubmit`: When the `partialSubmit` attribute of a command component is set to `true`, the page partially submits when the button or link is clicked. You might use this in conjunction with an `actionListener` method that performs some logic when the button or link is clicked.
- `partialTriggers`: All rendered components support the `partialTriggers` attribute. The value of this attribute is one or more IDs of other trigger components. When those trigger components are updated (for example through an automatic submit or a partial submit), the target component is also updated.

11.4.1 How to Enable PPR

The `SREdit.jspx` page of the SRDemo application uses partial page submits and partial triggers to support PPR.

Figure 11–12 shows the `SREdit.jspx` page with an unassigned service request. When the user clicks the flashlight icon (which is a `commandLink` component with an `objectImage` component), a popup dialog displays to allow the user to search and select a name. After selecting a name, the popup dialog closes and the **Assigned to** display-only fields (`outputText` components) and the date field below **Status** (`selectInputDate` component) are refreshed with the appropriate values; other parts of the edit page are not refreshed.

Figure 11–12 Edit Service Request Page (SREdit.jspx) with an Unassigned Request



To enable a command component to partially refresh another component:

1. On the trigger command component, set the `id` attribute to a unique value, and set the `partialSubmit` attribute to `true`.
2. On the target component that you want to partially refresh when the trigger command component is activated, set the `partialTriggers` attribute to the id of the command component.

> **Tip:** A component's unique ID must be a valid XML name, that is, you cannot use leading numeric values or spaces in the ID. JSF also does not permit colons (:) in the ID.

Example 11–36 shows the code snippets for the command and read-only output components used in the `SREdit.jspx` page to illustrate PPR.

Example 11–36 Code for Enabling Partial Page Rendering Through a Partial Submit

```
<af:panelLabelAndMessage label="#{res['sredit.assignedTo.label']}">
  <af:panelHorizontal>
    <af:outputText value="#{bindings.assignedToFirstName.inputValue}"
                   partialTriggers="staffLOVLink"/>
    <af:outputText value="#{bindings.assignedToLastName.inputValue}"
                  partialTriggers="staffLOVLink"/>
    <af:commandLink id="staffLOVLink" action="dialog:StaffSearch"
                    useWindow="true" immediate="true"
                    partialSubmit="true"
                    returnListener="#{backing_SREdit.handleStaffLOVReturn}"
                    partialTriggers="status"
                    disabled="#{bindings.ServiceRequeststatus.inputValue==2}">
      <af:objectImage height="24" width="24"
                      source="/images/searchicon_enabled.gif"/>
    </af:commandLink>
    <f:facet name="separator">
      <af:objectSpacer width="4" height="10"/>
    </f:facet>
  </af:panelHorizontal>
</af:panelLabelAndMessage>
```

> **Tip:** The `partialTriggers` attribute on a target component can contain the id of one or more trigger components. Use spaces to separate multiple ids.

11.4.2 What Happens at Runtime

ADF Faces command buttons and links can generate partial events. The `partialSubmit` attribute on `commandButton` or `commandLink` determines whether a partial page submit is used to perform an action or not. When `partialSubmit` is `true`, ADF Faces performs the action through a partial page submit. Thus you can use a command button or link to update a portion of a page, without having to redraw the entire page upon a submit. By default the value of `partialSubmit` is `false`, which means full page rendering is used in response to a partial event. Full page rendering is also automatically used when partial page rendering is not supported in the client browser or platform or when navigating to another page.

In the example, the `partialTriggers` attributes on the **Assigned to** display-only `outputText` components are set to the id of the `commandLink` component. When the `commandLink` component fires a partial event, the output components (which are listening for partial events from `commandLink`) know to refresh their values via partial page rendering.

11.4.3 What You May Need to Know About PPR and Screen Readers

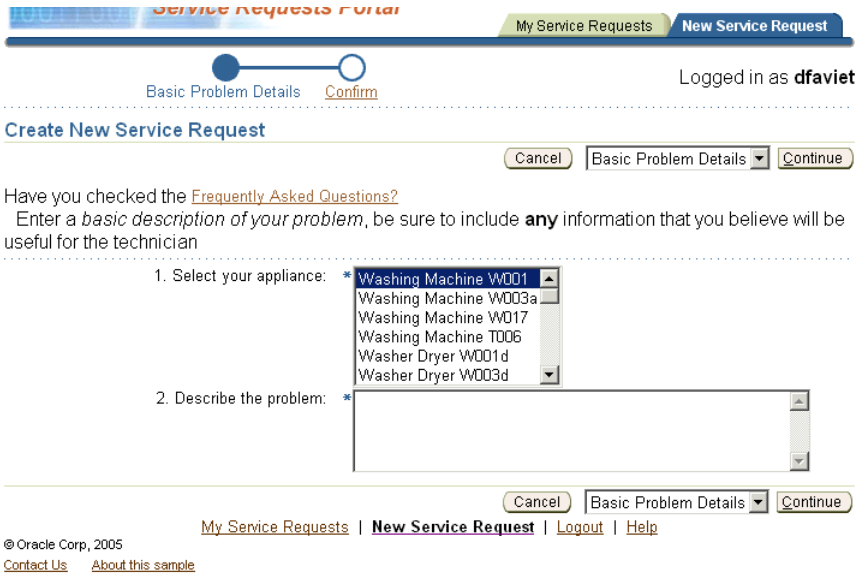
Screen readers do not reread the full page in a partial page request. PPR causes the screen reader to read the page starting from the component that fired the partial action. Hence, you should place the target components after the component that fires the partial request; otherwise the screen reader would not read the updated targets.

11.5 Creating a Multipage Process

If you have a set of pages that should be visited in a particular order, consider using the `processTrain` and `processChoiceBar` components to show the multipage process. In the SRDemo application, the `SRCreate.jspx` and `SRCreateConfirm.jspx` pages use a `processTrain` and `processChoiceBar` component to let a user create a new service request.

When rendered, the `processTrain` component shows the total number of pages in the process as well as the page where the user is currently at, and allows the user to navigate between those pages. For example, Figure 11–13 shows the first page in the create service request process, where the user selects one appliance from a listbox and enters a description of the problem in a textbox. The number of nodes (circles) in the train indicates the total number of predefined pages in the process; the solid node indicates that the user is currently working on that page in the process. To go to the next page in the process, the user clicks the active text link below the node.

Figure 11–13 First Page of the Create New Service Request Process (SRCreate.jspx)

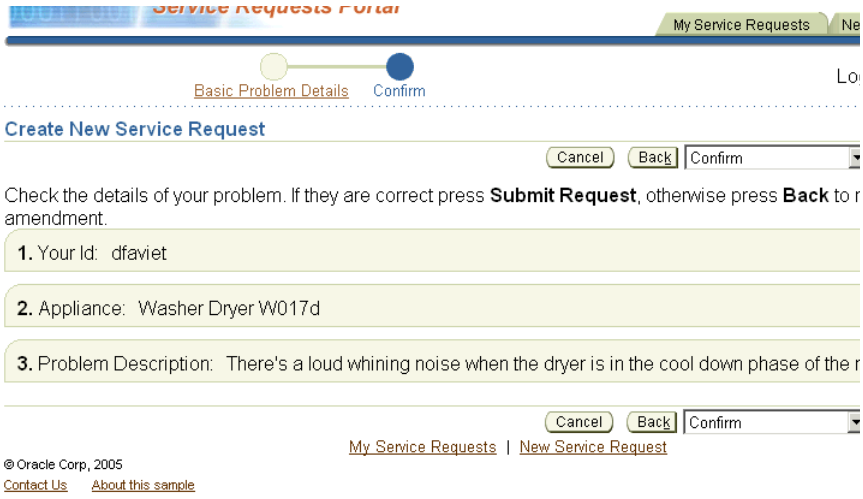


Note that the illustrations in this chapter use the Oracle skin and not the SRDemo skin.

The `processChoiceBar` component renders a dropdown menu for selecting a page in the process, and where applicable, one or more buttons for navigating forward and backward in the process.

On the first page in the create service request process, when the user clicks the **Confirm** text link or the **Continue** button, or selects **Confirm** from the dropdown menu, the application displays the second page of the process, as shown in Figure 11–14.

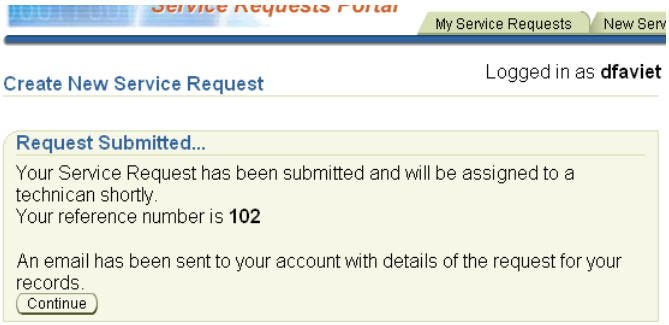
Figure 11–14 Second Page of the Create New Service Request Process (SRCreateConfirm.jspx)



From the second page, the user can return to the problem description page by clicking **Basic Problem Details** in the train or clicking the **Back** button, or by selecting **Basic Problem Details** from the dropdown menu.

If done the user clicks **Submit Request**, and the application displays the Request Submitted page, as shown in Figure 11–15.

Figure 11–15 Request Submitted Page (SRCreateDone.jspx)



11.5.1 How to Create a Process Train

To display a process train on each page, you bind the `processTrain` component to a process train model. At runtime the train model dynamically creates the train for each page in the process.

To create and use a process train:

1. Create a process train model. (See Section 11.5.1.1, "Creating a Process Train Model")
2. Create the JSF page for each node in the train. (See Section 11.5.1.2, "Creating the JSF Page for Each Train Node")
3. Create a navigation rule that has navigation cases for each node. (See Section 11.5.1.3, "Creating the JSF Navigation Rules")

11.5.1.1 Creating a Process Train Model

Use the `oracle.adf.view.faces.model.MenuModel` class and the `oracle.adf.view.faces.model.ProcessMenuModel` class to create a process train model that dynamically generates a process train. The `MenuModel` class is the same menu model mechanism that is used for creating menu tabs and menu bars, as described in Section 11.2.1, "How to Create Dynamic Navigation Menus".

To create a process train model:

1. Create a class that can get and set the properties for each node in the process train.

   Each node in the train needs to have a `label`, a `viewId` and an `outcome` property. Example 11–37 shows part of the `MenuItem` class used in the SRDemo application.

Example 11–37 MenuItem.java for Process Train Nodes

```
package oracle.srdemo.view.menu;
public class MenuItem {
    private String _label          = null;
    private String _outcome        = null;
    private String _viewId         = null;
    ...
    //extended security attributes
    private boolean _readOnly = false;
    private boolean _shown = true;
    public void setLabel(String label) {
        this._label = label;
    }

    public String getLabel() {
        return _label;
    }

// getter and setter methods for remaining attributes omitted
}
```

2. Configure a managed bean for each node in the train, with values for the properties that require setting at instantiation.

   Each bean should be an instance of the class you create in step 1. Example 11–38 shows the managed bean code for the process train nodes in `faces-config.xml`.

Example 11–38 Managed Beans for Process Train Nodes in the faces-config.xml File

```
<!--First train node -->
<managed-bean>
  <managed-bean-name>createTrain_Step1</managed-bean-name>
  <managed-bean-class>oracle.srdemo.view.menu.MenuItem</managed-bean-class>
  <managed-bean-scope>application</managed-bean-scope>
  <managed-property>
    <property-name>label</property-name>
    <value>#{resources['srcreate.train.step1']}</value>
  </managed-property>
  <managed-property>
    <property-name>viewId</property-name>
    <value>/app/SRCreate.jspx</value>
  </managed-property>
  <managed-property>
    <property-name>outcome</property-name>
    <value>GlobalCreate</value>
  </managed-property>
</managed-bean>
```

```
<!-- Second train node-->
<managed-bean>
  <managed-bean-name>createTrain_Step2</managed-bean-name>
  <managed-bean-class>oracle.srdemo.view.menu.MenuItem</managed-bean-class>
  <managed-bean-scope>application</managed-bean-scope>
  <managed-property>
    <property-name>label</property-name>
    <value>#{resources['srcreate.train.step2']}</value>
  </managed-property>
  <managed-property>
    <property-name>viewId</property-name>
    <value>/app/SRCreateConfirm.jspx</value>
  </managed-property>
  <managed-property>
    <property-name>outcome</property-name>
    <value>Continue</value>
  </managed-property>
</managed-bean>
```

3. Configure a managed bean that is an instance of a list with `application` as its scope.

   The list entries are the train node managed beans you create in step 2, listed in the order that they should appear on the train. Example 11–39 shows the managed bean code for creating the process train list.

Example 11–39 Managed Bean for Process Train List in the faces-config.xml File

```
<!-- create the list to pass to the train model -->
<managed-bean>
  <managed-bean-name>createTrainNodes</managed-bean-name>
  <managed-bean-class>java.util.ArrayList</managed-bean-class>
  <managed-bean-scope>application</managed-bean-scope>
  <list-entries>
    <value-class>oracle.srdemo.view.menu.MenuItem</value-class>
    <value>#{createTrain_Step1}</value>
    <value>#{createTrain_Step2}</value>
  </list-entries>
</managed-bean>
```

4. Create a class to facilitate the construction of a `ProcessMenuModel` instance. This class must have at least two properties, `viewIdProperty` and `instance`.

   Example 11–40 shows the `TrainModelAdapter` class used in the SRDemo application.

Example 11–40 TrainModelAdapter.java for Holding the Process Train Nodes

```
package oracle.srdemo.view.menu;
import oracle.adf.view.faces.model.MenuModel;
import oracle.adf.view.faces.model.ProcessMenuModel;
...
public class TrainModelAdapter implements Serializable {
    private String _propertyName = null;
    private Object _instance = null;
    private transient MenuModel _model = null;
    private Object _maxPathKey = null;
    public MenuModel getModel() throws IntrospectionException {
        if (_model == null)
        {
          _model = new ProcessMenuModel(getInstance(),
                                        getViewIdProperty(),
                                        getMaxPathKey());
        }
        return _model;
    }
    public String getViewIdProperty() {
        return _propertyName;
    }
    /**
     * Sets the property to use to get at view id
     * @param propertyName
     */
    public void setViewIdProperty(String propertyName) {
        _propertyName = propertyName;
        _model = null;
    }
    public Object getInstance() {
        return _instance;
    }
    /**
     * Sets the treeModel
     * @param instance must be something that can be converted into a TreeModel
     */
    public void setInstance(Object instance) {
        _instance = instance;
        _model = null;
    }
    public Object getMaxPathKey()
    {
      return _maxPathKey;
    }
    public void setMaxPathKey(Object maxPathKey)
    {
      _maxPathKey = maxPathKey;
    }
}
```

If you wish to write your own menu model instead of using `ProcessMenuModel`, you can use `ProcessUtils` to implement the PlusOne or MaxVisited behavior for controlling page access. For information about how to control page access using those process behaviors, see Section 11.5.1.1.1, "What You May Need to Know About Controlling Page Access".

5. Configure a managed bean to reference the class you create in step 4. This is the bean to which the `processTrain` component is bound.

   The bean should be instantiated to have the `instance` property value set to the managed bean that creates the train list (as configured in step 3). The instantiated bean should also have the `viewIdProperty` value set to the `viewId` property on the bean created in step 1. Example 11–41 shows the managed bean code for creating the process train model.

Example 11–41 Managed Bean for Process Train Model in the faces-config.xml File

```
<!-- create the train menu model -->
<managed-bean>
  <managed-bean-name>createTrainMenuModel</managed-bean-name>
  <managed-bean-class>
    oracle.srdemo.view.menu.TrainModelAdapter</managed-bean-class>
  <managed-bean-scope>application</managed-bean-scope>
  <managed-property>
    <property-name>viewIdProperty</property-name>
    <value>viewId</value>
  </managed-property>
  <managed-property>
    <property-name>instance</property-name>
    <value>#{createTrainNodes}</value>
  </managed-property>
</managed-bean>
```

11.5.1.1.1 What You May Need to Know About Controlling Page Access

When you want to control the pages users can access based on the page they are currently on, you can use one of two process scenarios provided by ADF Faces, namely Max Visited or Plus One.

Suppose there are five pages or nodes in a process train, and the user has navigated from page 1 to page 4 sequentially. At page 4 the user jumps back to page 2. Where the user can go next depends on which process scenario is used.

In the Max Visited process, from the current page 2 the user can go back to page 1, go ahead to page 3, or jump ahead to page 4. That is, the Max Visited process allows the user to return to a previous page or advance to any page up to the furthest page already visited. The user cannot jump ahead to page 5 from page 2 because page 5 has not yet been visited.

Given the same situation, in the Plus One process the user can only go ahead to page 3 or go back to page 1. That is, the Plus One process allows the user to return to a previous page or to advance one node in the train further than they are on currently. The user cannot jump ahead to page 4 even though page 4 has already been visited.

If you were to use the Max Visited process, you would add code similar to the next code snippet, for the `createTrainMenuModel` managed bean (see Example 11–41) in `faces-config.xml`:

```
<managed-property>
  <property-name>maxPathKey</property-name>
  <value>TRAIN_DEMO_MAX_PATH_KEY</value>
</managed-property>
```

ADF Faces knows to use the Max Visited process because a `maxPathKey` value is passed into the `ProcessMenuModel` (see Example 11–40).

The Create New Service Request process uses the Plus One process because `faces-config.xml` doesn't have the `maxPathKey` managed-property setting, thus `null` is passed for `maxPathKey`. When `null` is passed, ADF Faces knows to use the PlusOne process.

The process scenarios also affect the `immediate` and `readOnly` attributes of the command component used within a `processTrain` component. For information, see Section 11.5.1.2.1, "What You May Need to Know About the Immediate and ReadOnly Attributes".

11.5.1.2 Creating the JSF Page for Each Train Node

Each train node has its own page. To display the process train, on each page bind the `processTrain` component to the process train model, as shown in Example 11–42.

A `processTrain` component is usually inserted in the `location` facet of a `panelPage` or `page` component. Like a menu component, a `processTrain` component has a `nodeStamp` facet that accepts one `commandMenuItem` component. It is the `commandMenuItem` component that provides the actual label you see below a train node, and the navigation outcome when the label is activated.

Example 11–42 ProcessTrain Component in the SRCreate.jspx File

```
<af:panelPage..>
  ...
  <f:facet name="location">
    <af:processTrain var="train"
                     value="#{createTrainMenuModel.model}">
      <f:facet name="nodeStamp">
        <af:commandMenuItem text="#{train.label}"
                            action="#{train.getOutcome}"
                            readOnly="#{createTrainMenuModel.model.readOnly}"
                            immediate="false"/>
      </f:facet>
    </af:processTrain>
  </f:facet>
  ...
</af:panelPage>
```

> **Note:** You can use the same code for the process train on each page because the process train model dynamically determines the train node links, the order of the nodes, and whether the nodes are enabled, disabled, or selected.

Typically, you use a `processTrain` component with a `processChoiceBar` component. The `processChoiceBar` component, which is also bound to the same process train model, gives the user additional navigation choices for stepping through the multipage process. Example 11–43 shows the code for the `processChoiceBar` component in the `SRCreate.jspx` page. A `processChoiceBar` component is usually inserted in the `actions` facet of a `panelPage` or `page` component.

Example 11–43 ProcessChoiceBar Component in the SRCreate.jspx File

```
<af:panelPage ..>
  <f:facet name="actions">
    <af:panelButtonBar>
      <af:commandButton text="#{res['srdemo.cancel']}"
                        action="#{backing_SRCreate.cancelButton_action}"
                        immediate="true"/>
      <af:processChoiceBar var="choice"
                           value="#{createTrainMenuModel.model}">
        <f:facet name="nodeStamp">
          <af:commandMenuItem text="#{choice.label}"
                              action="#{choice.getOutcome}"
                              readOnly="#{createTrainMenuModel.model.readOnly}"
                              immediate="false"/>
        </f:facet>
      </af:processChoiceBar>
    </af:panelButtonBar>
  </f:facet>
  ...
</af:panelPage>
```

As illustrated in Figure 11–13 and Figure 11–14, the `processChoiceBar` component automatically provides a **Continue** button and a **Back** button for navigating forward and backward in the process. You don't have to write any code for these buttons. If you want to provide additional buttons (such as the **Cancel** and **Submit Request** buttons in Figure 11–14), use a `panelButtonBar` to lay out the button components and the `processChoiceBar` component.

Note: If your multipage process has only two pages, ADF Faces uses **Continue** as the label for the button that navigates forward. If there is more than two pages in the process, the forward button label is **Next**.

11.5.1.2.1 What You May Need to Know About the Immediate and ReadOnly Attributes

The two process scenarios provided by ADF Faces and described in Section 11.5.1.1.1, "What You May Need to Know About Controlling Page Access" have an effect on both the `immediate` and `readOnly` attributes of the `commandMenuItem` component used within `processTrain`. When binding `processTrain` to a process train model, you can bind the node's `immediate` or `readOnly` attribute to the model's `immediate` or `readOnly` attribute. The `ProcessMenuModel` class then uses logic to determine the value of the `immediate` or `readOnly` attribute.

When the data on the current page does not need to be validated, the `immediate` attribute should be set to `true`. For example, in the Plus One scenario described in Section 11.5.1.1.1, if the user is on page 4 and goes back to page 2, the user has to come back to page 4 again later, so that data does not need to be validated when going to page 1 or 3, but should be validated when going ahead to page 5.

The `ProcessMenuModel` class uses the following logic to determine the value of the `immediate` attribute:

- Plus One: `immediate` is set to `true` for any previous step, and `false` otherwise.
- Max Visited: When the current page and the maximum page visited are the same, the behavior is the same as the Plus One scenario. If the current page is before the maximum page visited, then `immediate` is set to `false`.

The `readOnly` attribute should be set to `true` only if that page of the process cannot be reached from the current page. The `ProcessMenuModel` class uses the following logic to determine the value of the `readOnly` attribute:

- Plus One: `readOnly` will be `true` for any page past the next available page.
- Max Visited: When the current step and the maximum page visited are the same, the behavior is the same as the Plus One scenario. If the current page is before the maximum page visited, then `readOnly` is set to `true` for any page past the maximum page visited.

11.5.1.3 Creating the JSF Navigation Rules

The `<from-outcome>` and `<to-view-id>` values in the navigation cases must match the properties set in the process train model.

In the SRDemo application, a global navigation rule is used for the first page of the Create New Service Request process because the `SRCreate.jspx` page is accessible from any page in the application. The second page of the process, `SRCreateConfirm.jspx`, is not included in the global navigation rule because it is only accessible from the `SRCreate.jspx` page. Example 11–44 shows the navigation rules and cases for the process.

Example 11–44 Navigation Rules for Process Train Nodes in the faces.config.xml File

```
<navigation-rule>
  <from-view-id>*</from-view-id>
  <navigation-case>
    <from-outcome>GlobalCreate</from-outcome>
    <to-view-id>/app/SRCreate.jspx</to-view-id>
  </navigation-case>
  ...
</navigation-rule>
<navigation-rule>
  <from-view-id>/app/SRCreate.jspx</from-view-id>
  <navigation-case>
    <from-outcome>Continue</from-outcome>
    <to-view-id>/app/SRCreateConfirm.jspx</to-view-id>
  </navigation-case>
  ...
</navigation-rule>
<navigation-rule>
  <from-view-id>/app/SRCreateConfirm.jspx</from-view-id>
  <navigation-case>
    <from-outcome>Back</from-outcome>
    <to-view-id>/app/SRCreate.jspx</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>Complete</from-outcome>
    <to-view-id>/app/SRCreateDone.jspx</to-view-id>
  </navigation-case>
</navigation-rule>
```

11.5.2 What Happens at Runtime

Java automatically adds a no-arg constructor to `TrainModelAdapter` because the `TrainModelAdapter` class is used as a managed bean. `TrainModelAdapter` constructs the process train model, which is a `ProcessMenuModel` instance, via the `createTrainMenuModel` managed bean. The `createTrainNodes` managed bean creates and injects the train node list into the train model. The train model provides the model that correctly highlights and enables the nodes on the train as you step through the process.

The individual train node managed beans (for example, `createTrain_Step1`) are instantiated with values for `label`, `viewId`, and `outcome` that are used by the train model to dynamically generate the train nodes. The default JSF `actionListener` mechanism uses the outcome values to handle the page navigation.

In the SRDemo application, the individual train node managed beans access `String` resources in the resource bundle via the `resources` managed bean, so that the correct node label is dynamically retrieved and display at runtime.

At runtime if `maxPathKey` has a value (set in `faces-config.xml`), ADF Faces knows to use the Max Visited process scenario. If `maxPathKey` is `null` (as in the SRDemo application), ADF Faces uses the Plus One process to control page access from the current page.

Like the `menuTab` component, the `processTrain` and `processChoiceBar` components have a `nodeStamp` facet, which takes one `commandMenuItem` component. By using `train` as the variable and binding the `processTrain` component to the process train model, you need only one `commandMenuItem` component to display all train node items using `#{train.label}` as the `text` value and `#{train.getOutcome}` as the `action` value on the command component. Similarly, by using `choice` as the variable and binding the `processChoiceBar` component to the process train model, you need only one `commandMenuItem` component to display all items as menu options using `#{choice.label}` as the `text` value and `#{choice.getOutcome}` as the `action` value.

The enabling and disabling of a node is not controlled by the `MenuItem` class, but by the process train model based on the current view using the EL expression `#{createTrainMenuModel.model.readOnly}` on the `readOnly` attribute of the `processTrain` or `processChoiceBar` component.

> **Tip:** Disabled menu choices are not rendered on browsers that don't support disabled items in a dropdown menu. On browsers that support disabled items in a dropdown menu, the unreachable items will look disabled.

11.5.3 What You May Need to Know About Process Trains and Menus

The `ProcessMenuModel` class extends the `ViewIdPropertyMenuModel` class, which is used to create dynamic menus, as described in Section 11.2, "Using Dynamic Menus for Navigation". Like menus and menu items, each node on a train is defined as a menu item. But unlike menus where the menu items are gathered into the intermediate menu tree object (`MenuTreeModelAdapter`), the complete list of train nodes is gathered into an `ArrayList` that is then injected into the `TrainModelAdapter` class. Note, however, that both `ViewIdPropertyMenuModel` and `ProcessMenuModel` can always take a List and turn it into a tree internally.

In the SRDemo application, the nodes on the train are not secured by user role as any user can create a new service request, which means that the train model can be stored as an `application` scoped managed bean and shared by all users.

The menu model is stored as a `session` scoped managed bean because the menu tab items are secured by user role, as some tabs are not available to some user roles.

To add a new page to a process train, configure a new managed bean for the page (Example 11–38), add the new managed bean to the train list (Example 11–39), and add the navigation case for the new page (Example 11–44).

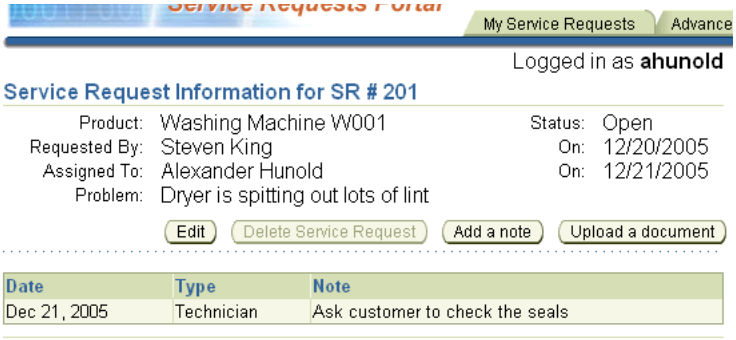
11.6 Providing File Upload Capability

File uploading is a capability that is required in many web applications. Standard J2EE technologies such as Servlets and JSP, and JSF 1.1.x, do not directly support file uploading. The ADF Faces framework, however, has integrated file uploading support at the component level via the `inputFile` component.

During file uploading, ADF Faces temporarily stores incoming files either in memory or on disk. You can set a default directory storage location, and default values for the amount of disk space and memory that can be used in any one file upload request.

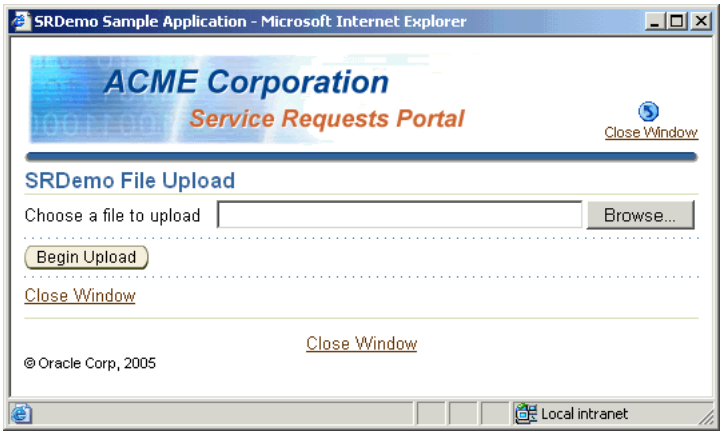
Figure 11–16 shows the `SRMain.jspx` page of the SRDemo application, where users can upload files for a particular service request.

Figure 11–16 File Upload Button on the SRMain Page



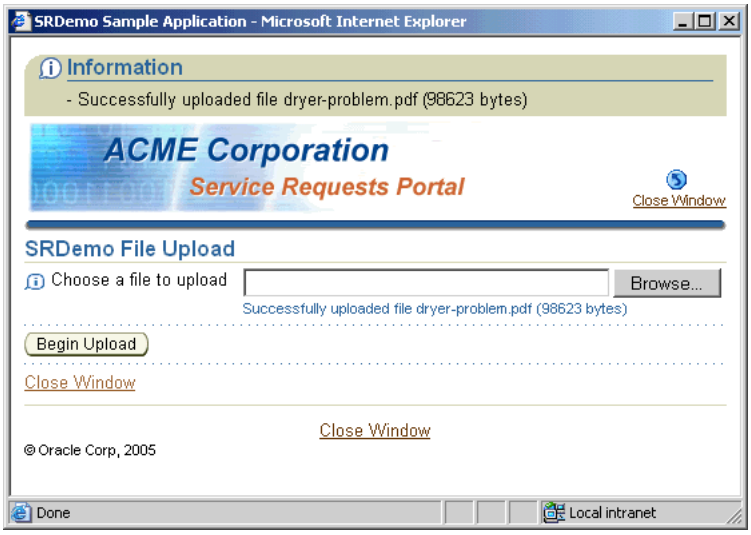
When the user clicks **Upload document**, the upload form displays in a popup dialog, as shown in Figure 11–17.

Figure 11–17 File Upload Form in the SRDemo Application



The user can enter the full pathname of the file for uploading or click **Browse** to locate and select the file. When **Begin Upload** is clicked, ADF Faces automatically uploads the selected file. Upon a successful upload, ADF Faces displays some information about the uploaded file, as shown in Figure 11–18. If uploading is unsuccessful for some reason, the application displays the error stack trace in the same popup dialog.

Figure 11–18 File Upload Success Information



11.6.1 How to Support File Uploading on a Page

Use the following tasks to provide file uploading support in a JSF application.

To provide file uploading support:

1. Make sure the ADF Faces filter has been installed.

   The ADF Faces filter is a servlet filter that ensures ADF Faces is properly initialized by establishing an `AdfFacesContext` object. JDeveloper automatically installs the filter for you in `web.xml` when you insert an ADF Faces component into a JSF page for the first time. Example 11–45 shows the ADF Faces filter and mapping configuration setting in `web.xml`.

Example 11–45 ADF Faces Filter in the web.xml File

```
<!-- Installs the ADF Faces Filter -- >
<filter>
  <filter-name>adfFaces</filter-name>
  <filter-class>oracle.adf.view.faces.webapp.AdfFacesFilter</filter-class>
</filter>

<!-- Adds the mapping to ADF Faces Filter -- >
<filter-mapping>
  <filter-name>adfFaces</filter-name>
  <servlet-name>Faces Servlet</servlet-name>
</filter-mapping>
```

2. In `web.xml` set a context initialization parameter for the storage location of uploaded files. It's up to you where you want to save the uploaded files. Example 11–46 shows the context parameter used in the SRDemo application for uploaded files.

Example 11–46 Uploaded File Storage Location in the web.xml File

```
<context-param>
  <description>Parent directory location of SRDemo fileuploads</description>
  <param-name>SRDemo.FILE_UPLOADS_DIR</param-name>
  <param-value>/tmp/srdemo_fileuploads</param-value>
</context-param>
```

3. Create a backing bean for handling uploaded files. Example 11–47 shows the managed bean code in `faces-config.xml` for the SRDemo file upload page.

Example 11–47 Managed Bean for the SRFileUpload Page in the faces.config.xml File

```
<managed-bean>
  <managed-bean-name>backing_SRFileUpload</managed-bean-name>
  <managed-bean-class>
    oracle.srdemo.view.backing.SRFileUpload</managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
  ...
</managed-bean>
```

4. In the JSF page you can use either `af:form` or `h:form` for file uploading. Make sure you set the enclosing form to support file uploads, as shown in the next code snippet:

   ```
   <af:form usesUpload="true"/>
   ..
   <h:form enctype="multipart/form-data"/>
   ```

5. Use the `inputFile` component to provide a standard input field with a label, and a **Browse** button, as shown in Figure 11–17.

   The `inputFile` component delivers standard value change events as files are uploaded, and manages the processing of the uploaded contents for you. It is up to you how you want to handle the contents.

   To process file uploading, you could either implement a value change listener method in the backing bean to handle the event, or bind the `value` attribute of `inputFile` directly to a managed bean property of type `oracle.adf.view.faces.model.UploadedFile`. Either way you have to write your own Java code in the backing bean for handling the uploaded files.

   The following code snippet shows the code for an `inputFile` component if you were to bind the component to a managed bean property of type `oracle.adf.view.faces.model.UploadedFile`.

   ```
   <af:inputFile value="#{myuploadBean.myuploadedFile}".../>
   ```

   The SRDemo file upload form uses a value change listener method. Example 11–48 shows the code for the method binding expression in the `valueChangeListener` attribute of the `inputFile` component.

Example 11–48 InputFile Component in the SRFileUpload.jspx File

```
<af:inputFile label="#{res['srfileupload.uploadlabel']}"
              valueChangeListener="#{backing_SRFileUpload.fileUploaded}"
              binding="#{backing_SRFileUpload.srInputFile}"
              columns="40"/>
```

6. In the page's backing bean, write the code for handling the uploaded contents. For example, you could write the contents to a local directory in the file system. Example 11–49 shows the value change listener method that handles the value change event for file uploading in the SRDemo application.

Example 11–49 Value Change Listener Method for Handling a File Upload Event

```
public void fileUploaded(ValueChangeEvent event) {

    InputStream in;
    FileOutputStream out;

    // Set fileUPloadLoc to "SRDemo.FILE_UPLOADS_DIR" context init parameter
    String fileUploadLoc =
FacesContext.getCurrentInstance().getExternalContext().getInitParameter("SRDemo.FI
LE_UPLOADS_DIR");

    if (fileUploadLoc == null) {
       // Backup value if context init parameter not set.
      fileUploadLoc = "/tmp/srdemo_fileuploads";
    }


   //get svrId and append to file upload location
   Integer svrId =
(Integer)JSFUtils.getManagedBeanValue("userState.currentSvrId");
   fileUploadLoc += "/sr_" + svrId + "_uploadedfiles";


    // Create upload directory if it does not exists.
    boolean exists = (new File(fileUploadLoc)).exists();
    if (!exists) {
        (new File(fileUploadLoc)).mkdirs();
    }


    UploadedFile file = (UploadedFile)event.getNewValue();


    if (file != null && file.getLength()>0) {
        FacesContext context = FacesContext.getCurrentInstance();
        FacesMessage message =
                    new
FacesMessage(JSFUtils.getStringFromBundle("srmain.srfileupload.success")+" "+
                                    file.getFilename() + " (" +
                                    file.getLength() +
                                    " bytes)");
        context.addMessage(event.getComponent().getClientId(context),
                           message);
```

```
        try {
            out =
                new FileOutputStream(fileUploadLoc + "/" + file.getFilename());
            in = file.getInputStream();

            for (int bytes = 0; bytes < file.getLength(); bytes++) {
              out.write(in.read());
            }

            in.close();
            out.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    else {
        // need to check for null value here as otherwise closing
        // the dialog after a failed upload attempt will lead to
        // a nullpointer exception
        String filename = file != null ? file.getFilename() : null;
        String byteLength = file !=null ? "" + file.getLength() : "0";

        FacesContext context = FacesContext.getCurrentInstance();
        FacesMessage message =
        new FacesMessage(FacesMessage.SEVERITY_WARN,
JSFUtils.getStringFromBundle("srmain.srfileupload.error") + " " +
                        filename + " (" + byteLength + " bytes)", null);
        context.addMessage(event.getComponent().getClientId(context),message);
    }
}
```

7. Use a `commandButton` component to submit the form. Example 11–50 shows the `commandButton` code in the SRDemo file upload form, and also the action method code in the page's backing bean.

Example 11–50 Code for the Command Button and Action Method

```
<af:commandButton text="#{res['srfileupload.uploadbutton']}"
                  action="#{backing_SRFileUpload.UploadButton_action}"/>

...
...
public String UploadButton_action() {
    if (this.getSrInputFile().getValue() == null){
        FacesContext context = FacesContext.getCurrentInstance();
        FacesMessage message =
        new FacesMessage(FacesMessage.SEVERITY_WARN,
JSFUtils.getStringFromBundle("srmain.srfileupload.emptyfielderror"), null);
        context.addMessage(this.getSrInputFile().getId(), message);

    }

    return null;
}
```

8. If using a popup dialog, add a `commandLink` component to let the user close the dialog. For more information about closing a popup dialog, see Section 11.3.1.3, "Creating the Dialog Page and Returning a Dialog Value". Example 11–51 shows the code for the `commandLink` component and the action method in the page's backing bean.

Example 11–51 Code for the Command Link and Action Method

```
<af:commandLink action="#{backing_SRFileUpload.closeFileUpload_action}"../>
..
public String closeFileUpload_action() {
  AdfFacesContext.getCurrentInstance().returnFromDialog(null, null);
  return null;
}
```

11.6.2 What Happens at Runtime

The SRDemo application creates a directory such as `C:\tmp\srdemo_fileuploads` to store uploaded files. Uploaded files for a service request are placed in a subdirectory prefixed with the service request id, for example `C:\tmp\srdemo_fileuploads\sr_103_uploadedfiles`.

The `oracle.adf.view.faces.webapp.UploadedFileProcessor` API is responsible for processing file uploads. Each application has a single `UploadedFileProcessor` instance, which is accessible from `AdfFacesContext`.

The `UploadedFileProcessor` processes each uploaded file as it comes from the incoming request, converting the incoming stream into an `oracle.adf.view.faces.model.UploadedFile` instance, and making the contents available for the duration of the current request. In other words, the `value` attribute of the `inputFile` component is automatically set to an instance of `UploadedFile`. If the `inputFile` component's value is bound to a managed bean property of type `oracle.adf.view.faces.model.UploadedFile`, ADF Faces sets an `UploadedFile` object on the model.

The `oracle.adf.view.faces.model.UploadedFile` API describes the contents of a single file. It lets you get at the actual byte stream of the file, as well as the file's name, its MIME type, and its size. The `UploadedFile` might be stored as a file in the file system, or it might be stored in memory; the API hides that difference.

ADF Faces limits the size of acceptable incoming requests to avoid denial-of-service attacks that might attempt to fill a hard drive or flood memory with uploaded files. By default, only the first 100 kilobytes in any one request are stored in memory. Once that has been filled, disk space is used. Again, by default, that is limited to 2,000 kilobytes of disk storage for any one request for all files combined. The `AdfFacesFilter` throws an `EOFException` once the default disk storage and memory limits are reached. To change the default values, see Section 11.6.4, "Configuring File Uploading Initialization Parameters".

11.6.3 What You May Need to Know About ADF Faces File Upload

Consider the following if you're using ADF Faces file upload:

- Most applications don't need to replace the default `UploadedFileProcessor` instance, but if your application needs to support uploading of very large files, you may wish to replace the default processor with a custom `UploadedFileProcessor` implementation. For more information see Section 11.6.5, "Configuring a Custom Uploaded File Processor".
- The ADF Faces Filter ensures that the `UploadedFile` content is cleaned up after the request is complete. Thus, you cannot cache `UploadedFile` objects across requests. If you need to keep a file, you must copy it into persistent storage before the request finishes.

11.6.4 Configuring File Uploading Initialization Parameters

During file uploading, ADF Faces temporarily stores incoming files either on disk or in memory. ADF Faces defaults to the application server's temporary directory, as provided by the `javax.servlet.context.tempdir` property. If that property is not set, the system `java.io.tempdir` property is used.

If you wish you can set a default temporary storage location, and default values for the amount of disk space and memory that can be used in any one file upload request. You can specify the following file upload context parameters in `web.xml`:

- `oracle.adf.view.faces.UPLOAD_TEMP_DIR`—Specifies the directory where temporary files are to be stored during file uploading. Default is the user's temporary directory.
- `oracle.adf.view.faces.UPLOAD_MAX_DISK_SPACE`—Specifies the maximum amount of disk space that can be used in a single request to store uploaded files. Default is 2000K.
- `oracle.adf.view.faces.UPLOAD_MAX_MEMORY`—Specifies the maximum amount of memory that can be used in a single request to store uploaded files. Default is 100K.

Example 11–52 shows the context initialization parameters for file uploading that you use in `web.xml`.

Example 11–52 Context Parameters for File Uploading in the web.xml File

```
<context-param>
  <param-name>oracle.adf.view.faces.UPLOAD_TEMP_DIR</param-name>
  <param-value>/tmp/Adfuploads</param-value>
</context-param>

<context-param>
  <param-name>oracle.adf.view.faces.UPLOAD_MAX_DISK_SPACE</param-name>
  <param-value>10240000</param-value>
</context-param>

<context-param>
  <param-name>oracle.adf.view.faces.UPLOAD_MAX_MEMORY</param-name>
  <param-value>5120000</param-value>
</context-param>
```

Note: The file upload initialization parameters are processed by the default `UploadedFileProcessor` only. If you replace the default processor with a custom `UploadedFileProcessor` implementation, the parameters are not processed.

11.6.5 Configuring a Custom Uploaded File Processor

Most applications don't need to replace the default `UploadedFileProcessor` instance provided by ADF Faces, but if your application needs to support uploading of very large files or rely heavily on file uploads, you may wish to replace the default processor with a custom `UploadedFileProcessor` implementation. For example, you could improve performance by using an implementation that immediately stores files in their final destination, instead of requiring ADF Faces to handle temporary storage during the request.

To replace the default processor, specify the custom implementation using the `<uploaded-file-processor>` element in `adf-faces-config.xml`. Example 11–53 shows the code for registering a custom `UploadedFileProcessor` implementation.

Example 11–53 Registering a Custom Uploaded File Processor in the adf-faces-config.xml File

```
<adf-faces-config xmlns="http://xmlns.oracle.com/adf/view/faces/config">
...
  <!-- Use my UploadFileProcessor class -->
  <uploaded-file-processor>
    com.mycompany.faces.myUploadedFileProcessor
  </uploaded-file-processor>
...
</adf-faces-config>
```

> **Tip:** Any file uploading initialization parameters specified in `web.xml` are processed by the default `UploadedFileProcessor` only. If you replace the default processor with a custom `UploadedFileProcessor` implementation, the file uploading parameters are not processed.

11.7 Creating Databound Dropdown Lists

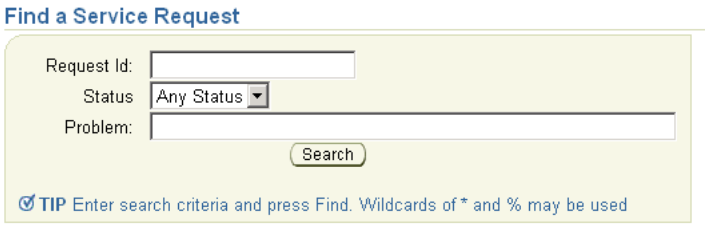
ADF Faces selection list components include `selectOneChoice` and `selectOneListbox`, which work in the same way as standard JSF list components. ADF Faces list components, however, provide extra functionality such as support for label and message display, automatic form submission, and partial page rendering.

In the SRDemo application, the SRSearch page uses a `selectOneChoice` component to let users pick the type of service requests to perform a search on. With the `selectOneChoice` component, you can provide a static list of items for selection, or you can create a list that is populated dynamically. In either case, you use a `f:selectItems` tag to provide the items for display and selection.

11.7.1 How to Create a Dropdown List with a Fixed List of Values

The SRSearch page uses a `selectOneChoice` component to let users pick the type of service request to perform a search on. For example, instead of searching on all service requests, the user can refine the search on requests that have the status of open, pending, or closed. Figure 11–19 shows the search form in the SRDemo application where a `selectOneChoice` component is used.

Figure 11–19 SelectOneChoice Component for Selecting a Service Request Status



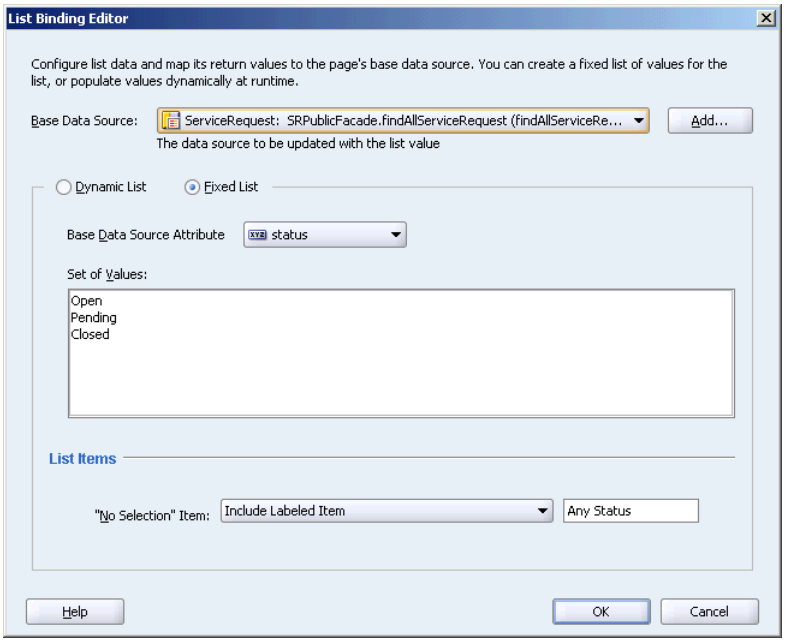
The search form is created using a method that takes parameters. For information about how to create a search form using parameters, see Section 10.8, "Creating Search Pages". The following procedure describes, without using parameters, how to create a dropdown list that is bound to a fixed list of values.

To create a dropdown list bound to a fixed list of values using the Data Control Palette:

1. From the Data Control Palette, expand a business service method, and then expand a method return that returns a data collection. Drag and drop the data collection attribute you desire onto the page, and then choose **Create > Single Selections > ADF Select One Choice** from the context menu. The List Binding Editor displays, as illustrated in Figure 11–20.

   Using the service request status example, you would expand **findAllServiceRequest()**, then expand **ServiceRequest**, and drag and drop the **status** attribute. Because you want users to be able to search on a service request type, therefore you use the **status** attribute on the **ServiceRequest** data collection, which is a collection of all requests returned by the **findAllServiceRequest()** method.

Figure 11–20 List Binding Editor with the Fixed List Option Selected



2. In the List Binding Editor, select **Fixed List**. Then select the **status** attribute from the **Base Data Source Attribute** dropdown list.

   The **Fixed List** option lets users choose a value from a predefined list, which is useful when you want to update a data object attribute with values that you code yourself, rather than getting the values from another data collection.

   When a value is selected from the list, **Base Data Source Attribute** is the attribute of the bound data collection that is to be updated to the selected value.

3. Enter the following in the **Set of Values** box, pressing Enter to set a value before typing the next value:

   - `Open`
   - `Pending`
   - `Closed`

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

4. In the **List Items** section, select **Include Labeled Item** from the **"No Selection" Item** dropdown list. Then enter `Any Status` in the box next to it.

   The `selectOneChoice` component supports a `null` value, that is, 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 of values that is defined in step 3.

   Tip: In the SRDemo application, the `findServiceRequestSearch(Integer, String, String)` method contains the logic to find and return service records based on three parameters, one of which is `statusParam`. Each method parameter has an associated variable. For information about variable iterators and variables, see Section 10.8.2, "What Happens When You Use Parameter Methods".

   If you created the search form using the method with parameters (as described in Section 10.8.1, "How to Create a Search Form"), delete the `inputText` component created for the **Status** field, and replace it with a `selectOneChoice` component by dragging and dropping **statusParam** from the Data Control Palette. In the List Binding Editor, for the **Base Data Source Attribute**, select the variable name **findServiceRequestSearch_statusParam**.

11.7.2 What Happens When You Create a Dropdown List Bound to a Fixed List

When you drag and drop from the Data Control Palette, JDeveloper does many things for you. For a full description of what happens and what is created when you use the Data Control Palette, see Section 5.2.3, "What Happens When You Use the Data Control Palette".

Example 11–54 shows the code for the `selectOneChoice` component after you've completed the List Binding Editor.

Example 11–54 SelectOneChoice Component After You Complete Binding

```
<af:selectOneChoice value="#{bindings.ServiceRequeststatus.inputValue}"
                    label="#{bindings.ServiceRequeststatus.label}: ">
  <f:selectItems value="#{bindings.ServiceRequeststatus.items}"/>
</af:selectOneChoice>
```

The `f:selectItems` tag, which provides the list of items for selection, is bound to the `items` property on the `ServiceRequeststatus` list binding object in the binding container.

In the page definition file (for example, `SRSearchPageDef.xml`), JDeveloper adds the list binding object definition in the `bindings` element, as shown in Example 11–55.

Example 11–55 List Binding Object for the Fixed Dropdown List in the Page Definition File

```
<bindings>
  ...
  <list id="ServiceRequeststatus" IterBinding="findAllServiceRequestIter"
        ListOperMode="0" StaticList="true" NullValueFlag="1">
    <AttrNames>
      <Item Value="status"/>
    </AttrNames>
    <ValueList>
      <Item Value="Any Status"/>
      <Item Value="Open"/>
      <Item Value="Pending"/>
      <Item Value="Closed"/>
    </ValueList>
  </list>
  ...
</bindings>
```

The `id` attribute specifies the name of the list binding object. The `IterBinding` attribute specifies the iterator binding object, which exposes and iterates over the collection returned by the **findAllServiceRequest()** method. The `AttrNames` element defines the attribute returned by the iterator. The `ValueList` element specifies the fixed list of values to be displayed for selection at runtime.

For more information about the page definition file and ADF data binding expressions, see Section 5.5, "Working with Page Definition Files" and Section 5.6, "Creating ADF Data Binding EL Expressions".

11.7.3 How to Create a Dropdown List with a Dynamic List of Values

Instead of getting values from a static list, you can populate a `selectOneChoice` component with values dynamically at runtime. The steps for creating a dropdown list bound to a dynamic list are almost the same as those for creating a dropdown list bound to a fixed list, with the exception that you define two data sources—one for the list data collection that provides the dynamic list of values, and the other for the base data collection that is to be updated based on the user's selection.

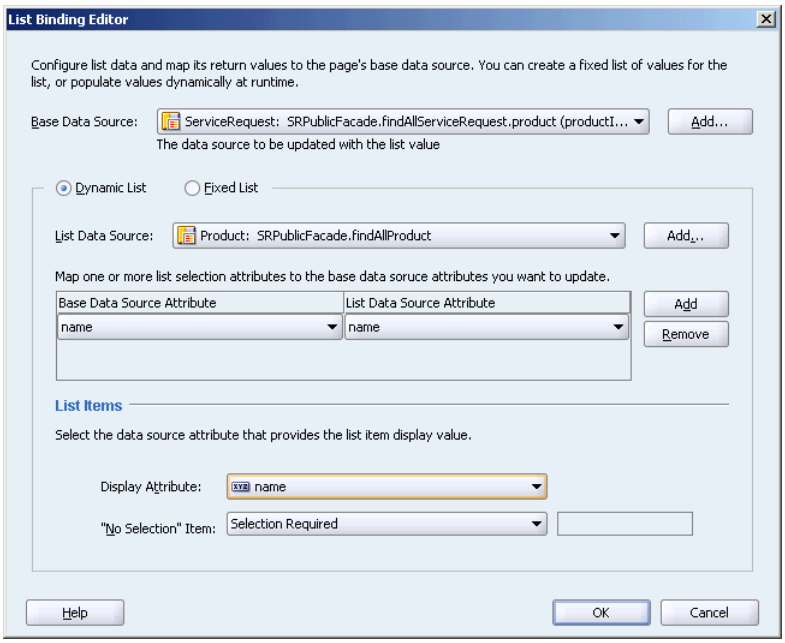
To create a dropdown list bound to a dynamic list of values using the Data Control Palette:

1. From the Data Control Palette, expand a business service method, and then expand a method return that returns a data collection. Next, expand an accessor return that returns a detail collection. Drag and drop the attribute you desire onto the page, and then choose **Create > Single Selections > ADF Select One Choice** from the context menu. The List Binding Editor displays, as illustrated in Figure 11–21.

   For example, if users want to be able to pick a product before searching on service requests, you might expand **findAllServiceRequest()**, followed by **ServiceRequest**, and **product**. Then drag and drop the **name** attribute. Because you want users to be able to search service requests based on a product name, therefore you use the **name** attribute on the **Product** detail collection.

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

Figure 11–21 List Binding Editor with the Dynamic List Option Selected



2. In the List Binding Editor, select **Dynamic List**.
3. In the **Base Data Source** dropdown list, select the data collection that is to be updated with the list value selected by a user. For example, **ServiceRequest: SRPublicFacade.findAllServiceRequest.product**.
4. In the **List Data Source** dropdown list, select the data collection that provides the list of values dynamically. For example, **Product: SRPublicFacade findAllProduct**.
5. In the mapping area, select **name** from **Base Data Source Attribut**e, and **name** from **List Data Source Attribute**. This maps the list source attribute to the base source attribute you want to update.
6. In the **List Items** section, select **name** from the **Display Attribute** dropdown list. This populates the values users see in the list.

11.7.4 What Happens When You Create a Dropdown List Bound to a Dynamic List

When you drag and drop from the Data Control Palette, JDeveloper does many things for you. For a full description of what happens and what is created when you use the Data Control Palette, see Section 5.2.3, "What Happens When You Use the Data Control Palette".

Example 11–56 shows the code for the `selectOneChoice` component after you've completed the List Binding Editor.

Example 11–56 SelectOneChoice Component After You Complete Binding

```
<af:selectOneChoice value="#{bindings.Productname.inputValue}"
                    label="#{bindings.Productname.label}">
  <f:selectItems value="#{bindings.Productname.items}"/>
</af:selectOneChoice>
```

The `f:selectItems` tag, which provides the list of items for selection, is bound to the `items` property on the `Productname` list binding object in the binding container.

For further descriptions about ADF data binding expressions, see Section 5.6, "Creating ADF Data Binding EL Expressions".

In the page definition file (for example, `SRDemopage.xml`), JDeveloper adds the list binding object definition into the `bindings` element, as shown in Example 11–57.

Example 11–57 List Binding Object for the Dynamic Dropdown List in the Page Definition File

```
<bindings>
  ...
  <list id="Productname" IterBinding="productIterator" StaticList="false"
        ListOperMode="0" ListIter="findAllProductIter"..>
    <AttrNames>
      <Item Value="name"/>
    </AttrNames>
    <ListAttrNames>
      <Item Value="name"/>
    </ListAttrNames>
    <ListDisplayAttrNames>
      <Item Value="name"/>
    </ListDisplayAttrNames>
  </list>
  ...
</bindings>
```

The `id` attribute specifies the name of the list binding object. The `IterBinding` attribute specifies the iterator binding object, which exposes and iterates over the collection returned by the **findAllProduct()** method. The `AttrNames` element defines the base data source attribute returned by the iterator. The `ListAttrNames` element defines the list data source attribute that is mapped to the base data source attribute returned by the iterator. The `ListDisplayAttrNames` element defines the list data source attribute that populates the values users see in the list.

For complete information about page definition files, see Section 5.5, "Working with Page Definition Files".

11.7.5 How to Use Variables with Dropdown Lists

Sometimes you might want to use a variable with a `selectOneChoice` component to hold the value of the item selected by a user. On the SRSkills page (as shown later in Figure 11–25), a manager selects a staff member name from the dropdown list to display the member's assigned product skills in the shuttle component. The `selectOneChoice` component in the SRSkills page populates a variable in the page definition file when the user makes a selection.

The following procedure shows how to manually add a variable to a page definition file.

To create a variable iterator and variable in a page definition file:

1. Open the page definition file (for example, `<pageName>PageDef.xml`) for the JSF page in which a `selectOneChoice` component will be used.
2. In the Structure window, right-click the topmost node and choose **Insert inside <pageName>PageDef > executables** to add the **executables** node, if not added already.
3. In the Structure window, right-click **executables** and choose **Insert inside executables > variableIterator** to add the **variables** node, if not added already.

4. In the Structure window, right-click **variables** and choose **Insert inside variables > variable**.
5. In the Insert Variable dialog, enter a name and type for the variable. For example, you might enter `someStaffIdVar` for the name, and `java.lang.Integer` for the type, if you want the variable to hold data about the selected staff member.

Example 11–58 shows the page definition file after you've created a variable.

Example 11–58 Variable Iterator and Variable in the Page Definition File

```
<?xml version="1.0" encoding="UTF-8" ?>
<pageDefinition xmlns="http://xmlns.oracle.com/adfm/uimodel"
                version="10.1.3.36.61" id="app_management_untitled4PageDef"
                Package="oracle.srdemo.view.pageDefs">
  <executables>
    <variableIterator id="variables">
      <variable Name="someStaffIdVar" Type="java.lang.Integer"/>
    </variableIterator>
  </executables>
</pageDefinition>
```

For more information about variable iterators and variables, see Section 5.5.2.2, "Binding Objects Defined in the executables Element".

The next procedure shows how to use the variable to create a dropdown list that lets users select a staff member's name from a dynamic list.

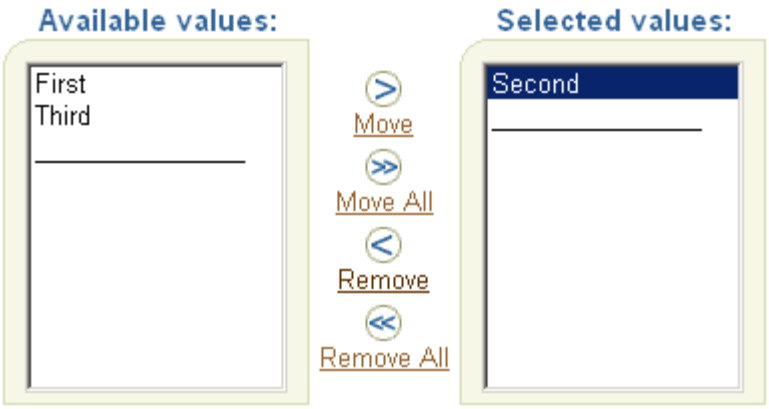
To create a dynamic dropdown list:

1. Open the JSF page in which you want to add a `selectOneChoice` component.
2. From the Data Control Palette, expand **findAllStaff() > User**. Drag and drop the **userId** attribute to the page, and then choose **Create > Single Selections > ADF Select One Choice** from the context menu.
3. In the List Binding Editor, select **variables** from the **Base Data Source** dropdown list.
4. Select **Dynamic List**.
5. From the **List Data Source** dropdown list, select **User: SRPublicFacade:findAllStaff**.
6. In the mapping area, select **someStaffIdVar** from the **Base Data Source Attribute** dropdown list, and **userId** from the **List Data Source Attribute** dropdown list.
7. In the **List Items** section, from the **Display Attribute** dropdown list, select **Select Multiple**, and add **firstName** and **lastName** to the **Attributes to Display** list in the Select Multiple Display Attributes dialog.
8. From the **"No Selection" Item** dropdown list, select **Include Labeled Item**.

11.8 Creating a Databound Shuttle

The `selectManyShuttle` and `selectOrderShuttle` components render two list boxes, and buttons that allow the user to select multiple items from the leading (or "available") list box and move or shuttle the items over to the trailing (or "selected") list box, and vice versa. Figure 11–22 shows an example of a rendered `selectManyShuttle` component. You can specify any text you want for the headers that display above the list boxes.

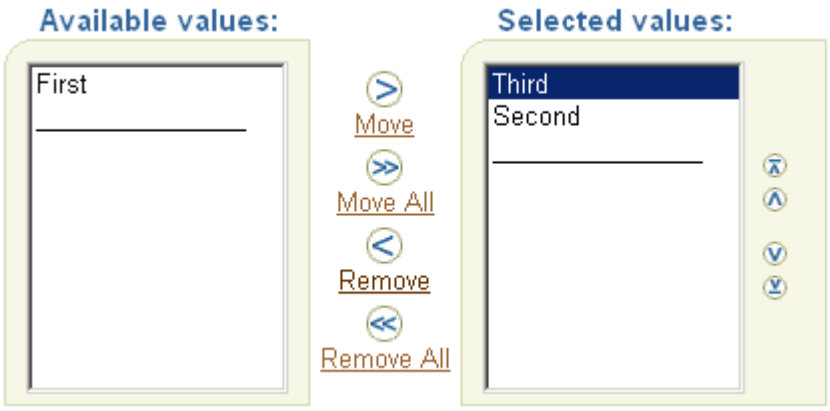
Figure 11–22 Shuttle (SelectManyShuttle) Component



Note: In addition to using the supplied **Move** and **Remove** buttons to shuttle items from one list to the other, you can also double-click an item in either list. Double-clicking an item in one list moves the item to the other list. For example, if you double-click an item in the leading list, the item is automatically moved to the trailing list, and vice versa.

The only difference between `selectManyShuttle` and `selectOrderShuttle` is that in the `selectOrderShuttle` component, the user can reorder the items in the trailing list box by using the up and down arrow buttons on the side, as shown in Figure 11–23.

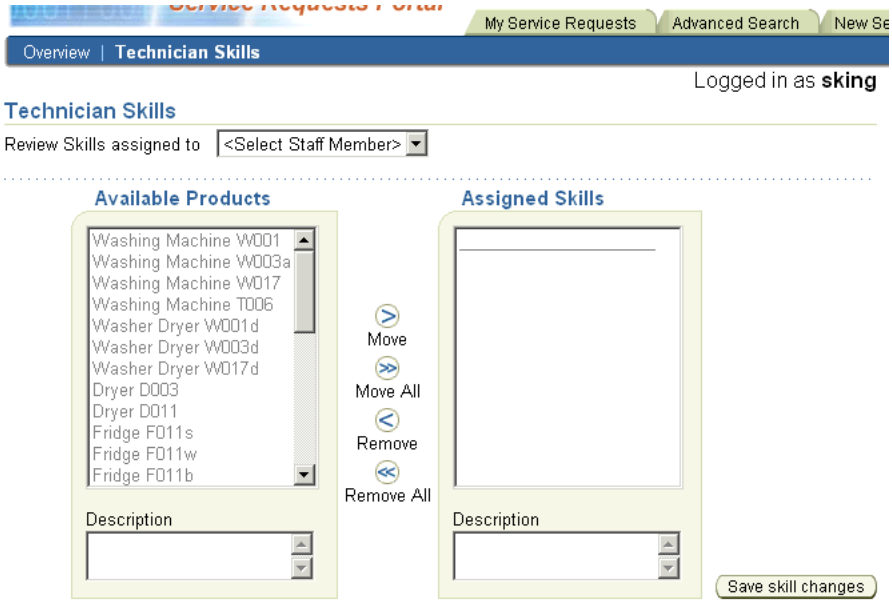
Figure 11–23 Shuttle Component (SelectOrderShuttle) with Reorder Buttons



11.8.1 How to Create a Databound Shuttle

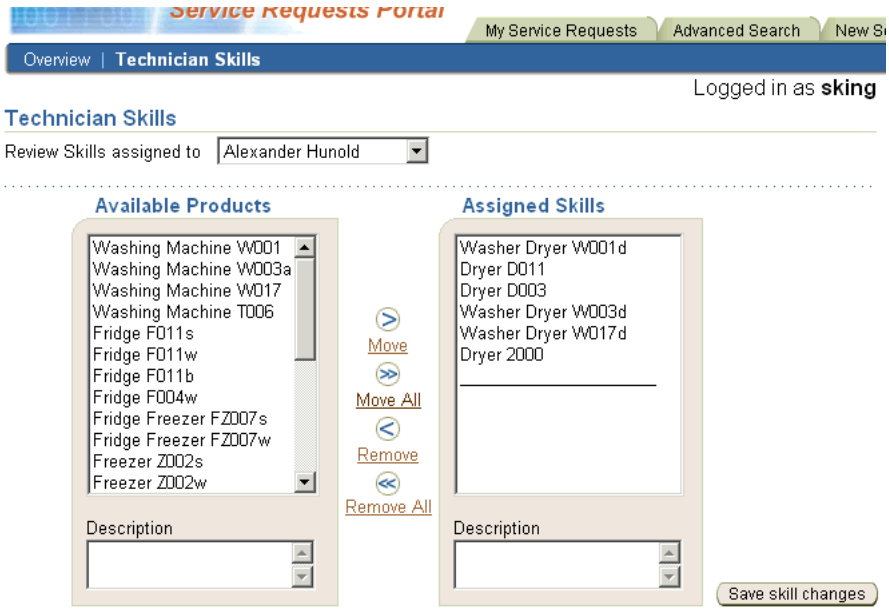
In the SRDemo application, the SRSkills page uses a `selectManyShuttle` component to let managers assign product skills to a technician. Figure 11–24 shows the SRSkills page created for the sample application. The leading list box on the left displays products such as washing machines and dryers; the trailing list box on the right displays the products that a technician is skilled at servicing.

Figure 11–24 SelectManyShuttle Component on the SRSkills Page



To review and change product skill assignments, a manager first selects a technician's name from the dropdown list above the shuttle component. The application then displays the technician's existing skill assignments in the trailing list, as shown in Figure 11–25.

Figure 11–25 Shuttle Component with the Trailing List Populated



Below the leading and trailing lists are optional boxes for displaying a description of a product. To view a description of a product, the manager can select an item from either list box, and the application displays the product's description in the box below the list.

To add new skill assignments, the manager selects the products from the leading list (**Available Products**) and then clicks the **Move** button.

To remove skills from the **Assigned Skills** list, the manager selects the products from the trailing list and then clicks the **Remove** button.

Like other ADF Faces selection list components, the `selectManyShuttle` component can use the `f:selectItems` tag to provide the list of items available for display and selection in the leading list.

Before you can bind the `f:selectItems` tag, create a class that maintains a list of the valid products (skills) for the shuttle, and the indexes of the products that are assigned to (selected for) a technician. The class should use the page's binding container to get the product master list for populating the shuttle's leading list. Example 11–59 shows the `SkillsHelper` class that is created to manage the population and selection state of the shuttle component on the SRSkills page.

Example 11–59 SkillsHelper Class

```
package oracle.srdemo.view;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
...
public class SkillsHelper {
    private BindingContainer bindings;
    private List<Product> productMasterList;
    private List <SelectItem> allProducts;
    private HashMap productLookUp;
    private int[] selectedProducts;
    private int skillsFor;
    private boolean skillsChanged = false;
    public List<SelectItem> getAllProducts() {
        if (allProducts == null) {
            OperationBinding oper =
getBindings().getOperationBinding("findAllProduct");
            productMasterList = (List<Product>)oper.execute();
            int cap = productMasterList.size();
            allProducts = new ArrayList(cap);
            productLookUp = new HashMap(cap);
            //for(Product prod: products) {
            for (int i=0;i<cap;i++){
                Product prod = productMasterList.get(i);
                SelectItem item = new SelectItem(i,
prod.getName(),prod.getDescription());
                allProducts.add(item);
                productLookUp.put(prod.getProdId(), i);
            }
        }

        return allProducts;
    }
```

```
    public void setAllProducts(List<SelectItem> allProducts) {
        this.allProducts = allProducts;
    }

    public void setSelectedProducts(int[] selectedProducts) {
        skillsChanged = true;
        this.selectedProducts = selectedProducts;
    }

    public int[] getSelectedProducts() {
        Integer currentTechnician =
(Integer)ADFUtils.getBoundAttributeValue(getBindings(),"currentTechnician");
        if (currentTechnician != null){
            if (skillsFor != currentTechnician.intValue()){
                skillsFor = currentTechnician.intValue();
                skillsChanged = false;
                OperationBinding getAssignedSkillsOp =
getBindings().getOperationBinding("findExpertiseByUserId");
                List<ExpertiseArea> skills =
(List<ExpertiseArea>)getAssignedSkillsOp.execute();
                selectedProducts = new int[skills.size()];
                for (int i=0;i<skills.size();i++){
                    Integer lookup =
(Integer)productLookUp.get(skills.get(i).getProdId());
                    selectedProducts[i] = lookup.intValue();
                }
            }
        }

        return selectedProducts;
    }

    public List<Integer> getSelectedProductIds(){
        ArrayList prodIdList = new ArrayList(selectedProducts.length);
        for (int i:selectedProducts){
            prodIdList.add(productMasterList.get(i).getProdId());
        }
        return prodIdList;
    }
    public void setBindings(BindingContainer bindings) {
        this.bindings = bindings;
    }

    public BindingContainer getBindings() {
        return bindings;
    }

    public void setSkillsChanged(boolean skillsChanged) {
        this.skillsChanged = skillsChanged;
    }

    public boolean isSkillsChanged() {
        return skillsChanged;
    }
}
```

The methods of interest in the `SkillsHelper` class are `getAllProducts()` and `getSelectedProducts()`.

The `getAllProducts()` method is the method that populates the shuttle's leading list. The first time this method is called, the `findAllProduct()` method on the `SRPublicFacade` session bean is invoked, and the list of products is cached in an array list of `SelectItem` objects. The `getAllProducts()` method also maintains a hashmap that enables reverse lookup of the list item index number based on the product ID.

The `getSelectedProducts()` method returns an array of `int` values, defining the list of items that appear on the shuttle's trailing list. This method also checks whether the currently selected technician (from the dropdown list above the shuttle) has changed. If the currently selected technician has changed, the `findExpertiseByUserId()` method on the `SRAdminFacade` session bean is invoked, and the new current technician's list of skills is retrieved and displayed in the trailing list of the shuttle.

The `SkillsHelper` class is maintained as a `session` scoped managed bean named `skillsHelper`. Example 11–60 shows the managed beans configured for working with the shuttle component in the SRDemo application.

Example 11–60 Managed Beans for the Shuttle Component in the faces-config.xml File

```
<managed-bean>
  <managed-bean-name>skillsHelper</managed-bean-name>
  <managed-bean-class>oracle.srdemo.view.SkillsHelper</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
  <managed-property>
    <property-name>bindings</property-name>
    <value>#{data.SRSkillsPageDef}</value>
  </managed-property>
</managed-bean>
<managed-bean>
  <managed-bean-name>backing_SRSkills</managed-bean-name>
  <managed-bean-class>oracle.srdemo.view.backing.SRSkills</managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
</managed-bean>
```

All the bindings of the SRSkills page are defined in the file `app_management_SRSkillsPageDef.xml`, a reference of which is injected into the `SkillsHelper` class. Example 11–61 shows the page definition file for the SRSkills page.

Example 11–61 Page Definition File for the SRSkills Page

```
<?xml version="1.0" encoding="UTF-8" ?>
<pageDefinition xmlns="http://xmlns.oracle.com/adfm/uimodel"
                version="10.1.3.36.2" id="SRSkillsPageDef"
                Package="oracle.srdemo.view.pageDefs"..>
  <executables>
    <methodIterator id="findAllStaffIter" Binds="findAllStaff.result"
                    DataControl="SRPublicFacade" RangeSize="-1"
                    BeanClass="oracle.srdemo.model.entities.User"/>
    <methodIterator id="findAllProductIter" Binds="findAllProduct.result"
                    DataControl="SRPublicFacade" RangeSize="-1"
                    BeanClass="oracle.srdemo.model.entities.Product"/>
    <variableIterator id="variables">
      <variable Name="selectedStaffIdVar" Type="java.lang.Integer"/>
    </variableIterator>
  </executables>
```

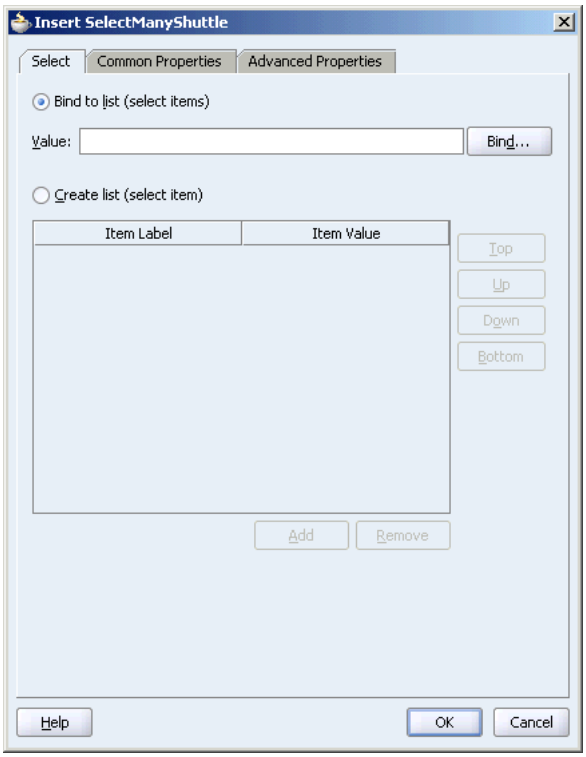
```
  <bindings>
    <methodAction id="findAllStaff" InstanceName="SRPublicFacade.dataProvider"
                  DataControl="SRPublicFacade" MethodName="findAllStaff"
                  RequiresUpdateModel="true" Action="999"
                  ReturnName="SRPublicFacade.methodResults.SRPublicFacade_
dataProvider_findAllStaff_result"/>
    <methodAction id="findAllProduct" InstanceName="SRPublicFacade.dataProvider"
                  DataControl="SRPublicFacade" MethodName="findAllProduct"
                  RequiresUpdateModel="true" Action="999"
                  ReturnName="SRPublicFacade.methodResults.SRPublicFacade_
dataProvider_findAllProduct_result"/>
    <attributeValues IterBinding="variables" id="currentTechnician">
      <AttrNames>
        <Item Value="selectedStaffIdVar"/>
      </AttrNames>
    </attributeValues>
    <methodAction id="findExpertiseByUserId"
                  InstanceName="SRAdminFacade.dataProvider"
                  DataControl="SRAdminFacade" MethodName="findExpertiseByUserId"
                  RequiresUpdateModel="true" Action="999"
                  ReturnName="SRAdminFacade.methodResults.SRAdminFacade_
dataProvider_findExpertiseByUserId_result">
      <NamedData NDName="userIdParam"
                 NDType="java.lang.Integer"
                 NDValue="#{bindings.currentTechnician.inputValue}"/>
    </methodAction>
    <list id="findAllStaffList" StaticList="false" ListOperMode="0"
          IterBinding="variables" ListIter="findAllStaffIter"
          NullValueFlag="1" NullValueId="findAllStaffList_null">
      <AttrNames>
        <Item Value="selectedStaffIdVar"/>
      </AttrNames>
      <ListAttrNames>
        <Item Value="userId"/>
      </ListAttrNames>
      <ListDisplayAttrNames>
        <Item Value="firstName"/>
        <Item Value="lastName"/>
      </ListDisplayAttrNames>
    </list>
    <methodAction id="updateStaffSkills"
                  InstanceName="SRAdminFacade.dataProvider"
                  DataControl="SRAdminFacade" MethodName="updateStaffSkills"
                  RequiresUpdateModel="true" Action="999">
      <NamedData NDName="userId"
                 NDValue="${bindings.currentTechnician.inputValue}"
                 NDType="java.lang.Integer"/>
      <NamedData NDName="prodIds" NDValue="${skillsHelper.selectedProductIds}"
                 NDType="java.util.List"/>
    </methodAction>
  </bindings>
</pageDefinition>
```

The next procedure assumes you've already created the relevant bindings, a class similar to the `SkillsHelper` class in Example 11–59, and configured the required managed beans in `faces-config.xml`, as shown in Example 11–60.

To create a shuttle component:

1. From the **ADF Faces Core** page of the Component Palette, drag and drop **SelectManyShuttle** onto the page. JDeveloper displays the Insert SelectManyShuttle dialog, as illustrated in Figure 11–26.

Figure 11–26 Insert SelectManyShuttle Dialog



2. Select **Bind to list (select items)** and click **Bind...** to open the Expression Builder.
3. In the Expression Builder, expand **JSF Managed Beans > skills**. Double-click **allProducts** to build the expression `#{skillsHelper.allProducts}`. Click **OK**.

   This binds the `f:selectItems` tag to the `getAllProducts()` method that populates the shuttle's leading list.
4. In the Insert SelectManyShuttle dialog, click **Common Properties**. Click **Bind...** next to the **Value** field to open the Expression Builder again.
5. In the Expression Builder, expand **JSF Managed Beans > skills**. Double-click **selectedProducts** to build the expression `#{skillsHelper.selectedProducts}`. Click **OK**.

   This binds the `value` attribute of the `selectManyShuttle` component to the `getSelectedProducts()` method that returns an array of `int` values, defining the list items on the shuttle's trailing list.

Example 11–62 shows the code for the `selectManyShuttle` component after you complete the Insert SelectManyShutle dialog.

Example 11–62 SelectManyShuttle Component in the SRSkills.jspx File

```
<af:selectManyShuttle value="#{skillsHelper.selectedProducts}"
                      ...
  <f:selectItems value="#{skillsHelper.allProducts}"/>
</af:selectManyShuttle>
```

For more information about using the shuttle component, see the ADF Faces Core tags at

http://www.oracle.com/technology/products/jdev/htdocs/partners/addins/exchange/jsf/doc/tagdoc/core/index.html

11.8.2 What Happens at Runtime

When the SRSkills page is first accessed, the variable iterator executes and instantiates its variable, `selectedStaffIdVar`. At this point, the variable does not contain a value. When the manager selects a name from the dropdown list, the variable is populated and the attribute binding can then provide the value for the `findExpertiseByUserId()` method's parameter `userIdParam`, using the EL expression for the value of the `NamedData` Element.

When the **Save skill changes** command button (see Figure 11–24) is clicked, the current technician's user ID and the associated array of product IDs (assigned skills) are retrieved and sent to the `updateStaffSkills()` method on the `SRAdminFacade` bean.

Example 11–63 shows the code for the `commandButton` component on the `SRSkills.jspx` page.

Example 11–63 CommandButton Component in the SRSkills.jspx File

```
<af:commandButton action="#{backing_SRSkills.saveSkillChanges_action}"
                  actionListener="#{bindings.updateStaffSkills.execute}"../>
```

12

Using Validation and Conversion

This chapter describes how to add validation and conversion capabilities to your application. It also describes how to handle and display any errors, including those not caused by validation.

This chapter includes the following sections:

- Section 12.1, "Introduction to Validation and Conversion"
- Section 12.3, "Adding Validation"
- Section 12.4, "Creating Custom JSF Validation"
- Section 12.5, "Adding Conversion"
- Section 12.6, "Creating Custom JSF Converters"
- Section 12.7, "Displaying Error Messages"
- Section 12.8, "Handling and Displaying Exceptions in an ADF Application"

12.1 Introduction to Validation and Conversion

ADF Faces input components have built-in validation capabilities. You set validation on a component either by setting the `required` attribute or by using one of the prebuilt ADF Faces validators. ADF applications also have validation capabilities at the model layer, allowing you to set validation on a binding to an attribute. In addition, you can create your own ADF Faces validators to suit your business needs.

ADF Faces input components also have built-in conversion capabilities, which allow users to enter information as `Strings`, and which the application can automatically convert to another data type, such as `Date`. Conversely, data stored as something other than a `String` can be converted to a `String` for display and updating.

Many components, such as `selectInputDate`, automatically provide this capability. Other components, such as `inputText`, automatically add a built-in ADF Faces or JSF reference implementation converter when you drag and drop from the Data Control Palette an attribute that is of a type for which a converter exists.

When validators or converters fail, associated error messages can be displayed to the user. These messages can be displayed in popup dialogs for client-side validation, or they can be displayed on the page itself next to the component whose validation or conversion failed.

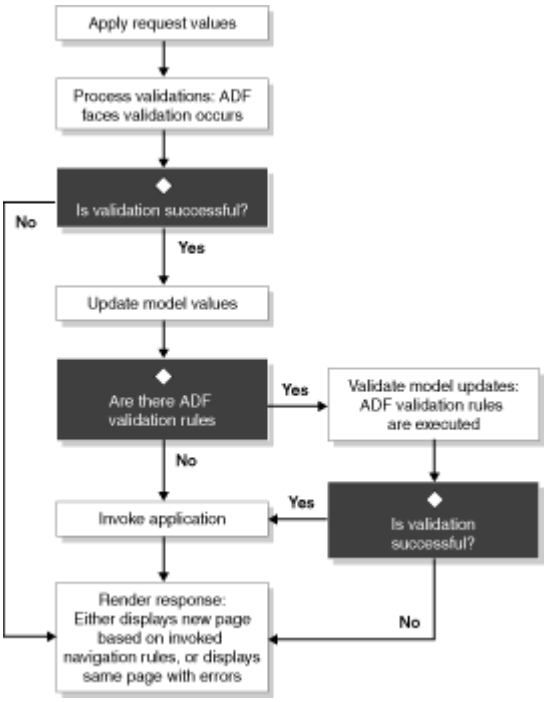
Read this chapter to understand:

- The different types of validation and how to add the capability to your application
- The ADF Faces converters and how to use them in an application
- The different ways you can display error messages
- How errors are handled by the ADF Model and displayed by ADF Faces error message components
- How exceptions thrown by the ADF application are handled, and how to customize the error handling process

12.2 Validation, Conversion, and the Application Lifecycle

Figure 12–1 shows how validation and conversion work in the integrated JSF and ADF lifecycle.

Figure 12–1 Validation and Conversion in the Lifecycle



When a form with data is submitted, the browser sends a request value to the server for each UI component whose `value` attribute is bound. The request value is first stored in an instance of the component in the JSF Apply Request Values phase. If the value requires conversion (for example, if it is displayed as a `String` but stored as a `DateTime` object), the data is converted to the correct type. Then, if you set ADF Faces validation for any of the components that hold the data, the value is validated against the defined rules during the Process Validations phase, before the value is applied to the model.

If validation or conversion fails, the lifecycle proceeds to the Render Response phase and a corresponding error message is displayed on the page. If validation and conversion are successful, then the `UpdateModel` phase starts and the validated and converted values are used to update the model.

At this point, if there are any ADF Model validation rules, the values are validated against those rules in the ADF Validate Model Updates phase. As with ADF Faces validation, if validation fails, the lifecycle proceeds to the Render Response phase. See Section 6.2.3, "What Happens at Runtime: The JSF and ADF Lifecycles" for more information.

When a validation or conversion error occurs, the component (in the case of JSF validation or conversion) or attribute (in the case of ADF Model validation) whose validation or conversion failed places an associated error message in the queue and invalidates itself. The current page is then redisplayed with an error message. Both ADF Faces components and the ADF Model provide a way of declaratively setting these messages. For information about how other errors are handled by an ADF application, see Section 12.8, "Handling and Displaying Exceptions in an ADF Application".

12.3 Adding Validation

You can add validation so that when a user edits or enters data in a field and submits the form, the data is validated against any set rules and conditions. If validation fails, the application displays an error message.

Those rules and conditions can be set at one of the following layers:

- View layer: You can use ADF Faces validation when you need client-side validation. Many ADF Faces components have attributes that provide validation. For information, see Section 12.3.1.1.1, "Using Validation Attributes". In addition, ADF Faces provides separate validation classes that can be run on both the client and the server. For details, see Section 12.3.1.1.2, "Using JSF and ADF Faces Validators". You can also create your own validators. For information about custom validators, see Section 12.4.3, "How to Create a Custom JSF Validator".
- Model layer: By default, when you use the Data Control Palette to create input text components, the components contain the `af:validator` tag that is bound to the `validator` property on the attribute's binding. This binding allows a JSF application to run ADF Model validation during the JSF Process Validations phase. To set ADF Model validation, you declaratively set validation rules on bindings to attributes of a collection. For more information, see Section 12.3.1.2, "Adding ADF Model Validation".
- Business layer: You can also set validation on objects in the business layer. An advantage to this type of validation is that it can be reused when that attribute's value is accessed by any page. However, it requires that the application accesses the business component in order for validation to be run. For the purposes of this chapter, only the view and model layer validation will be discussed.

12.3.1 How to Add Validation

You set ADF Faces validation on the JSF page and you set ADF Model validation on the page definition file. Message display for both is handled on the JSF page. For more information about displaying messages created by validation errors, see Section 12.7, "Displaying Error Messages".

12.3.1.1 Adding ADF Faces Validation

By default, ADF Faces validation occurs on both the client and server side. Although both syntactic and semantic validation are performed on the client side and server side, the client side performs only a subset of the validation performed by the server

side. Client-side validation allows validators to catch and display data without requiring a round-trip to the server.

> **Note:** If the JavaScript `form.submit()` function is called on a JSF page, the ADF Faces support for client-side validation is bypassed. ADF Faces provides a `submitForm()` method that you can use instead, or you could use the `autoSubmit` attribute on ADF Faces input components.

To set ADF Faces to not run client-side validation, add the `<client-validation-disabled>` element in `adf-faces-config.xml` and set it to `true`.

ADF Faces provides the following types of validation:

- UI component attributes: ADF Faces input components provide attributes that can be used to validate data. For example, you can supply simple validation using the `required` attribute on ADF Faces input components to specify whether a value must be supplied. When set to `true`, the component must have a value. Otherwise the application displays an error message. For more information, see Section 12.3.1.1.1, "Using Validation Attributes".
- Default ADF Faces validators: The validators supplied by ADF Faces and the JSF reference implementation provide common validation checks, such as validating date ranges and validating the length of entered data. For more information, see Section 12.3.1.1.2, "Using JSF and ADF Faces Validators".
- Custom ADF Faces validators: You can create your own validators and then select them to be used in conjunction with UI components. For more information, see Section 12.4, "Creating Custom JSF Validation".

12.3.1.1.1 Using Validation Attributes

Many ADF Faces UI components have attributes that provide simple validation. Table 12–1 shows these attributes, along with a description of the validation logic they provide and the UI components that contain them.

Table 12–1 ADF Faces Validation Attributes

Attribute	Description	Available on
`MaxValue`	The maximum value allowed for the `Date` value.	`chooseDate`
`MinValue`	The minimum value allowed for the `Date` value.	`chooseDate`
`Required`	When set to `true` (or set to an EL expression that evaluates to `true`), the component must have a non-null value or a `String` value of at least one character. For table selection components (see Section 7.6, "Enabling Row Selection in a Table"), if the `required` attribute is set to `true`, then at least one row in the table must be selected.	All input components, all select components, `tableSelectMany`, `tableSelectOne`
`MaximumLength`	The maximum number of characters that can be entered. Note that this value is independent of the value set for the `columns` attribute. See also `ByteLengthValidator` in Table 12–3, " ADF Faces Validators".	`inputText`

When you use the Data Control Palette to create input components, the `required` attribute is bound to the `mandatory` property of its associated binding, as shown in the following EL expression:

```
<af:inputText required="#{bindings.problemDescription.mandatory}"
```

The EL expression evaluates to whether or not the attribute on the object to which it is bound can be `null`. You can choose to keep the expression as is, or you can manually set the `required` attribute to "true" or "false".

> **Tip:** The object to which the UI component is bound varies depending on how the input component was created. For example, if a form was created using a method to create a parameter form, then the input components are usually bound to variables, since the attribute values do not yet exist. You need to set the `isNotNull` property on the variable if you wish to use the default EL expression. If a form was created using a collection returned by a method, then the input component is probably bound to an attribute on an entity object, and you need to set that object's `isNotNull` property.

To use UI component attributes that provide validation:

1. In the Structure window, select the UI component.
2. In the Property Inspector, enter a value for the validation attribute. See Table 12–1 for a list of validation attributes you could use.
3. (Optional) Set the `tip` attribute to display text that will guide the user to entering correct data (for example, a valid range for numbers). This text will display under the component.
4. (Optional) If you set the `required` attribute to `true` (or if you used an EL expression that can evaluate to `true`), you can also enter a value for the `RequiredMessageDetail` attribute. Instead of displaying a default message, ADF Faces will display this message, if validation fails.

   For tables with a selection component set to `required`, you must place the error message in the `summary` attribute of the table in order for the error message to display.

   Messages can include optional placeholders (such as {0}, {1}, and so on) for parameters. At runtime, the placeholders are replaced with the appropriate parameter values. The order of parameters is:

   - Component label input value (if present)
   - Minimum value (if present)
   - Maximum value (if present)
   - Pattern value (if present)

   Example 12–1 shows a `RequiredMessageDetail` attribute that uses parameters.

Example 12–1 Parameters in a RequiredMessageDetail Attribute

```
<af:inputText value="#{bindings.productId.inputValue}"
              label="Product ID"
              requiredMessageDetail="You must enter a {0}."
              required="true"
</af:inputText>
```

This message evaluates to `You must enter a Product ID`.

For additional help with UI component attributes, in the Property Inspector, right-click the attribute name and choose **Help**.

12.3.1.1.2 Using JSF and ADF Faces Validators

JSF and ADF Faces validators provide more complex validation routines. Table 12–2 describes the JSF reference implementation validators and Table 12–3 describes the built-in ADF Faces validators.

Table 12–2 JSF Reference Implementation Validators

Validator	Tag Name	Description
`DoubleRangeValidator`	`f:validateDoubleRange`	Validates that a component value is within a specified range. The value must be convertible to floating-point type or a floating-point.
`LengthValidator`	`f:validateLength`	Validates that the length of a component value is within a specified range. The value must be of type `java.lang.String`
`LongRangeValidator`	`f:validateLongRange`	Validates that a component value is within a specified range. The value must be any numeric type or `String` that can be converted to a `long`

Table 12–3 ADF Faces Validators

Validator	Tag Name	Description
`ByteLengthValidator`	`af:validateByteLength`	Validates the number of bytes in a `String` when Java encoding is used. For example, six English characters do not use the same byte storage as 6 Japanese characters. You specify the encoding to use as an attribute of the validator. In cases where the server must limit the number of bytes required to store a string, use this validator instead of specifying the `maximumLength` attribute on an input component.
`DateTimeRangeValidator`	`af:validateDateTimeRange`	Validates that the entered date is within a given range. You specify the range as attributes of the validator.
`RegExpValidator`	`af:validateRegExp`	Validates the data using Java regular expression syntax.

Note: ADF Faces also provides the `af:validator` tag, which you can use to register a custom validator on a component. For information about using custom validators, see Section 12.4, "Creating Custom JSF Validation".

By default, whenever you drop an attribute from the Data Control Palette as an input text component, JDeveloper automatically adds the `af:validator` tag to the component, and binds it to the `validator` property on the associated binding. The binding allows access to ADF Model validation for processing on the client side. For more information, see Section 12.3.2, "What Happens When You Create Input Fields Using the Data Control Palette". For information about ADF Model validation, see Section 12.3.1.2, "Adding ADF Model Validation".

To add ADF Faces validators:

1. In the Structure window, right-click the component for which you'd like to add a validator.
2. In the context menu, choose **Insert inside <UI component> > ADF Faces Core** to insert an ADF Faces validator. (To insert a JSF reference implementation validator, choose **Insert inside <UI component> > JSF Core**.)
3. Choose a validator tag (for example, **ValidateDateTimeRange**).
4. In the Property Inspector, set values for the attributes, including any messages for validation errors. For additional help, right-click any of the attributes and choose **Help**.

   ADF Faces lets you customize the detail portion of a validation error message. By setting a value for an **XxxMessageDetail** attribute, where **Xxx** is the validation error type (for example, `maximumMessageDetail`), ADF Faces displays the custom message instead of a default message, if validation fails.

12.3.1.2 Adding ADF Model Validation

Table 12–4 describes the ADF Model validation rules that you can configure for an attribute.

Table 12–4 ADF Model Validation Rules

Validator Rule Name	Description
Compare	Compares the attribute's value with a literal value
List	Validates whether or not the value is in or is not in a list of values
Range	Validates whether or not the value is within a range of values
Length	Validates the value's character or byte size against a size and operand (such as greater than or equal to)
Regular Expression	Validates the data using Java regular expression syntax

To create an ADF Model validation rule:

1. Open the page definition that contains the attribute for which you want to create a rule.
2. In the Structure window, select the attribute, list, or table binding.

3. In the Property Inspector, select the **Edit Validation Rule** link.
4. In the Validation Rules Editor, select the attribute name and click **New**.
5. In the Add Validation Rule dialog, select a validation rule and configure the rule accordingly. For additional help on creating the different types of rules, click **Help**.

12.3.2 What Happens When You Create Input Fields Using the Data Control Palette

When you use the Data Control Palette to create input text fields (for example, by dropping an attribute from the Data Control Palette as an `inputText` component), JDeveloper automatically provides ADF Faces validation code on the JSF page by:

- Adding an `af:messages` tag as a child of the `afh:body` tag.

  By default the `globalOnly` attribute is set to `false`, and the `message` and `text` attributes are not set. For more information, see Section 12.7, "Displaying Error Messages".

- Binding the `required` attribute for input fields to the `mandatory` property of the associated attribute binding, as shown in the following EL expression:

  ```
  <af:inputText required="#{bindings.problemDescription.mandatory}"
  ```

  The expression evaluates to whether or not a `null` value is allowed based on the attribute of the associated business object. By default, all components whose `required` attribute evaluates to `true` will display an asterisk.

- Adding an `af:validator` tag as a child of the input component, and binding the tag to the `validator` property of the associated binding, as shown below:

  ```
  <af:inputText value="#{bindings.someAttribute.inputValue}"...>
    <af:validator binding="#{bindings.someAttribute.validator}"/>
  </af:inputText>
  ```

  The binding allows the JSF lifecycle to access, on the client side, any ADF Model validation that you may have set for the associated attribute. If you don't wish to use ADF Model validation, then you can delete the `af:validator` tag and insert the validation tag of your choice, or if you don't want to use any validation, you can simply delete the tag. If you do want to use only ADF Model validation, you must keep the tag as is.

  > **Tip:** If you delete the `af:validator` tag and its binding, and want to add ADF Model validation at a later point, you must add the tag back into the code with the `binding` attribute bound to the associated attribute's `validator` property.

To create a simple input form for products in the SRDemo application, for example, you might drop the product constructor method from the Data Control Palette as a parameter form. Example 12–2 shows the JSF code created by JDeveloper.

Example 12–2 JSF Code for a Create Product Page

```
<afh:body>
  <af:messages/>
  <h:form>
    <af:panelForm>
      <af:inputText value="#{bindings.productId.inputValue}"
                    label="#{bindings.productId.label}"
                    required="#{bindings.productId.mandatory}"
                    columns="#{bindings.productId.displayWidth}">
```

```
        <af:validator binding="#{bindings.productId.validator}"/>
        <f:convertNumber groupingUsed="false"
                         pattern="#{bindings.productId.format}"/>
      </af:inputText>
      <af:inputText value="#{bindings.name.inputValue}"
                    label="#{bindings.name.label}"
                    required="#{bindings.name.mandatory}"
                    columns="#{bindings.name.displayWidth}">
        <af:validator binding="#{bindings.name.validator}"/>
      </af:inputText>
      <af:inputText value="#{bindings.image.inputValue}"
                    label="#{bindings.image.label}"
                    required="#{bindings.image.mandatory}"
                    columns="#{bindings.image.displayWidth}">
        <af:validator binding="#{bindings.image.validator}"/>
      </af:inputText>
      <af:inputText value="#{bindings.description.inputValue}"
                    label="#{bindings.description.label}"
                    required="#{bindings.description.mandatory}"
                    columns="#{bindings.description.displayWidth}">
        <af:validator binding="#{bindings.description.validator}"/>
      </af:inputText>
    </af:panelForm>
    <af:commandButton actionListener="#{bindings.createProducts.execute}"
                      text="createProducts"
                      disabled="#{!bindings.createProducts.enabled}"/>
  </h:form>
</afh:body>
```

Note that each `inputText` component's `required` attribute is bound to the `mandatory` property of its associated binding. The EL expression evaluates to whether or not the attribute on the object to which it is bound can be `null`.

When you create an ADF Model validation rule for an attribute, JDeveloper adds the validation rule to the attribute binding, which in turn references the associated validation bean and provides the needed property values for the validation to run. Example 12–3 shows the page definition code created if you add a `Length` validation rule to the `productDescription` attribute setting the maximum size for the attribute to `20`.

Example 12–3 Page Definition Validation Rule

```
<attributeValues id="description" IterBinding="variables"
                 ApplyValidation="true">
  <LengthValidationBean xmlns="http://xmlns.oracle.com/adfm/validation"
            OnAttribute="createProducts_description"
            DataType="CHARACTER" CompareType="LESSTHAN"
            ResId="description_Rule_0" Inverse="false"
            CompareLength="20"/>
  <AttrNames>
     <Item Value="createProducts_description"/>
  </AttrNames>
</attributeValues>
```

12.3.3 What Happens at Runtime

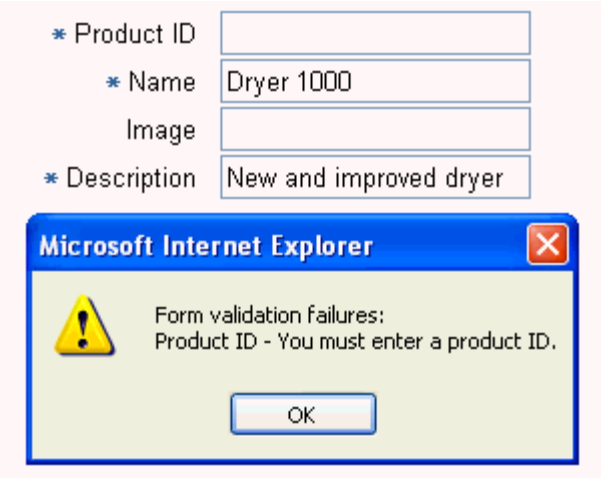
When the user submits the page, the ADF Faces `validate()` method first checks for a submitted value if the `required` attribute of a component is `true`. If the value is `null` or a zero-length string, the component is invalidated. At this point, what happens depends on whether or not client-side validation is enabled.

If client-side validation is enabled, an error message is placed in the queue. If there are other validators registered on the component, they are not called at all, and the current page is redisplayed with a dialog displaying the error message.

Note: JSF reference implementation validators are not run on the client side.

In Example 12–2, the image attribute is not required. However, all other columns are required, as set by the `mandatory` property. This is denoted in the web page by asterisk icons. Figure 12–2 shows the error message displayed if no data is entered for the product ID, and if client-side validation is enabled.

Figure 12–2 Client-Side Error for a Required Value



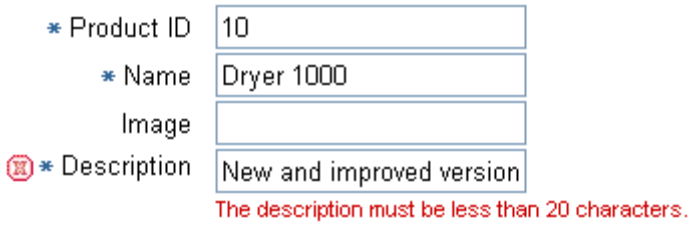
If the submitted value is a non-null value or a string value of at least one character, the validation process continues and all validators on the component are called one at a time. Because the `af:validator` tag on the component is bound to the `validator` property on the binding, any validation routines set on the model are also accessed and executed at this time.

The process then continues to the next component. If all validations are successful, the Update Model Values phase starts and a local value is used to update the model. If any validation fails, the current page is redisplayed along with the error dialog.

When client-side validation is disabled, all validations are done on the server. First, the ADF Faces validation is performed during the Process Validations phase. If any errors are encountered, the values are invalidated and the associated messages are added to the queue in `FacesContext`. Once all validation is run on the components, control passes to the model layer, which runs the Validate Model Updates phase. As with the Process Validations phase, if any errors are encountered, the values are invalidated and the associated messages are added to the queue in `FacesContext` (for information on how errors other than validation or conversion are handled, see Section 12.8, "Handling and Displaying Exceptions in an ADF Application").

The lifecycle then jumps to the Render Response phase and redisplays the current page. ADF Faces automatically displays an error icon next to the label of any input component that generated an error, and it displays the associated messages below the input field. If there is a tip associated with the field, the error message displays below the tip. Figure 12–3 shows a server-side validation error.

Figure 12–3 Server-Side Validation Error



12.3.4 What You May Need to Know

You can both set the `required` attribute and use validators on a component. However, if you set the required attribute to `true` and the value is `null` or a zero-length string, the component is invalidated and any other validators registered on the component are not called.

This combination might be an issue if there is a valid case for the component to be empty. For example, if the page contains a **Cancel** button, the user should be able to click that button and navigate off the page without entering any data. To handle this case, you set the `immediate` attribute on the **Cancel** button's component to `true`. This attribute allows the action to be executed during the Apply Request Values phase, thus bypassing the validation whenever the action is executed.

12.4 Creating Custom JSF Validation

You can add your own validation logic to meet your specific business needs. If you need custom validation logic for a component on a single page, you can create a validation method on the page's backing bean. Creating the validation method on a backing bean is also useful when you need validation to access other fields on the page. For example, if you have separate date fields (month, day, year) and each has its own validator, users will not get an error if they enter February 30, 2005. Instead, a backing bean for the page can contain a validation method that validates the entire date.

If you need to create logic that will be reused by various pages within the application, or if you want the validation to be able to run on the client side, you should create a JSF validator class. You can then create an ADF Faces version, which will allow the validator to run on the client.

12.4.1 How to Create a Backing Bean Validation Method

When you need custom validation for a component on a single page, you can create a method that provides the needed validation on a backing bean.

To add a backing bean validation method:

1. Insert the component that will require validation into the JSF page.
2. In the visual editor, double-click the component to launch the Bind Validator Property dialog.
3. In the Bind Validator Property dialog, enter or select the managed bean that will hold the validation method, or click **New** to create a new managed bean. Use the default method signature provided or select an existing method if the logic already exists.

   When you click **OK** in the dialog, JDeveloper adds a skeleton method to the code and opens the bean in the source editor.
4. Add the needed validation logic. This logic should use `javax.faces.validator.ValidatorException` to throw the appropriate exceptions and `javax.faces.application.FacesMessage` to generate the corresponding error messages. For more information about the `Validator` interface and `FacesMessage`, see the Javadoc for `javax.faces.validator.Validator` and `javax.faces.application.FacesMessage`, or visit `http://java.sun.com/`.

12.4.2 What Happens When You Create a Backing Bean Validation Method

When you create a validation method, JDeveloper adds a skeleton method to the managed bean you selected. Example 12–4 shows the code JDeveloper generates.

Example 12–4 Managed Bean Code for a Validation Method

```
public void inputText_validator(FacesContext facesContext,
                                UIComponent uiComponent, Object object) {
        // Add event code here...
}
```

The SREdit page in the SRDemo application uses a validation method to ensure that the new date entered is not earlier than the original date. Example 12–5 shows the validation method on that page's backing bean.

Example 12–5 SREdit Date Validation Method

```
public void assignedDateValidator(FacesContext facesContext,
                                  UIComponent uiComponent,
                                  Object newValue) {
    //The new value is passed into us
    Timestamp newAssignedDate = (Timestamp)newValue;

    //Get the start date for the SR which is already bound on this screen
    Timestamp requestDate =
        (Timestamp)ADFUtils.getBoundAttributeValue(getBindings(),
                                                   "requestDate");
    // Now compare and raise an error if the rule is broken
    if (newAssignedDate.compareTo(requestDate) < 0)
        {
          throw new ValidatorException(JSFUtils.getMessageFromBundle
           ("sredit.error.assignedBeforeStart", FacesMessage.SEVERITY_ERROR));
        }

 }
```

JDeveloper binds the validator attribute of the component to the backing bean's validation method using an EL expression. Example 12–6 shows the code JDeveloper adds to the SREdit page.

Example 12–6 JSF Code for a Custom Validation Method

```
<af:selectInputDate value="#{bindings.assignedDate.inputValue}"
                    label="#{bindngs.assignedDate.label}"
                    ...
                    validator="#{backing_SREdit.assignedDateValidator}">
```

> **Tip:** JDeveloper also adds an `af:validator` tag that is bound to the `validator` property of the associated binding. This allows the JSF lifecycle to access any ADF Model validation you may have set for the associated attribute. If you do not set any ADF Model validation, you may remove this binding.

When the form containing the input component is submitted, the method to which the `validator` attribute is bound is executed.

12.4.3 How to Create a Custom JSF Validator

Creating a custom validator requires writing the business logic for the validation by creating a `Validator` implementation that contains a method overriding the `validate` method of the `Validator` interface, and then registering the custom validator with the application. You can also create a tag for the validator, or you can use the `af:validator` tag and nest the custom validator as a property of that tag.

You can then create a client-side version of the validator. ADF Faces client-side validation works in the same way that standard validation works on the server, except that JavaScript is used on the client: JavaScript validator objects can throw `ValidatorExceptions`, and they support the `validate()` method.

> **Note:** If the JavaScript `form.submit()` function is called, the ADF Faces support for client-side validation is bypassed. ADF Faces provides a `submitForm()` method that you can use instead, or you can use the `autoSubmit` attribute on ADF Faces input components.

To create a custom JSF validator:

1. Create a Java class that implements the `javax.faces.validator.Validator` interface. The implementation must contain a public no-args constructor, a set of accessor methods for any attributes, and a `validate` method that overrides the `validate` method of the `Validator` interface.

   Alternatively, you can implement the `javax.faces.FormatValidator` interface, which has accessor methods for setting the `formatPatterns` attribute. This attribute specifies the acceptable patterns for the data entered into the input component. For example, if you want to validate the pattern of a credit card number, you create a `formatPatterns` attribute for the allowed patterns. The implementation must contain a constructor, a set of accessor methods for any attributes, and a `validate` method that overrides the `validate` method on the `Validator` interface.

   For both interfaces, the `validate` method takes the `FacesContext` instance, the UI component, and the data to be validated. For example:

   ```
   public void validate(FacesContext facesContext,
                        UIComponent uiComponent,
                        Object object) {
   ..
   }
   ```

   For more information about these classes, refer to the Javadoc or visit `http://java.sun.com/`.

2. Add the needed validation logic. This logic should use `javax.faces.validator.ValidatorException` to throw the appropriate exceptions and `javax.faces.application.FacesMessage` to generate the corresponding error messages. For more information about the `Validator` interface and `FacesMessage`, see the Javadoc for `javax.faces.validator.Validator` and `javax.faces.application.FacesMessage`, or visit http://java.sun.com/.

   > **Note:** To allow the page author to configure the attributes from the page, you need to create a tag for the validator. See step 5 for more information. If you don't want the attributes configured on the page, then you must configure them in this implementation class.

3. If your application saves state on the client, make your custom validator implementation implement the `Serializable` interface, or implement the `StateHolder` interface, and the `saveState(FacesContext)` and `restoreState(FacesContext, Object)` methods of `StateHolder`. For more information, see the Javadoc for the `StateHolder` interface of the `javax.faces.component` package.

4. Register the validator in the `faces-config.xml` file.
   - Open the `faces-config.xml` file and select the **Overview** tab in the editor window. The `faces-config.xml` file is located in the `<View_Project>/WEB-INF` directory.
   - In the window, select **Validators** and click **New**. Click **Help** or press F1 for additional help in registering the validator.
5. Optionally create a tag for the validator that sets the attributes for the class. You create a tag by adding an entry for the tag in the application's tag library definition file (TLD). To do so:
   - Open or create a TLD for the application. For more information about creating a TLD, visit http://java.sun.com/.
   - Define the validator ID and class as registered in the `faces-config.xml` file.
   - Define any properties or attributes as registered in that configuration file.

   Note: If you do not create a tag for the validator, you must configure any attributes in the `Validator` implementation.

To create a client-side version of the validator:

1. Write a JavaScript version of the validator, passing relevant information to a constructor.
2. Implement the interface `oracle.adf.view.faces.validator.ClientValidator`, which has two methods. The first method is `getClientScript()`, which returns an implementation of the JavaScript `Validator` object. The second method is `getClientValidation()`, which returns a JavaScript constructor that is used to instantiate an instance of the validator.

For a complete example of how to add client-side validation to a validator implementation, see "Client-Side Converters and Validators" in *Development Guidelines for Oracle ADF Faces Applications*.

To use a custom validator on a JSF page:

- To use a custom validator that has a tag on a JSF page, you need to manually nest it inside the component's tag.

  Example 12–7 shows a custom validator nested inside an `inputText` component. Note that the tag attributes are used to provide the values for the validator's properties that were declared in the `faces-config.xml` file.

Example 12–7 A Custom Validator Tag on a JSF Page

```
<h:inputText id="empnumber" required="true">
  <hdemo:emValidator emPatterns="9999|9 9 9 9|9-9-9-9" />
</h:inputText>
```

To use a custom validator without a custom tag:

To use a custom validator without a custom tag, you must nest the validator's ID (as configured in `faces-config.xml` file) inside the `af:validator` tag.

1. From the Structure window, right-click the input component for which you want to add validation, and choose **Insert inside <component>** > **ADF Faces Core** > **Validator**.
2. Select the validator's ID from the dropdown list and click **OK**.

   JDeveloper inserts code on the JSF page that makes the validator ID a property of the `validator` tag.

Example 12–8 shows the code on a JSF page for a validator using the `af:validator` tag.

Example 12–8 A Custom Validator Nested Within a Component on a JSF Page

```
<af:inputText id="empnumber" required="true">
  <af:validator validatorID="emValidator"/>
</af:inputText>
```

12.4.4 What Happens When You Use a Custom JSF Validator

When you use a custom JSF validator, the application accesses the validator class referenced in either the custom tag or the `af:validator` tag and executes the `validate` method. This method accesses the data from the component in the current `FacesContext` and executes logic against it to determine if it is valid. If the validator has attributes, those attributes are also accessed and used in the validation routine. Like standard validators, if the custom validation fails, associated messages are placed in the message queue in `FacesContext`.

12.5 Adding Conversion

A web application can store data of many types (such as `int`, `long`, `date`) in the model layer. When viewed in a client browser, however, the user interface has to present the data in a manner that can be read or modified by the user. For example, a date field in a form might represent a `java.util.Date` object as a text string in the format pattern `mm/dd/yyyy`. When a user edits a date field and submits the form, the string must be converted back to the type that is required by the application. Then the data is validated against any rules and conditions.

When you create an `inputText` component by dropping an attribute that is of a type for which there is a converter, JDeveloper automatically adds that converter's tag as a child of the input component. This tag invokes the converter, which will convert the `String` entered by the user back into the type expected by the object.

The JSF standard converters, which handle conversion between `Strings` and simple data types, implement the `javax.faces.convert.Converter` interface. The supplied JSF standard converter classes are:

- `BigDecimalConverter`
- `BigIntegerConverter`
- `BooleanConverter`
- `ByteConverter`
- `CharacterConverter`
- `DateTimeConverter`
- `DoubleConverter`
- `FloatConverter`
- `IntegerConverter`
- `LongConverter`
- `NumberConverter`
- `ShortConverter`

Table 12–5 shows the converters provided by ADF Faces.

Table 12–5 ADF Faces Converters

Converter	Tag Name	Description
`ColorConverter`	`af:convertColor`	Converts `java.lang.String` objects to `java.awt.Color` objects. You specify a set of color patterns as an attribute of the converter.
`DateTimeConverter`	`af:convertDateTime`	Converts `java.lang.String` objects to `java.util.Date` objects. You specify the pattern and style of the date as attributes of the converter.
`NumberConverter`	`af:convertNumber`	Converts `java.lang.String` objects to `java.lang.Number` objects. You specify the pattern and type of the number as attributes of the converter.

As with validators, the ADF Faces converters are also run on the client side unless client-side validation is explicitly disabled in the `adf-faces-config.xml` file.

Note: JSF reference implementation converters are not run on the client-side

In addition to JavaScript-enabled converters for color, date, and number, ADF Faces also provides JavaScript-enabled converters for input text fields that are bound to any of these Java types:

- `java.lang.Integer`
- `java.lang.Long`
- `java.lang.Short`
- `java.lang.Byte`
- `java.lang.Float`
- `java.lang.Double`

Unlike the converters listed in Table 12–5, the JavaScript-enabled converters are automatically used whenever needed. They do not have associated tags that can be nested in the component.

12.5.1 How to Use Converters

Whenever you drop an attribute from the Data Control Palette for which there is an ADF Faces converter, JDeveloper automatically adds the converter to the input component. You can also manually insert a converter.

To add ADF Faces converters that have a tag:

1. In the Structure window, right-click the component for which you'd like to add a converter.
2. In the context menu, choose **Insert inside <UI component> > ADF Faces Core** to insert an ADF Faces converter or **JSF Core** to insert a JSF converter.
3. Choose a converter tag (for example, **ConvertDateTime**).
4. In the Property Inspector, set values for the attributes, including any messages for conversion errors. For additional help, right-click any of the attributes and choose **Help**.

   ADF Faces lets you customize the detail portion of a conversion error message. By setting a value for an **XxxMessageDetail** attribute, where **Xxx** is the conversion error type (for example, `convertDateMessageDetail`), ADF Faces displays the custom message instead of a default message, if conversion fails.

12.5.2 What Happens When You Create Input Fields Using the Data Control Palette

When you use the Data Control Palette to create input fields that are of a type supported by a converter, JDeveloper automatically provides ADF Faces conversion code on the JSF page by:

- Adding an `af:messages` tag as a child of the `body` tag. By default the `globalOnly` attribute is set to `false`, and the message and text attributes are not set. For more information, see Section 12.7, "Displaying Error Messages".
- Adding a converter tag as a child of the input component.

  By default, the `pattern` attribute is bound to the `format` property of the associated binding. The `format` property determines how the `String` is formatted. For example, for the `convertNumber` converter, it might determine whether decimals are used. This binding evaluates to the `format` property as it is set on the data control itself.

For example, if you drop the `prodId` attribute from the `findAllProducts` method as an `inputText` component, JDeveloper automatically adds the `convertNumber` converter as a child of the input component, as shown in Example 12–9.

Example 12–9 Converter Tag in a JSF Page

```
<af:inputText value="#{bindings.productId.inputValue}"
              label="#{bindings.productId.label}"
              required="#{bindings.productId.mandatory}"
              columns="#{bindings.productId.displayWidth}">
  <af:validator binding="#{bindings.productId.validator}"/>
  <f:convertNumber groupingUsed="false"
                   pattern="#{bindings.productId.format}"/>
</af:inputText>
```

12.5.3 What Happens at Runtime

When the user submits the page containing converters, the ADF Faces `validate()` method calls the converter's `getAsObject()` method to convert the string value to the required object type. When there isn't an attached converter and if the component is bound to a bean property in the model, then JSF automatically uses the converter that has the same data type as the bean property. If conversion fails, the submitted value is marked as invalid and JSF adds an error message to a queue that is maintained by `FacesContext`. If conversion is successful and there are no validators attached to the component, the converted value is stored as a local value that is later used to update the model.

12.6 Creating Custom JSF Converters

You can create your own converters to meet your specific business needs. As with creating custom JSF validators, you can create custom JSF converters that run on the server side, and then also create a JavaScript version that can run on the client side. However, unlike creating custom validators, you can create only converter classes. You cannot add a method to a backing bean to provide conversion.

12.6.1 How to Create a Custom JSF Converter

Creating a custom converter requires writing the business logic for the conversion by creating an implementation of the `Converter` interface that contains methods overriding the `getAsObject` and `getAsString` methods of the `Converter` interface, and then registering the custom converter with the application. You then use the `f:converter` tag and nest the custom converter as a property of that tag, or you can use the `converter` attribute on the input component to bind to that converter.

You can also create a client-side version of the converter. ADF Faces client-side converters work in the same way standard JSF conversion works on the server, except that JavaScript is used on the client: JavaScript converter objects can throw `ConverterExceptions`, and they support the `getAsObject` and `getAsString` methods.

> **Note:** If the JavaScript `form.submit()` function is called, the ADF Faces support for client-side conversion is bypassed. ADF Faces provides a `submitForm()` method that you can use instead, or you can use the `autoSubmit` attribute on ADF Faces input components.

To create a custom JSF converter:

1. Create a Java class that implements the `javax.faces.converter.Converter` interface. The implementation must contain a public no-args constructor, a set of accessor methods for any attributes, and `getAsObject` and `getAsString` methods, which override the same methods of the `Converter` interface.

   The `getAsObject` method takes the `FacesContext` instance, the UI component, and the String to be converted to a specified object. For example:

   ```
   public Object getAsObject(FacesContext context,
                             UIComponent component,
                             java.lang.String value){
   ..
   }
   ```

   The `getAsString` method takes the `FacesContext` instance, the UI component, and the object to be converted to a String. For example:

   ```
   public String getAsString(FacesContext context,
                             UIComponent component,
                             Object value){
   ..
   }
   ```

   For more information about these classes, refer to the Javadoc or visit http://java.sun.com/.

2. Add the needed conversion logic. This logic should use `javax.faces.converter.ConverterException` to throw the appropriate exceptions and `javax.faces.application.FacesMessage` to generate the corresponding error messages. For more information about the `Converter` interface and `FacesMessage`, see the Javadoc for `javax.faces.converter.Converter` and `javax.faces.application.FacesMessage`, or visit http://java.sun.com/.

3. If your application saves state on the client, make your custom converter implementation implement the `Serializable` interface, or implement the `StateHolder` interface, and the `saveState(FacesContext)` and `restoreState(FacesContext, Object)` methods of `StateHolder`. For more information, see the Javadoc for the `StateHolder` interface of `javax.faces.component` package.

4. Register the converter in the `faces-config.xml` file.

   - Open the `faces-config.xml` file and select the **Overview** tab in the editor window. The `faces-config.xml` file is located in the `<View_Project>/WEB-INF` directory.
   - In the window, select **Converters** and click **New**. Click **Help** or press F1 for additional help in registering the converter.

To create a client-side version of the converter:

1. Write a JavaScript version of the converter, passing relevant information to a constructor.
2. Implement the interface `oracle.adf.view.faces.converter.ClientConverter`, which has two methods. The first method is `getClientScript()`, which returns an implementation of the JavaScript `Converter` object. The second method is `getClientConversion()`, which returns a JavaScript constructor that is used to instantiate an instance of the converter.

For a complete example of how to add client-side conversion to a converter implementation, see "Client-Side Converters and Validators" in *Development Guidelines for Oracle ADF Faces Applications*.

To use a custom converter on a JSF page:

- Bind your converter class to the `converter` attribute of the input component.

Example 12–10 shows a custom converter referenced by the `converter` attribute of an `inputText` component.

Example 12–10 A Custom Converter on a JSF Page

```
<af:inputText value="#{bindings.name.inputValue}"
              label="#{bindings.name.label}"
              required="#{bindings.name.mandatory}"
              columns="#{bindings.name.displayWidth}"
              converter="srdemo.MyConverter">
</af:inputText>
```

Note: If a custom converter is registered in an application under a class for a specific data type, whenever a component's value references a value binding that has the same type as the custom converter object, JSF will automatically use the converter of that class to convert the data. In that case, you don't need to use the `converter` attribute to register the custom converter on a component, as shown in the following code snippet:

```
<h:inputText value="#{myBean.myProperty}"/>
```

where `myProperty` has the same type as the custom converter.

12.6.2 What Happens When You Use a Custom Converter

When you use a custom converter, the application accesses the converter class referenced in the `converter` attribute, and executes the `getAsObject` or `getAsString` method as appropriate. These methods access the data from the component in the current `FacesContext` and execute the conversion logic.

12.7 Displaying Error Messages

By default, ADF Faces validation and conversion run on the client side. When a component's data fails validation, an alert dialog displays an error message for the component. You do not need to do any additional work to have client-side error messages display in this way. Figure 12–4 shows the message displayed when data is not entered for an input component that has a `required` attribute set to `true`.

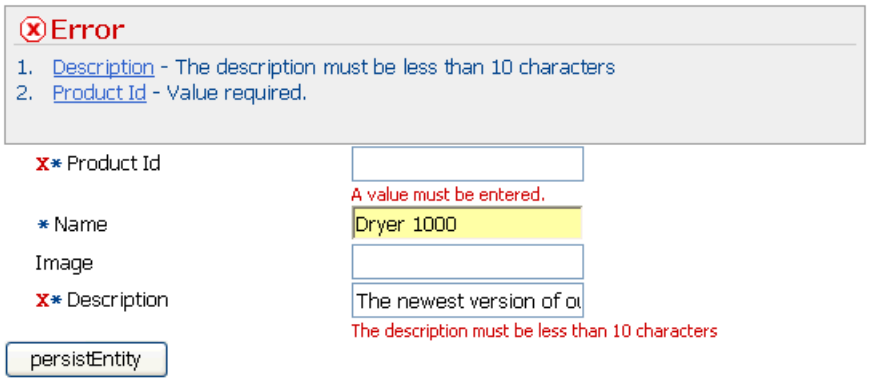
Figure 12–4 A Client-Side Error Message

ADF Faces provides default text for messages displayed when validation or conversion fails. You can replace the default messages with your own messages by setting the text on the `xxxMessageDetail` attributes of the validator or converter (such as `convertDateMessageDetail` or `notInRangeDetailMessage`), or by binding those attributes to a resource bundle using an EL expression. For more information about using resource bundles, see Section 14.4.1, "How to Internationalize an Application".

When you use the Data Control Palette to create input components, JDeveloper inserts the `af:messages` tag at the top of the page. This tag can display all error messages in the queue for any validation that occurs on the server side, in a box offset by color. If you choose to turn off client-side validation for ADF Faces, those error messages are displayed along with any ADF Model error messages. ADF Model messages are shown first. Messages are shown both within the `af:messages` tag and with the associated components.

Figure 12–5 shows the error message for an ADF Model validation rule, which states that the description is too long, along with an error message for an ADF Faces component `required` attribute violation.

Figure 12–5 Displaying Server-Side Error Messages With the ADF Faces Messages Tag



12.7.1 How to Display Server-Side Error Messages on a Page

You can display server-side error messages in a box at the top of a page using the `af:messages` tag. When you drop any item from the Data Control Palette onto a page as an input component, JDeveloper automatically adds this tag for you.

To display error messages in an error box:

1. In the Structure window, select the `af:messages` tag.

   This tag is created automatically whenever you drop an input widget from the Data Control Palette. However, if you need to insert the tag, simply insert the following code within the `afh:body` tag:

   ```
   <afh:body>
     <af:messages globalOnly="false" />
     ...
   </afh:body>
   ```

2. In the Property Inspector set the following attributes:

   - `globalOnly`: By default ADF Faces displays global messages (i.e., messages that are not associated with components) followed by individual component messages. If you wish to display only global messages in the box, set this attribute to `true`. Component messages will continue to display with the associated component.
   - `message`: The main message text that displays just below the message box title, above the list of individual messages.
   - `text`: The text that overrides the default title of the message box.

3. Ensure that client-side validation has been disabled. If you do not disable client-side validation, the alert dialog will display if there are any ADF Faces validation errors, as the server-side validation will not have taken place.

   Tip: To disable client-side validation, add the `<client-validation-disabled>` element in `adf-faces-config.xml` and set it to `true`.

12.7.2 What Happens When You Choose to Display Error Messages

When a conversion or validation error occurs on an ADF Faces input component, the component creates a `FacesMessage` object and adds it to a message queue on the `FacesContext` instance. During the Render Response phase, the message associated with the validator or converter is displayed using the built-in message display attribute for the ADF Faces input component. This attribute displays the detail error message next to the component. The message is also displayed by the optional `af:messages` tag, which displays all summary messages in a message box.

12.8 Handling and Displaying Exceptions in an ADF Application

Exceptions thrown by any part of an ADF application are also handled and displayed on the JSF page. By default, all exceptions thrown in the application are caught by the binding container. When an exception is encountered, the binding container routes the exception to the application's active error handler, which by default is the `DCErrorHandlerImpl` class. The `reportException(BindingContainer, Exception)` method on this class passes the exception to the binding container to process. The binding container then processes the exception by placing it on the exception list in a cache.

If exceptions are encountered on the page during the page lifecycle, (for example, during validation), they are also caught by the binding container and cached, and are additionally added to `FacesContext`.

During the Prepare Render phase, the ADF lifecycle executes the `reportErrors(context)` method. This method is implemented differently for each view technology. By default, the `reportErrors` method on the `FacesPageLifecycle` class:

- Accesses the exception list from the binding container.
- Calls the `addError` helper method, which creates and adds the messages to the `FacesContext`. By default, messages display the JBO exception number and exception text.
- Clears the exceptions list in the binding container.

You can customize this default framework behavior. For example, you can create a custom error handler for exceptions, or you can change how the lifecycle reports exceptions. You can also customize how a single page handles exceptions.

12.8.1 How to Change Exception Handling

You can change the default exception handling by extending the default error handler, `DCErrorHandlerImpl`. Doing so also requires that you create a custom ADF lifecycle class that will call the new error handler during the Prepare Model phase.

You can also create a custom ADF lifecycle class to change how the lifecycle reports errors by overriding the `reportErrors` method.

If you only want to change how exceptions are created for a single page, you can create a lifecycle class just for that page.

To create a custom error handler:

1. Create a class that extends the `DCErrorHandlerImpl` class.
2. In the new class, override the `public void reportException(DCBindingContainer, Exception)` method.

   Example 12–11 shows the `SRDemoErrorHandler Class` that the SRDemo application uses to handle errors.

Example 12–11 SRDemoErrorHandler Class

```
public class SRDemoErrorHandler extends DCErrorHandlerImpl{
  /**
   * Constructor for custom error handler.
   *
   * @param setToThrow should exceptions throw or not
   */
  public SRDemoErrorHandler(boolean setToThrow) {
    super(setToThrow);
  }
  public void reportException(DCBindingContainer bc, Exception ex) {
     //Force JboException's reported to the binding layer to avoid
     //printing out the JBO-XXXXX product prefix and code.
    disableAppendCodes(ex);
    super.reportException(bc, ex);
  }
```

```
  private void disableAppendCodes(Exception ex) {
    if (ex instanceof JboException) {
      JboException jboEx = (JboException) ex;
      jboEx.setAppendCodes(false);
      Object[] detailExceptions = jboEx.getDetails();
      if ((detailExceptions != null) && (detailExceptions.length > 0)) {
        for (int z = 0, numEx = detailExceptions.length; z < numEx; z++) {
          disableAppendCodes((Exception) detailExceptions[z]);
        }
      }
    }
  }
}
```

3. Globally override the error handler. To do this, you must create a custom page lifecycle class that extends `FacesPageLifecycle`. In this class, you override the `public void prepareModel(LifecycleContext)` method, which sets the error handler. To have it set the error handler to the custom handler, have the method check whether or not the custom error handler is the current one in the binding context. If it is not, set it to be. (Because by default the `ADFBindingFilter` always sets the error handler to be `DCErrorHandlerImpl`, your method must set it back to the custom error handler.) You must then call `super.prepareModel`.

   Example 12–12 shows the `prepareModel` method from the `frameworkExt.SRDemoPageLifecycle` class that extends the `FacesPageLifecycle` class. Note that the method checks whether or not the error handler is an instance of the `SRDemoErrorHandler`, and if it is not, it sets it to the new error handler.

Example 12–12 PrepareModel Method

```
public void prepareModel(LifecycleContext ctx) {
    if (!(ctx.getBindingContext().getErrorHandler() instanceof
          SRDemoErrorHandler)) {
        ctx.getBindingContext().setErrorHandler(new SRDemoErrorHandler(true));
    }
    super.prepareModel(ctx);
}
```

4. You now must create a new Phase Listener that will return the custom lifecycle. See the procedure "To create a new phase listener:" later in the section.

To customize how the lifecycle reports errors:

1. Create a custom page lifecycle class that extends `FacesPageLifecycle`.
2. Override the `public void reportErrors(PageLifecycleContext)` method to customize the display of error messages.

   Example 12–13 shows the `reportErrors` method and associated methods in the `frameworkExt.SRDemoPageLifecycle` class that extends the `FacesPageLifecycle` class to change how the errors are reported.

Example 12–13 ReportErrors Method in the SRDemoPageLifecycle Class

```
public void reportErrors(PageLifecycleContext ctx) {
    DCBindingContainer bc = (DCBindingContainer)ctx.getBindingContainer();
    if (bc != null) {
        List<Exception> exceptions = bc.getExceptionsList();
        if (exceptions != null) {
            Locale userLocale =
                ctx.getBindingContext().getLocaleContext().getLocale();
            /*
             * Iterate over the top-level exceptions in the exceptions list and
             * call addError() to add each one to the Faces errors list
             * in an appropriate way.
             */
            for (Exception exception: exceptions) {
                try {
                    translateExceptionToFacesErrors(exception, userLocale,
                                                    bc);
                } catch (KnowErrorStopException stop) {
                    FacesContext fctx = FacesContext.getCurrentInstance();
                    fctx.addMessage(null,
                                    JSFUtils.getMessageFromBundle
                                    (stop.getMessage(),
                                     FacesMessage.SEVERITY_FATAL));
                    break;
                }
            }
        }
    }
}

protected void translateExceptionToFacesErrors(Exception ex, Locale locale,
                                               BindingContainer bc) throws
                                          KnowErrorStopException {
    List globalErrors = new ArrayList();
    Map attributeErrors = new HashMap();
    processException(ex, globalErrors, attributeErrors, null, locale);
    int numGlob = globalErrors.size();
    int numAttr = attributeErrors.size();
    if (numGlob > 0) {
        for (int z = 0; z < numGlob; z++) {
            String msg = (String)globalErrors.get(z);
            if (msg != null) {
                JSFUtils.addFacesErrorMessage(msg);
            }
        }
    }
    if (numAttr > 0) {
        Iterator i = attributeErrors.keySet().iterator();
        while (i.hasNext()) {
            String attrNameKey = (String)i.next();
```

```
            /*
             * Only add the error to show to the user if it was related
             * to a field they can see on the screen. We accomplish this
             * by checking whether there is a control binding in the current
             * binding container by the same name as the attribute with
             * the related exception that was reported.
             */
            ControlBinding cb =
                ADFUtils.findControlBinding(bc, attrNameKey);
            if (cb != null) {
                String msg = (String)attributeErrors.get(attrNameKey);
                if (cb instanceof JUCtrlAttrsBinding) {
                    attrNameKey = ((JUCtrlAttrsBinding)cb).getLabel();
                }
                JSFUtils.addFacesErrorMessage(attrNameKey, msg);
            }
        }
    }
}

/**
* Populate the list of global errors and attribute errors by
* processing the exception passed in, and recursively processing
* the detail exceptions wrapped inside of any oracle.jbo.JboException
* instances.
*
* If the error is an attribute-level validation error, we can tell
* because it should be an instanceof oracle.jbo.AttrValException
* For each attribute-level error, we retrieve the name of the attribute
* in error by calling an appropriate getter method on the exception
* object which exposes this information to us. Since attribute-level
* errors could be wrapping other more specific attribute-level errors
* that were the real cause (especially due to Bundled Exception Mode).
* We continue to recurse the detail exceptions and we only consider
* relevant to report the exception that is the most deeply nested, since
* it will have the most specific error message for the user. If multiple
* exceptions are reported for the same attribute, we simplify the error
* reporting by only reporting the first one and ignoring the others.
* An example of this might be that the user has provided a key value
* that is a duplicate of an existing value, but also since the attribute
* set failed due to that reason, a subsequent check for mandatory attribute
* ALSO raised an error about the attribute's still being null.
*
* If it's not an attribute-level error, we consider it a global error
* and report each one.
*
* @param ex the exception to be analyzed
* @param globalErrs list of global errors to populate
* @param attrErrs map of attrib-level errors to populate, keyed by attr name
* @param attrName attribute name of wrapping exception (if any)
* @param locale the user's preferred locale as determined by the browser
*/
private void processException(Exception ex, List globalErrs, Map attrErrs,
                              String attrName,
                              Locale locale) throws KnowErrorStopException {
    /*
     * Process the exceptions
     * Start with some special cases that are known bad situations where we
     * need to format some useful advice rather than just parroting the
     * exception text
```

```
 */
if (ex instanceof EJBException) {
    String msg = ex.getLocalizedMessage();
    if (msg == null) {
        msg = firstLineOfStackTrace(ex, true);
    }
    Exception causeEx = ((EJBException)ex).getCausedByException();
    if (causeEx instanceof TopLinkException) {
        int toplinkErrorCode =
            ((TopLinkException)causeEx).getErrorCode();
        switch (toplinkErrorCode) {
        case 7060:
            {
                throw new KnowErrorStopException("srdemo.topLinkError.7060");
            }
        case 4002:
            {
                throw new KnowErrorStopException("srdemo.topLinkError.4002");
            }
        }
    }
    globalErrs.add(msg);
} else if (ex instanceof AdapterException){
    AdapterException causeEx = ((AdapterException)ex);

    int err = Integer.parseInt( causeEx.getErrorCode());
    switch (err){
        case 40010:{
            throw new KnowErrorStopException("srdemo.urlDCError.40010");
        }
        case 29000:{
            throw new KnowErrorStopException("srdemo.urlDCError.29000");
        }
        default:{
            throw new KnowErrorStopException("srdemo.urlDCError.other");
        }
    }

} else if (!(ex instanceof JboException)) {
    String msg = ex.getLocalizedMessage();
    if (msg == null) {
        msg = firstLineOfStackTrace(ex, true);
    }
    globalErrs.add(msg);
    /*
   * If this was an unexpected error, print out stack trace
   */
    reportUnexpectedException(ex);
    return;
}
if (ex instanceof AttrValException) {
    AttrValException ave = (AttrValException)ex;
    attrName = ave.getAttrName();
    Object obj = attrErrs.get(attrName);
```

```
            /*
             * If we haven't already recorded an error for this attribute
             * and if it's a leaf detail, then log it as the first error for
             * this attribute. If there are details, then we'll recurse
             * into the details below. But, in the meantime we've recorded
             * What attribute had the validation error in the attrName
             */
            Object[] details = ave.getDetails();
            if (obj != null) {
                /*
                 * We've already logged an attribute-validation error for this
                 * attribute, so ignore subsequent attribute-level errors
                 * for the same attribute. Note that this is not ignoring
                 * NESTED errors of an attribute-level exception, just the
                 * second and subsequent PEER errors of the first attribute-level
                 * error. This means the user might receive errors on several
                 * different attributes, but for each attribute we're choosing
                 * to tell them about just the first problem to fix.
                 */
                return;
            } else {
                /*
                 * If there aren't any further, nested details, then log first error
                 */
                if ((details == null) || (details.length == 0)) {
                    attrErrs.put(attrName, ave.getLocalizedMessage(locale));
                }
            }
        }
        JboException jboex = (JboException)ex;
        /*
         * It is a JboException so recurse into the exception tree
         */
        Object[] details = jboex.getDetails();
        /*
         * We only want to report Errors for the "leaf" exceptions
         * So if there are details, then don't add an errors to the lists
         */
        if ((details != null) && (details.length > 0)) {
            for (int j = 0, count = details.length; j < count; j++) {
                processException((Exception)details[j], globalErrs, attrErrs,
                                 attrName, locale);
            }
        } else {
            /*
             * Add a new Error to the collection
             */
            if (attrName == null) {
                String errorCode = jboex.getErrorCode();
                globalErrs.add(jboex.getLocalizedMessage(locale));
            } else {
                attrErrs.put(attrName, jboex.getLocalizedMessage(locale));
            }
            if (!(jboex instanceof ValidationException)) {
                reportUnexpectedException(jboex);
            }
        }
    }
```

```
/**
* Prints the stack trace for an unexpected exception to standard out.
*
* @param ex The unexpected exception to report.
*/
protected void reportUnexpectedException(Exception ex) {
    ex.printStackTrace();
}
/**
* Picks off the exception name and the first line of information
* from the stack trace about where the exception occurred and
* returns that as a single string.
*/
private String firstLineOfStackTrace(Exception ex, boolean logToError) {
   if (logToError) {
        ex.printStackTrace(System.err);
    }
    StringWriter sw = new StringWriter();
    PrintWriter pw = new PrintWriter(sw);
    ex.printStackTrace(pw);
    LineNumberReader lnr =
        new LineNumberReader(new StringReader(sw.toString()));
    try {
        String lineOne = lnr.readLine();
        String lineTwo = lnr.readLine();
        return lineOne + " " + lineTwo;
    } catch (IOException e) {
        return null;
    }
}
```

3. You now must create a new phase listener that will return the custom lifecycle.

To create a new phase listener:

1. Extend the `ADFPhaseListener` class.
2. Override the `protected PageLifecycle createPageLifecycle ()` method to return a new custom lifecycle.

   Example 12–14 shows the `createPageLifecycle` method in the `frameworkExt.SRDemoADFPhaseListener` class.

Example 12–14 CreatePageLifecycle Method in SRDemoADFPhaseListener

```
public class SRDemoADFPhaseListener extends ADFPhaseListener {
  protected PageLifecycle createPageLifecycle() {
    return new SRDemoPageLifecycle();
  }
}
```

3. Register the phase listener in the `faces-config.xml` file.
   - Open the `faces-config.xml` file and select the **Overview** tab in the editor window. The `faces-config.xml` file is located in the `<View_Project>/WEB-INF` directory.
   - In the window, select **Life Cycle** and click **New**. Click **Help** or press F1 for additional help in registering the converter.

To override exception handling for a single page:

1. Create a custom page lifecycle class that extends the `FacesPageLifecycle` class.
2. Override the `public void reportErrors(PageLifecycleContext)` method to customize the display of error messages. For an example of overriding this method, see the procedure "To customize how the lifecycle reports errors:" earlier in this section.
3. Open the page definition for the page. In the Structure window, select the page definition node. In the Property Inspector, enter the new class as the value for the `ControllerClass` attribute.

12.8.2 What Happens When You Change the Default Error Handling

When you create your own error handler, the application uses that class instead of the `DCErrorHandler` class. Because you created and registered a new lifecycle, that lifecycle is used for the application. This new lifecycle instantiates your custom error handler.

When an error is subsequently encountered, the binding container routes the error to the custom error handler. The `reportException(BindingContainer, Exception)` method then executes.

If you've overridden the `reportErrors` method in the custom lifecycle class, then during the Prepare Render phase, the lifecycle executes the new `reportErrors(context)` method.

13

Adding ADF Bindings to Existing Pages

This chapter describes how to use the Data Control Palette to add ADF bindings to existing UI components. Instead of using the Data Control Palette to design your application pages, you can design the UI first using other tools, such as the Component Palette, and add the ADF bindings later.

This chapter includes the following sections:

- Section 13.1, "Introduction to Adding ADF Bindings to Existing Pages"
- Section 13.2, "Designing Pages for ADF Bindings"
- Section 13.3, "Using the Data Control Palette to Bind Existing Components"
- Section 13.4, "Adding ADF Bindings to Text Fields"
- Section 13.5, "Adding ADF Bindings to Tables"
- Section 13.6, "Adding ADF Bindings to Actions"
- Section 13.7, "Adding ADF Bindings to Selection Lists"
- Section 13.8, "Adding ADF Bindings to Trees and Tree Tables"

13.1 Introduction to Adding ADF Bindings to Existing Pages

While the Data Control Palette enables you to design and create bound components in a single drag and drop action, in some cases, it may be preferable to create the basic UI components first and add the bindings later. For example, if a development team includes UI designers, the designers can create the basic pages using JDeveloper tools, such as the Component Palette, and the developers can add the page functionality afterwards, including the ADF bindings.

Read this chapter to understand:

- How to design a page for easy insertion of ADF bindings
- Which UI components you can add ADF bindings to
- How to use the Data Control Palette to add ADF bindings to existing page components
- What happens to your components when ADF bindings are added

13.2 Designing Pages for ADF Bindings

When designing and creating a web page that will have ADF bindings added later, use the JDeveloper wizards, visual editors, and design tools (such as the Component Palette).

You can design your pages using any tags that you want; however, if you plan to add ADF bindings to certain components, you may want to design those components using tags that work with ADF bindings. Otherwise, the components will have to be entirely replaced when the bindings are added later.

When you add ADF bindings to an existing component, ADF preserves as much of the original component's properties as possible. However, the binding may overwrite such things as labels, column headings, and range navigation.

If a label value contains a static text string, the ADF binding overwrites the value with an EL expression that binds to an attribute name in the data control. However, if you define your UI labels using EL expressions that reference managed beans (for example, a standard binding on a resource bundle), that label is preserved when you add the ADF binding to that component. In many cases, it is preferable to design your basic UI components using labels that are bound to resource bundles, especially if you will be localizing your pages. For more information about resource bundles, see Section 14.4, "Internationalizing Your Application".

Range navigation is another property that is overwritten by the ADF binding, because the iterator referenced by the binding manages the current rowset. Later sections in this chapter discuss how to add ADF bindings to specific UI components and how those specific components are affected by the ADF bindings.

13.2.1 Creating the Page

When you use the Create JSF JSP wizard to create a page to which you intend to add ADF bindings, be sure to do the following actions to make future binding easier:

- Choose the **Do not Automatically Expose UI Components in a Managed Bean** option.

  This option turns off JDeveloper's auto-binding feature, which automatically associates every UI component in the page to a corresponding property in the backing bean for eventual programmatic manipulation. If you intend to add ADF bindings to a page, Oracle recommends that you do not use the auto-binding feature. If you use the auto-binding feature, you will have to remove the managed bean bindings later, after you have added the ADF bindings. The managed bean UI component property bindings do not affect the ADF bindings, but their presence may be confusing in the JSF code. For information about managed beans, see Section 4.5, "Creating and Using a Backing Bean for a Web Page".

- Add the ADF Faces tag libraries.

  While you can add ADF bindings to JSF components, the ADF Faces components provide greater functionality, especially when combined with ADF bindings.

- Add the desired page-level physical attributes such as background color, style sheets, or skins.

  The ADF bindings do not affect your page-level attributes. For information about using ADF Faces skins, see Section 14.3, "Using Skins to Change the Look and Feel".

13.2.2 Adding Components to the Page

When designing web pages, keep in mind that ADF bindings can be added only to certain ADF Faces tags or their equivalent JSF HTML tags. Table 13–1 lists the ADF Faces and JSF tags to which you can later add ADF bindings. On the Component Palette, the ADF Faces tags are available on the ADF Faces Core page, and the JSF tags are available on the JSF HTML page.

> **Tip:** To enable the use of JSF Reference Implementation UI component tags with ADF bindings, you must choose the **Include JSF HTML Widgets for JSF Databinding** option in the **ADF View Settings** of the project properties. However, using ADF Faces tags, especially with ADF bindings, provides greater functionality than does using the reference implementation JSF tags.

Table 13–1 Tags That Can Be Used for ADF Bindings

ADF Faces Tags Used in ADF Bindings	Equivalent JSF HTML Tags
Text Fields	
af:inputText	h:inputText
af:outputText	h:outputText
af:outputLabel	h:outputLabel
Tables	
af:table	h:dataTable
Actions	
af:commandButton	h:commandButton
af:commandLink	h:commandLink
Selection Lists	
af:selectOneChoice	h:selectOneMenu
af:selectOneListbox	h:selectOneListbox
af:selecOneRadio	h:selectOneRadio
af:selectBooleanRadio	h:selectBooleanCheckbox
Trees	
af:tree	n/a
af:treeTable	n/a

13.2.3 Other Design Considerations

When designing pages using the JDeveloper wizards and editors to which you will later add ADF bindings, you can either:

- Choose options that enable you to bind later and, instead, enter static labels and values. This approach enables you to design your UI using placeholder labels and values that will be replaced later by the values and labels returned by the ADF bindings.

  OR

- Bind labels to resource bundles, which contain the actual text to be displayed in the label. When you later add an ADF binding to a component, ADF retains any existing label bindings on resource bundles (or managed beans). For information about using resource bundles, see Section 14.4, "Internationalizing Your Application".

For information about creating JSF and ADF Faces components, see Section 4.4.1, "How to Add UI Components to a JSF Page".

13.2.3.1 Creating Text Fields in Forms

For text field labels, you can either enter static placeholder values or bind to resource bundles. If you are not binding labels to resource bundles, then use the Property Inspector or source editor to add or modify placeholder labels and values in text fields. Use placeholder labels and values that make it easier for the developer, who will later add the bindings, to determine the intent of the field. Static placeholder values will be replaced by the ADF bindings. However, as mentioned previously, any bindings to resource bundles will be retained.

For example, if you are creating a form that displays user information, you might use `User First Name`, `User Last Name`, and `User Address` as placeholder text field labels. The developer who adds the ADF bindings would then match the placeholder labels to actual attributes in a data source on the data control.

13.2.3.2 Creating Tables

When you drag a table component from the Component Palette and drop it on a page, JDeveloper displays a table wizard to help you define the table. Choose the **Bind Later** option in the ADF Faces Table wizard (or, for JSF tables, the **Number of Columns** option in the Create Data Table wizard), which enables you to specify the number of columns needed in the table instead of binding to a data source. If you are unsure of the total number of columns needed, enter an estimate. Later, when the bindings are added, the number of columns can easily be adjusted.

As with text fields, use placeholder labels or bindings on resource bundles in the column headings. If you are using the ADF table component, you can specify the column headings in the **Header Text** field on the Column Details page of the ADF Faces Table wizard. For JSF tables, you can enter the column headings directly in the table displayed in the visual editor.

13.2.3.3 Creating Buttons and Links

For the button or link label, use the Property Inspector or the source editor to add a static placeholder or a binding on a resource bundle. If the button or link will perform page navigation, you can specify an outcome value in the `action` attribute, to enable page navigation in your initial pages. However, when the ADF bindings are added, the `action` attribute is overwritten, and the action will have to be re-entered.

13.2.3.4 Creating Lists

When you drag a selection list from the Component Palette and drop it on a page, JDeveloper displays the Insert dialog to help you define the list. Use the **Create List** option on the Insert dialog to define the list. Only enter item labels or values if you will ultimately create a static list. If you intend to populate the list from a binding on a data collection, leave the item labels and values blank. For the list label, use the Property Inspector to enter a static placeholder or a binding on a resource bundle. For example, if you are creating a dropdown list of products, you might enter `Products` as the label for the list. Later, when the binding is added, static placeholder labels are replaced by an ADF binding expression.

13.2.3.5 Creating Trees or Tree Tables

When creating trees, use the `value` attribute to identify the root node and the `var` value to identify the branch node. When creating a tree table, choose the **Bind Later** option in the ADF Faces Tree Table wizard. You can specify a number of columns, but when the ADF binding is added all data is displayed in a single column.

13.3 Using the Data Control Palette to Bind Existing Components

To bind existing components to ADF data controls, you must add ADF binding expressions to the component tags. While you could manually add ADF binding expressions to existing tags, it is easier to use the Data Control Palette. Using the Data Control Palette ensures that all the necessary binding objects and references are automatically created for you. (For more information see, Section 13.3.2, "What Happens When You Use the Data Control Palette to Add ADF Bindings".)

13.3.1 How to Add ADF Bindings Using the Data Control Palette

The following procedure is a general description of how to use the Data Control Palette and the Structure window to add ADF bindings to existing components. Later sections in this chapter describe how to add ADF bindings to specific types of components.

To add ADF bindings using the Data Control Palette and Structure Window:

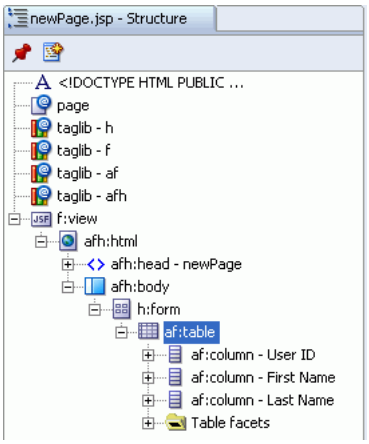
1. With your page displayed in the Design page of the visual editor, open the Structure window.

   Tip: You can drop the data control object on the component displayed in the **Design** page of the visual editor, but using the Structure window provides greater accuracy and precision. For example, if you try dropping a data control object on a component in the visual editor and do not get the **Bind Existing <*component name*>** option in the context menu, this means you did not drop the data control on the correct tag in the visual editor. In this case, try using the Structure window where each tag is clearly delineated.

2. In the Design page of the visual editor, select the UI component to which you want to add ADF bindings.

   The component must be one of the tags listed in Table 13–1. When you select a component in the visual editor, JDeveloper simultaneously selects that component tag in the Structure window, as shown in Figure 13–1. Use the Structure window to verify that you have selected the correct component. If the incorrect component is selected, make the adjustment in the Structure window.

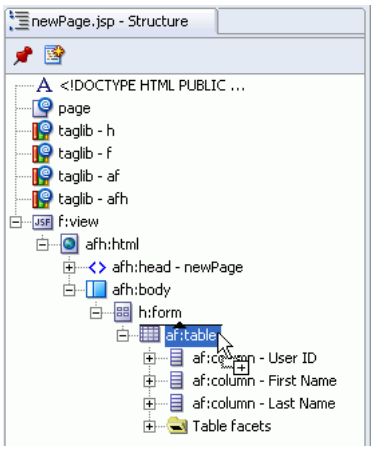
Figure 13–1 Structure Window with Tag Selected



3. Drag the appropriate data control object from the Data Control Palette to the Structure window and drop it on the selected UI component. (For information about the nodes on the Data Control Palette, see Section 5.2.1, "How to Understand the Items on the Data Control Palette".)

   Tip: As you position the data control object over the UI component in the Structure window, a horizontal line with an embedded up or down arrow appears at the top or bottom of the component, as shown in Figure 13–2. Whenever either of these lines appears, you can drop the data control object: in this case, it does not matter which direction the arrow is pointing.

Figure 13–2 Dropping a Data Control Object on a UI Component in the Structure Window



4. From the Data Control Palette context menu, choose the **Bind Existing *<component name>*** option, where *<component name>* is the name of the component, such as text field or table, as shown in Figure 13–3.

   Tip: If the context menu does not display a **Bind Existing *<component name>*** option, you have not dropped the data control object on the correct tag in the Structure window. You can add bindings only to the tags shown in Table 13–1.

Figure 13–3 Context Menu for Binding to an Existing Component

13.3.2 What Happens When You Use the Data Control Palette to Add ADF Bindings

When you use the Data Control Palette all of the required ADF objects are automatically created for you:

- The `DataBindings.cpx` file is created and a corresponding entry for the page is added to it.
- The ADF binding filter is registered in the `web.xml` file.
- The ADF phase listener is registered in the `faces-config.xml` file.
- A page definition file is created and configured with the binding object definitions for component on the page.

All of these objects are required for a component with ADF bindings to be rendered correctly on a page. If you do not use the Data Control Palette, you will have to create these things manually. For more information about these objects, see Chapter 5, "Displaying Data on a Page".

13.4 Adding ADF Bindings to Text Fields

You bind forms or other container components by binding the individual text fields that comprise the component: you cannot bind an entire form at one time. You bind a text field to an attribute in a collection.

13.4.1 How to Add ADF Bindings to Text Fields

To add ADF bindings to a text field, you drag an attribute from the Data Control Palette and drop it on the text field component displayed in the Structure window. For general tips about dropping items from the Data Control Palette onto the Structure window, see Section 13.3.1, "How to Add ADF Bindings Using the Data Control Palette".

To add ADF bindings to a text field:

1. With the page displayed in the Design page of the visual editor, open the Structure window.
2. In Design page of the visual editor, select the text field.

   This simultaneously selects the tag in the Structure window. The text field tag must be one of the tags listed previously in Table 13–1. If the incorrect tag is selected, make the adjustment in the Structure window.
3. From the Data Control Palette, drag an attribute to the Structure window and drop it on the selected text field.

4. On the Data Control Palette context menu, choose **Bind Existing Input Text**.

   The binding is added to the text field.

13.4.2 What Happens When You Add ADF Bindings to a Text Field

Example 13–1 displays an input text field component *before* the ADF bindings are added. The example is a simple `inputText` tag with a static label value of `First Name`.

Example 13–1 Text Field Component Before ADF Bindings

```
<af:inputText label="First Name"/>
```

Example 13–2 displays the same text field *after* the `firstName` attribute of the `User` data collection from the SRDemo data control was dropped on it. The `User` collection is returned by the `findAllStaff` method. Notice that the label was replaced with a binding expression. To modify the label displayed by an ADF binding, you can use control hints. Other tag attributes have been added with bindings on different properties on the `FirstName` attribute. For a description of each binding property, see Appendix B, "Reference ADF Binding Properties".

Example 13–2 Text Field Component After ADF Bindings Are Added

```
<af:inputText label="#{bindings.FirstName.label}"
              value="#{bindings.FirstName.inputValue}"
              required="#{bindings.FirstName.mandatory}"
              columns="#{bindings.FirstName.displayWidth}">
   <af:validator binding="#{bindings.FirstName.validator}"/>
</af:inputText>
```

In addition to adding the bindings to the text field, JDeveloper automatically adds entries for the databound text field to the page definition file. The page definition entries include an iterator binding object defined in the `executables` element and a a value binding defined in the `bindings` element. For more information about databound text fields and forms, see Chapter 6, "Creating a Basic Page".

13.5 Adding ADF Bindings to Tables

You can add ADF bindings to an entire table at one time. In fact, it is recommended to bind the entire table instead of the individual components that comprise the table. When you add a binding to a table, you can drag an entire collection from the Data Control Palette onto the table. You can bind an individual column, but only if the table is already bound to an iterator.

13.5.1 How to Add ADF Bindings to Tables

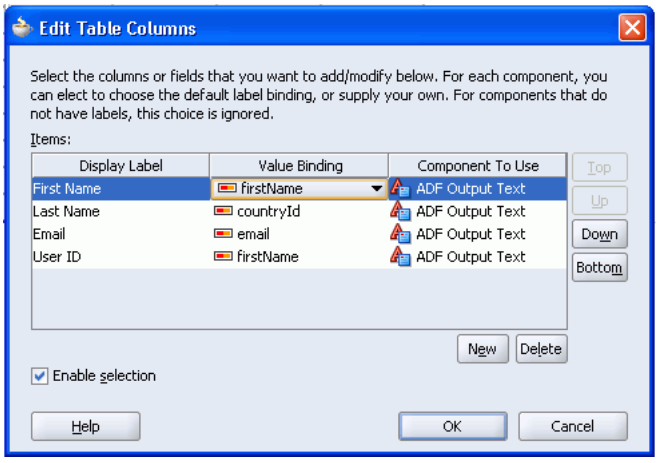
To add ADF bindings to a table, you drag a data collection from the Data Control Palette and drop it on the table tag displayed in the Structure window. For general tips about dropping items from the Data Control Palette onto the Structure window, see Section 13.3.1, "How to Add ADF Bindings Using the Data Control Palette".

To add ADF bindings to a table:

1. With the page displayed in the Design page of the visual editor, open the Structure window.
2. In the Design page of the visual editor, select the table.

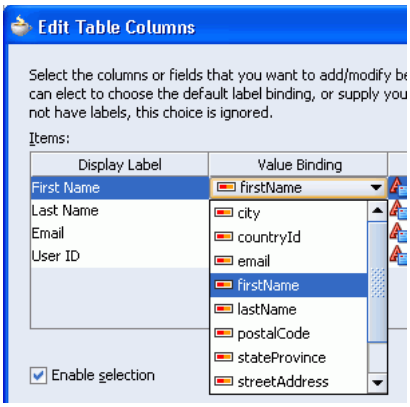
   The tag selected in the Structure window must be one of the tags listed previously in Table 13–1. JDeveloper simultaneously selects the corresponding tag in the Structure window. If the incorrect tag is selected, make the adjustment in the Structure window. For example, if a column tag is selected, select the table tag instead.
3. From the Data Control Palette, drag a collection to the Structure window and drop it on the selected table tag.
4. On the Data Control Palette context menu, choose **Bind Existing ADF Table** or **Bind Existing ADF Read-only Table**. The Edit Table Column dialog appears, as shown in Figure 13–4.

Figure 13–4 Edit Table Column Dialog



   The **Display Label** column in the dialog displays the placeholder column headings entered when the table was created. In the example, the placeholder column headings are **First Name**, **Last Name**, **Email**, and **User ID**. The **Value Binding** column displays the attributes from the data collection. The **Component to Use** column displays the types of components each table column will contain.
5. In the Edit Table Columns dialog, use the dropdowns in the **Value Binding** fields to choose the attributes from the data collection to be bound to each column in the table, as shown in Figure 13–5. If placeholder column headings were entered when the table was created, match the attributes to the appropriate column headings. For example, if a column heading is **First Name**, you would choose the `firstName` attribute from the **Value Binding** dropdown next to that column heading.

Figure 13–5 Value Binding Dropdown in the Edit Table Columns Dialog



Tip: If you need to add additional columns to the table, click **New**.

For more information about tables, see Chapter 7, "Adding Tables".

13.5.2 What Happens When You Add ADF Bindings to a Table

Example 13–3 displays a table *before* the ADF bindings are added. The table defines four columns and uses static placeholder values as column headings: `First Name`, `Last Name`, `Email`, and `User ID`. The table also defines a range navigation of 15 rows, table banding, and a selection facet.

Example 13–3 ADF Faces Table Before ADF Bindings

```
<af:table emptyText="No items were found" rows="15" banding="none"
          bandingInterval="1">
     <f:facet name="selection">
       <af:tableSelectOne/>
     </f:facet>
     <af:column sortable="false" headerText="First Name">
       <af:outputText value="#{row.col1}"/>
     </af:column>
     <af:column sortable="false" headerText="Last Name">
       <af:outputText value="#{row.col2}"/>
     </af:column>
</af:table>
```

Example 13–4 displays the same table *after* the `User` data collection returned by the `findAllStaff` method from the SRDemo data control was dropped on it. Notice that since the placeholder column headings were static values, they have been replaced with a binding on the `findAllStaff1` iterator. However, the selection facet and banding from the original table remain intact. The `selectionState` and `selectionListener` attributes have been added with bindings on the binding object.

The range navigation value is replaced by a binding on the iterator, which manages the current row. The `rangeSize` binding property, which defines the number of rows can be set in the page definition file. For a description of each binding property, see Appendix B, "Reference ADF Binding Properties".

Example 13–4 ADF Faces Table After ADF Bindings Are Added

```
<af:table emptyText="#{bindings.findAllStaff1.viewable ? 'No rows yet.' : 'Access
                       Denied.'}"
          rows="#{bindings.findAllStaff1.rangeSize}" banding="none"
          bandingInterval="1"
          value="#{bindings.findAllStaff1.collectionModel}" var="row"
          first="#{bindings.findAllStaff1.rangeStart}"
          selectionState="#{bindings.findAllStaff1.collectionModel.
                            selectedRow}"
          selectionListener="#{bindings.findAllStaff1.collectionModel.
                               makeCurrent}">
     <f:facet name="selection">
       <af:tableSelectOne/>
     </f:facet>
     <af:column sortable="false"
                headerText="#{bindings.findAllStaff1.labels.firstName}"
                sortProperty="firstName">
        <af:outputText value="#{row.firstName}"/>
     </af:column>
     <af:column sortable="false"
                headerText="#{bindings.findAllStaff1.labels.lastName}"
                sortProperty="lastName">
        <af:outputText value="#{row.lastName}"/>
     </af:column>
</af:table>
```

In addition to adding the bindings to the table, JDeveloper automatically adds entries for the databound table to the page definition file. The page definition entries include an iterator binding object defined in the `executables` element and the value bindings for the table in the `bindings` element. By default, the `RangeSize` property on the iterator binding is set to 10. This value is now bound to the range navigation in the table and overrides the original range navigation value set in the table before the bindings were added. In the example, the original table set the range navigation value at 15. If necessary, you can change the `RangeSize` value in the page definition to match the original value defined in the table.

The `bindings` element contains a `methodAction`, which encapsulates information about how to invoke the method iterator, and value bindings for the attributes available to the table. The value bindings include all the attributes of the returned collection, even if the table itself is displaying only a subset of those attributes.

13.6 Adding ADF Bindings to Actions

You can add ADF bindings to buttons or links. When you add a binding to a button or link, you use a method or operation from the data control. When a user clicks the button or link, the method or operation is invoked.

If you want the button or link to perform page navigation, after adding the ADF binding you must bind the `action` attribute of the component tag to a backing bean, which will handle the navigation. The backing bean must inject the ADF binding container and return an outcome value. For information about creating navigation rules and binding navigation components to backing beans, see Chapter 9, "Adding Page Navigation".

13.6.1 How to Add ADF Bindings to Actions

To add ADF bindings to a button or link, you drag a method or operation from the Data Control Palette and drop it on the button or link tag displayed in the Structure window. For general tips about dropping items from the Data Control Palette onto the Structure window, see Section 13.3.1, "How to Add ADF Bindings Using the Data Control Palette".

To add ADF bindings to a button or link:

1. With the page displayed in the Design page of the visual editor, open the Structure window.
2. In the Design page of the visual editor, select the button or link.

   The tag selected in the Structure window must be one of the tags listed previously in Table 13–1. JDeveloper simultaneously selects the corresponding tag in the Structure window. If the incorrect tag is selected, make the adjustment in the Structure window.
3. From the Data Control Palette, drag a method or operation to the Structure window and drop it on the selected button or link tag.
4. On the Data Control Palette context menu, choose **Bind Existing CommandButton** or **Bind Existing CommandLink**.
5. If the method requires a parameter, the Action Binding Editor appears where you define the parameter values to pass to the method. (For more information about passing parameters to methods, see Chapter 10, "Creating More Complex Pages".)

13.6.2 What Happens When You Add ADF Bindings to an Action

Example 13–5 displays a command button *before* the ADF bindings are added.

Example 13–5 ADF Faces Command Button Before ADF Bindings

```
<af:commandButton text="Display User"/>
```

Example 13–6 displays the same button after the `findAllStaff()` method from the SRDemo data control was dropped on it. The `findAllStaff` method returns the `User` collection. Since the original label was a static value, the binding replaced it with the name of the method; you can change the button label using the Property Inspector. An `actionListener` attribute was added with a binding on the `findAllStaff` method. The `actionListener` detects when the user clicks the button and executes the method as a result. If you want the button to navigate to another page, you can bind to a backing bean or add an `action` value. For more information, see Chapter 9, "Adding Page Navigation".

Example 13–6 ADF Faces Command Button After ADF Bindings Are Added

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

In addition to adding the bindings to the button, JDeveloper automatically adds a `methodAction` binding object to the page definition file.

For more information about databound buttons and links, see Chapter 6, "Creating a Basic Page".

13.7 Adding ADF Bindings to Selection Lists

You can add ADF bindings to any of the selection lists previously shown in Table 13–1. A databound selection list displays values from a data control collection or a static list and updates an attribute in another collection or a method parameter based on the user's selection. When adding a binding to a list, you use an attribute from the data control that will be populated by the selected value in the list.

13.7.1 How to Add ADF Bindings to Selection Lists

To add ADF bindings to a selection list, you drag an attribute from the Data Control Palette and drop it on the selection list tag displayed in the Structure window. For general tips about dropping items from the Data Control Palette onto the Structure window, see Section 13.3.1, "How to Add ADF Bindings Using the Data Control Palette".

To add ADF bindings to a selection list component:

1. With the page displayed in the Design page of the visual editor, open the Structure window.
2. In the Design page of the visual editor, select the selection list component.

   The tag selected in the Structure window must be one of the tags listed previously in Table 13–1. JDeveloper simultaneously selects the corresponding tag in the Structure window. If the incorrect tag is selected, make the adjustment in the Structure window.
3. From the Data Control Palette, drag an attribute to the Structure window and drop it on the selected selection list tag. Use the attribute in the data collection that you want to populate when the user selects an item from the list.
4. On the Data Control Palette context menu, choose **Bind Existing <*component name*>**.
5. In the List Binding Editor, define the data collection that will be updated by the list (**Base Data Source**), the data collection that will populate the list (**List Data Source**), and the attributes that will be displayed in the list. For information about using the List Binding Editor to define lists, see Section 11.7, "Creating Databound Dropdown Lists".

13.7.2 What Happens When You Add ADF Bindings to a Selection List

Example 13–7 displays a single-selection dropdown list *before* the ADF bindings are added. Notice that the component defines a label for the list, but that it does not define static list item labels and values. The item labels and values will be populated by the bindings.

Example 13–7 ADF Faces Single-Selection Dropdown Before ADF Bindings

```
<af:selectOneChoice label="Product:"/>
```

Example 13–8 displays the same list after the `prodID` attribute in the `Product` collection from the SRDemo data control was dropped on it. Because the original list label was a static value, the binding replaced it with a binding on the `ProductprodId` attribute, which was the attribute that was dragged from the Data Control Palette and dropped on the dropdown list component.

You can change the label using control hints. The list values are also bound to the same attribute. Notice that no display values or labels are defined in the component by the binding. Instead, the display values are defined in the page definition file.

> **Tip:** Any static item labels and values defined in the original selection list are not replaced by the ADF bindings. If you add static item labels and values to the original selection list, and then add a dynamic list with a binding on the data collection, the list will display both the values populated by the binding and the static values defined in the component itself. In most cases, you would not want this. Therefore, you must either design the initial component without using static item labels and values, or remove them after the bindings are added.

Example 13–8 ADF Faces Single-Selection Dropdown After ADF Bindings Are Added

```
<af:selectOneChoice label="#{bindings.ProductprodId.label}"
                    value="#{bindings.ProductprodId.inputValue}">
    <f:selectItems value="#{bindings.ProductprodId.items}"/>
</af:selectOneChoice>
```

In addition to adding the bindings to the list, JDeveloper automatically adds several binding objects for the list to the page definition file. The `executables` element defines the iterator binding for the collection that populates the list, and the iterator binding for the target collection.

The `bindings` element contains two method action binding objects, which encapsulate the information needed to invoke the methods that populate the list and update the data collection. The list binding includes a `ListDisplayAttrNames` element, which defines the data collection attributes that populate the values the user sees in the list. This element is added only if the list is a dynamic list, meaning that the list items are populated by a binding on the data collection. If the list is a static list, a `ValueList` element is added instead with the static values that will appear in the list.

For more information about databound lists, see Section 11.7, "Creating Databound Dropdown Lists".

13.8 Adding ADF Bindings to Trees and Tree Tables

You can add ADF bindings to ADF Faces tree and tree table components. The ADF Faces `tree` component displays a hierarchy of master-detail related data collections in a tree format. A databound ADF Faces tree displays multiple root nodes that are populated by a binding on a master data collection. Each node in the tree may have any number of branches, which are populated by bindings on detail data collections. Each node in the tree is indented to show its level in the hierarchy. The ADF tree component includes mechanisms for expanding and collapsing the tree. 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. The ADF Faces tree table components display a hierarchy of master-detail collections in a table. For more information about master-detail relationships and trees, see Chapter 8, "Displaying Master-Detail Data".

13.8.1 How to Add ADF Bindings to Trees and Tree Tables

To add ADF bindings to a tree or tree table, you drag a master data collection from the Data Control Palette and drop it on the tree tag displayed in the Structure window. For general tips about dropping items from the Data Control Palette onto the Structure window, see Section 13.3.1, "How to Add ADF Bindings Using the Data Control Palette".

To add ADF bindings to a tree component:

1. With the page displayed in the Design page of the visual editor, open the Structure window.
2. In the Design page of the visual editor, select the `tree` tag.

   JDeveloper simultaneously selects the corresponding tag in the Structure window. If the incorrect tag is selected, make the adjustment in the Structure window.
3. From the Data Control Palette, drag a data collection to the Structure window and drop it on the selected tree tag. The data collection you select should be the master collection, which will populate the root node of the tree.
4. On the Data Control Palette context menu, choose **Bind Existing Tree**.
5. Use the Tree Binding Editor to define the root and branch nodes of the tree. For information, see Section 8.4, "Using Trees to Display Master-Detail Objects".

13.8.2 What Happens When You Add ADF Bindings to a Tree or Tree Table

Example 13–9 displays a tree *before* the ADF bindings are added. Notice that the `value` attribute specifies the root node as `users`, and the `var` attribute specifies the first branch as `service requests`.

Example 13–9 ADF Faces Tree Before ADF Bindings

```
<af:tree value="users" var="service requests">
   <f:facet name="nodeStamp">
      <h:outputText/>
   </f:facet>
</af:tree>
```

Example 13–10 displays the same tree after the `User` data collection from the SRDemo data control was dropped on it. The `User` collection is returned by the `findAllStaff` method. The `User` data collection will populate the root node, and the `serviceRequests` collection was defined as a branch off the root nodes. The binding replaced the `value` attribute with a binding on the `findAllStaff1` binding object. The `var` attribute now contains a value of *`node`*, which provides access to the current node. The nodes themselves are defined in the page definition file.

Example 13–10 ADF Faces Tree After ADF Bindings Are Added

```
<af:tree value="#{bindings.findAllStaff1.treeModel}" var="node">
   <f:facet name="nodeStamp">
      <af:outputText value="#{node}"/>
   </f:facet>
</af:tree>
```

In addition to adding the bindings to the tree, JDeveloper automatically adds several binding objects for the tree to the page definition file. The `executables` element defines the iterator binding for the collection that populates the root node.

The `bindings` element contains a `methodAction` binding object, which encapsulates the information needed to invoke the method that populates the root node. In the value bindings, the tree is bound to the `findAllStaffIter` iterator. Each attribute returned by the iterator is listed in the `AttrNames` element, but only the attributes in the `nodeDefinition` element are displayed in the tree. The `Accesssors` element defines the accessor methods that will be used to retrieve the data that will populate the branches in the node. In the example, the `User` node, which is the root node, defines `serviceRequestCollectionAssignedTo` as the accessor method. This method returns the service requests for each user node.

For more information about trees and tree tables, see Chapter 8, "Displaying Master-Detail Data".

14

Changing the Appearance of Your Application

This chapter describes how to change the default appearance of your application by changing style properties, using ADF skins, and internationalizing the application.

This chapter includes the following sections:

- Section 14.1, "Introduction to Changing ADF Faces Components"
- Section 14.2, "Changing the Style Properties of a Component"
- Section 14.3, "Using Skins to Change the Look and Feel"
- Section 14.4, "Internationalizing Your Application"

14.1 Introduction to Changing ADF Faces Components

ADF Faces components delegate the functionality of the component to a component class, and the display of the component to a renderer. Renderers determine the different ways a component can be displayed on a client, or how to display the component on different clients. The component's tag used on a page determines the unique combination of a component class and a renderer. By default, all tags for ADF Faces combine the associated component class with an HTML renderer, and are part of the HTML render kit. For example, the command button and the command link components are both `UICommand` components; however, they use different renderers. You can create your own custom renderers; it is beyond the scope of this document to explain how to create JSF renderers or custom components.

You cannot customize the ADF Faces renderers. However, you can customize how components display using skins. By default, applications created using ADF Faces components use the Oracle skin. However, the SRDemo sample application uses a custom skin. Skins are an easy way to globally style an application. You can create your own skin to change the colors, fonts, and even the location of portions of ADF Faces components, by setting styles for components in one CSS file. You then configure the application to use the skin when displaying the application. Included with ADF Faces are HTML render kits for display on both desktop and PDA.

If you don't wish to change the entire look of an application, you can choose to change the inline styles for a component on a page. You can also programatically set styles conditionally. For example, you may want to display text red, only under certain conditions.

In addition to changing the appearance of your application, you can also internationalize your application, allowing users in different locales to view text strings in the language to which their browser is set.

Many ADF Faces components include text strings, and the components handle the translation of those strings for you automatically. Any text that is part of the component displays in the language of the user's browser.

You need to translate only the text you add to the application. You can also change other locale-specific properties, such as text direction and currency code.

Read this chapter to understand:

- How to use inline styles to change a component's appearance
- How to conditionally set a style property on a component
- How to create a custom skin
- How to internationalize your application

14.2 Changing the Style Properties of a Component

ADF Faces components use the CSS style properties, based on the Cascading Style Sheet specification. Cascading style sheets contain rules, composed of selectors and declarations that define how styles will be applied. These are then interpreted by the browser and override the browser's default settings. It is beyond the scope of this document to explain the concepts of CSS. Visit the W3C web site (`http://www.w3c.org/`) for extensive information on style sheets, including the official specification.

You can change a style property to alter a component's appearance. ADF Faces components use both inline style properties that can set individual attributes (such as `font-size` and `font-color`), as well as style classes used to group a set of inline styles. For example, the style class `.AFFieldText` sets all properties for the text displayed in an `inputText` component.

14.2.1 How to Set a Component's Style Attributes

You can set inline styles or you can declare a style class for an ADF Faces component on a page.

To set the style:

1. In the Structure window, select the component you wish to style.
2. In the Property Inspector, expand the **Core** node. This node contains all the attributes related to how the component displays.
3. To set a style class for the component, click in the **StyleClass** field and click the ellipses (**...**) button. In the StyleClass dialog, enter a style class for use on this component. For additional help in using the dialog, click **Help**.
4. To set an inline attribute, expand the **InlineStyle** node. Click in the field for the attribute to set, and use the dropdown menu to choose a value.

   You can also use EL expressions for the `InlineStyle` attribute itself to conditionally set inline style attributes. For example, in the SRSearch page of the SRDemo application, the date in the Assigned Date column displays red if a service request has not yet been assigned. Example 14–1 shows the code on the JSF page for the `outputText` component.

Example 14–1 EL Expression Used to Set a Style Attribute

```
<af:outputText value="#{row.assignedDate eq
    null?res['srsearch.highlightUnassigned']:row.assignedDate}"
              inlineStyle="#{row.assignedDate eq null?'color:rgb(255,0,0);':''}"/>
```

14.2.2 What Happens When You Format Text

As Example 14–1 shows, when you use the Property Inspector to set a style, JDeveloper adds the corresponding code for the component to the JSF page.

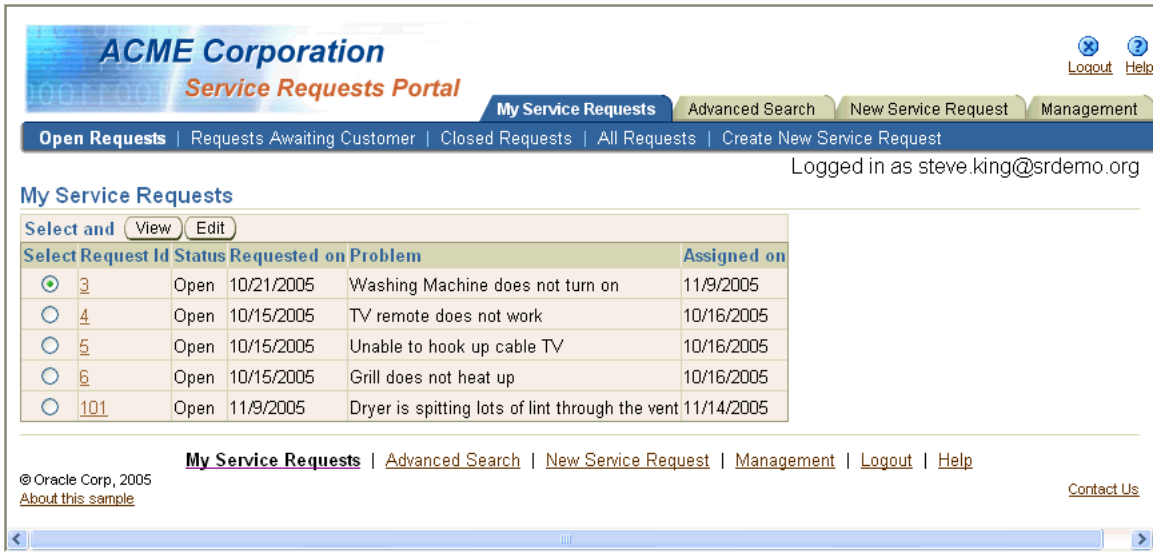
14.3 Using Skins to Change the Look and Feel

Skins allow you to globally change the appearance of ADF Faces components within an application. A skin is a global style sheet that only needs to be set in one place for the entire application. Instead of having to style each component, or having to insert a style sheet on each page, you can create one skin for the entire application. Every component will automatically use the styles as described by the skin. The application developer does not need to add any code, and any changes to the skin will be picked up at runtime, no change to code is needed.

Skins are also based on the Cascading Style Sheet specification. By default, ADF Faces applications use the Oracle skin. Components in the visual editor as well as in the web page display using the settings for this skin. Figure 14–1 shows the SRList page with the Oracle skin applied.

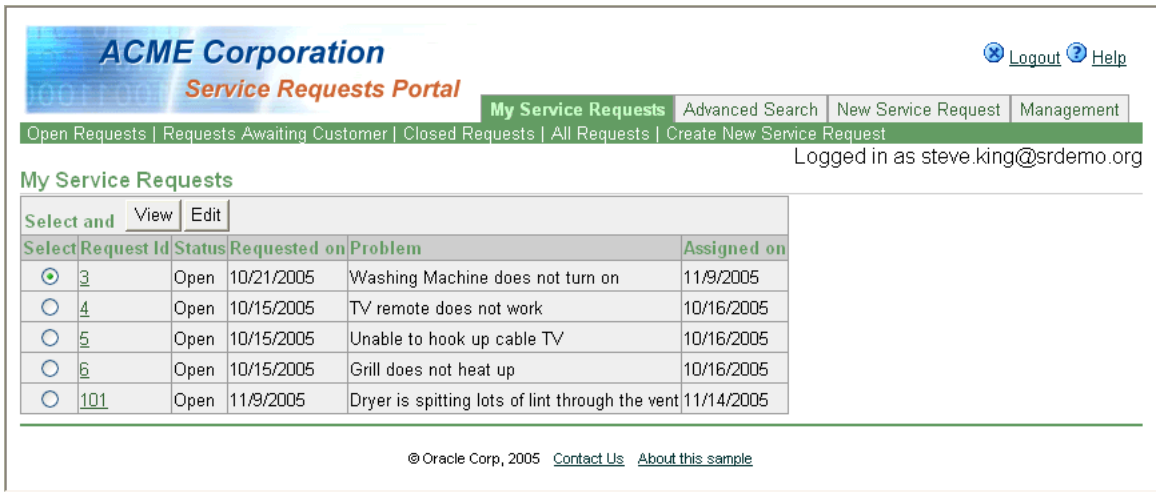
Note: The syntax in a skin style sheet is based on the CSS3 specification. However, many browsers do not yet adhere to this version. At runtime, ADF Faces converts the CSS to the CSS2 specification.

Figure 14–1 The SRList Page Using the Oracle Skin



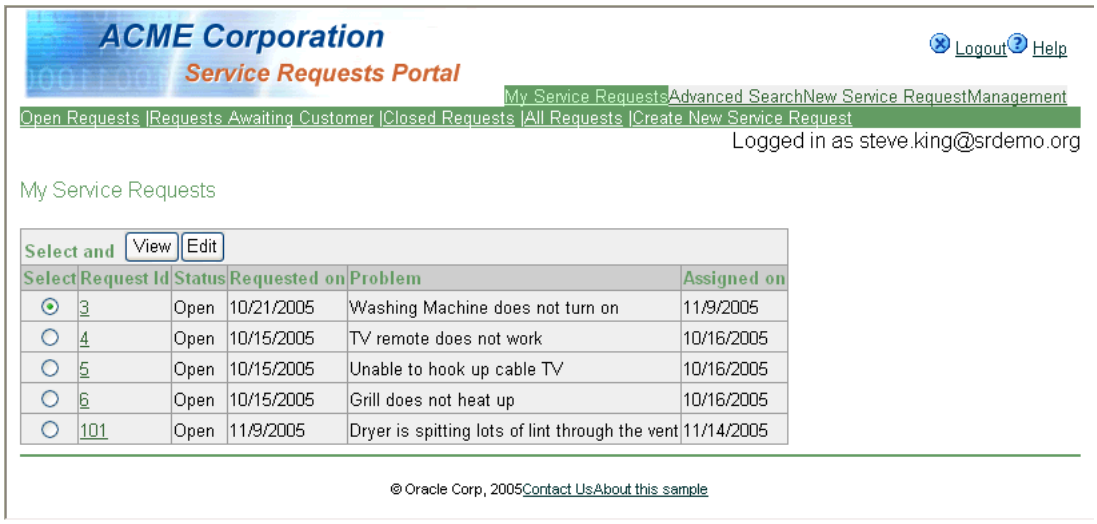
ADF Faces also provides two other skins. The Minimal skin provides some formatting, as shown in Figure 14–2. Notice that almost everything except the graphic for the page has changed, including the colors, the shapes of the buttons, and where the copyright information displays.

Figure 14–2 The SRList Page Using the Minimal Skin



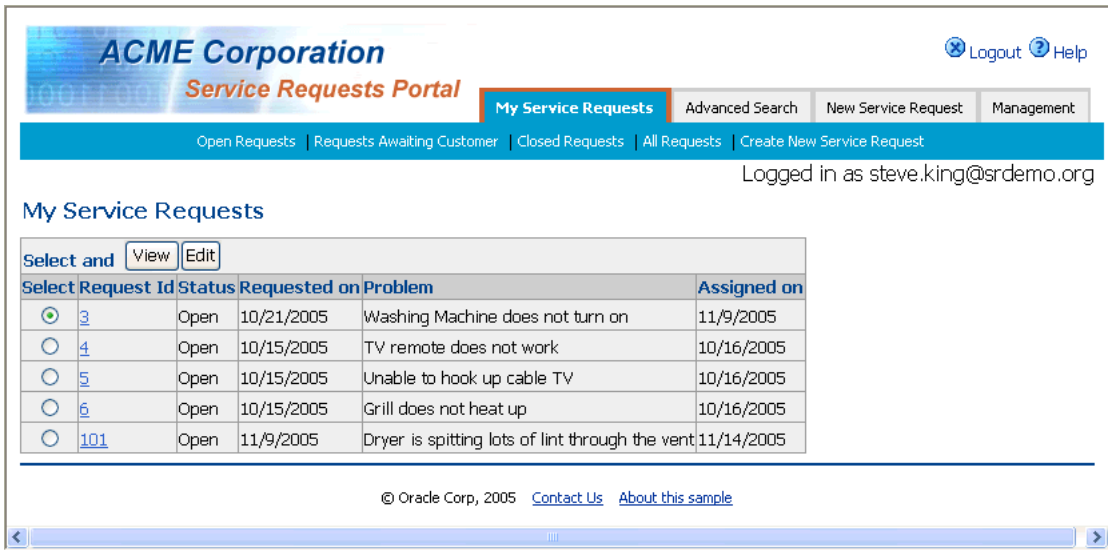
The third skin provided by ADF Faces is the Simple skin. This skin contains almost no special formatting, as shown in Figure 14–3.

Figure 14–3 The SRList Page Using the Simple Skin



The SRDemo application uses a custom skin created just for that application, as shown in Figure 14–4.

Figure 14–4 The SRList Page Using the Custom SRDemo Skin



In addition to using a CSS file to determine the styles, skins also use a resource bundle to determine the text within a component. For example, the word "Select" in the selection column shown in Figure 14–4 is determined using the skin's resource bundle. All the included skins use the same resource bundle.

14.3.1 How to Use Skins

Custom skins extend the Simple skin. To create a custom skin, you declare selectors in a style sheet that override the selectors in the Simple skin's style sheet. Any selectors that you choose not to override will continue to use the style as defined in the Simple skin. Once you create your skin's style sheet, you need to register it as a valid skin in the application, and then configure the application to use the skin.

The selectors used by the simple skin are listed in the "Selectors for Skinning ADF Faces Components" topic in JDeveloper's online help. It is located in the **Reference** > **Oracle ADF Faces** book. This document shows selectors broken down into three sections: global selectors, button selectors, and component-level selectors. Global selectors determine the style properties for multiple components. Examples include the default font family and background colors. Button selectors are used to style all buttons in the application.

Note: Button selectors style all buttons in the application. You cannot define separate selectors for different buttons. For example, the `af:commandButton` and `af:goButton` components will display the same.

Component selectors determine the styles for specific components or portions of a component. Icon selectors denote where the icon can be found.

Within each section are the selectors that can be styled. There are three types of selectors: standard selectors, selectors with pseudo elements, and selectors that use the `alias` pseudo classes. Standard selectors are those that directly represent an element that can have styles applied to it. For example `af|body` represents the `af:body` component. You can set CSS styles, properties, and icons for this type of element.

Pseudo elements are used to denote a specific area of component that can have styles applied. Pseudo elements are denoted by a double colon followed by the portion of the component the selector represents. For example, `af|column::cell-text` provides the styles and properties for the text in a cell of a column.

The `alias` pseudo class is used for a selector that sets styles for more than one component or more than one portion of a component. For example, the `.AFMenuBarItem:alias` selector defines skin properties that are shared by all `af:menuBar` items. Any properties defined in this alias are included in the `af|menuBar::enabled` and `af|menuBar::selected` style classes. If you change the `.AFMenuBarItem:alias` style, you will affect the `af|menuBar::enabled` and `af|menuBar::selected` selectors. You can also create your own pseudo classes for inclusion in other selectors.

You can create multiple skins. For example, you might create one skin for the version of an application for the web, and another for when the application runs on a PDA. Or you can change the skin based on the locale set on the current user's browser. Additionally, you can configure a component, for example a `selectOneChoice` component, to allow a user to switch between skins.

The text used in a skin is defined in a resource bundle. As with the selectors for the Simple skin, you can override the text by creating a custom resource bundle and declaring only the text you want to change. The keys for the text that you can override are documented in the "Reference: Keys for Resource Bundle Used by Skins" topic of the JDeveloper online help. Once you create your custom resource bundle, you register it with the skin.

Note: ADF Faces components provide automatic translation. The resource bundle used for the components' skin is translated into 28 languages. If a user sets the browser to use the German (Germany) language, any text contained within the components will automatically display in German. For this reason, if you create a resource bundle for a custom skin, you must also create localized versions of that bundle for any other languages the application supports. For more information about Internationalization, see Section 14.4, "Internationalizing Your Application".

14.3.1.1 Creating a Custom Skin

You create a custom skin by extending the Simple skin and overriding the selectors. You then need to register the skin with the application.

To create a custom skin:

1. Review your pages using the Simple skin to determine what you would like to change. For procedures on changing the skin, see Section 14.3.1.2, "Configuring an Application to Use a Skin".
2. In JDeveloper, create a CSS file:
   a. Right-click the project that contains the code for the user interface and choose **New** to open the New Gallery.
   b. In the New Gallery, expand the **Web Tier** node and select **HTML**.
   c. Double-click **CSS File**.
   d. Complete the Create Cascading Style Sheet dialog. Click **Help** for help regarding this dialog.

3. Refer to the "Selectors for Skinning ADF Faces Components" topic in JDeveloper's online help. It is located in the **Reference** > **Oracle ADF Faces** book. Add any selectors that you wish to override to your CSS file and set the properties as needed. You can set any properties as specified by the CSS specification.

   If you are overriding a selector for an icon, use a content relative path for the URL to the icon image (that is, start with a leading forward slash), and do not use quotes. Also, you must include the width and the height for the icon. Example 14–2 shows a selector for an icon.

Example 14–2 Selector for an Icon

```
.AFButtonDisabledStartIcon:alias
 {
      content:url(/skins/srdemo/images/btnDisabledStart.gif);
      width:7px; height:18px
 }
```

   Icons and buttons can both use the `rtl` pseudo class. This defines an icon or button for use when the application displays in right-to-left mode. Example 14–3 shows the `rtl` psuedo class used for an icon.

Example 14–3 Icon Selector Using the rtl Psuedo Class

```
.AFButtonDisabledStartIcon:alias:rtl
 {
      content:url(/skins/srdemo/images/btnDisabledStartRtl.gif);
      width:7px; height:18px
 }
```

   Tip: Overriding an alias will likely change the appearance of more than one component. Be sure to carefully read the reference document so that you understand what you may be changing.

4. You can create your own alias classes that you can then include on other selectors. To do so:

   a. Create a selector class for the alias. For example, the SRDemo skin has an alias used to set the color of a link when a cursor hovers over it:

      ```
      .MyLinkHoverColor:alias {color: #CC6633;}
      ```

   b. To include the alias in another selector, add a pseudo element to an existing selector to create a new selector, and then reference the alias using the `-ora-rule-ref:selector` property.

      For example, the SRDemo skin created a new selector for the `af|menuBar::enabled-link` selector in order to style the hover color, and then referenced the custom alias, as shown in Example 14–4.

Example 14–4 Referencing a Custom Alias in a New Selector

```
af|menuBar::enabled-link:hover
{
  -ora-rule-ref:selector(".MyLinkHoverColor:alias");
}
```

5. Save the file to a directory.

Once you've created the CSS, you need to register the skin and then configure the application to use the skin.

To create a custom bundle for the skin:

1. Review the "Reference: Keys for Resource Bundle Used by Skins" topic of the JDeveloper online help and your pages using the Simple skin to determine what text you would like to change. For procedures on changing the skin to the Simple skin, see Section 14.3.1.2, "Configuring an Application to Use a Skin".
2. In JDeveloper, create a resource bundle. It must be of type `java.util.ResourceBundle`. For detailed instructions, see Section 14.4.1, "How to Internationalize an Application".
3. Add any keys to your bundle that you wish to override and set the text as needed.

   Tip: If you internationalize your application, you must also create localized versions of this resource bundle. For more information and procedures, see Section 14.4.1, "How to Internationalize an Application".

To register a custom skin and bundle:

1. If one does not yet exist, create an `adf-faces-skins.xml` file (the file is located in the `<view_project_name>/WEB-INF` directory). This file will be used to declare each skin accessible to the application.
   a. Right-click your view project and choose **New** to open the New Gallery.

      The New Gallery launches. The file launches in the Source editor.

   b. In the **Categories** tree on the left, select **XML**. If **XML** is not displayed, use the **Filter By** dropdown list at the top to select **All Technologies**.
   c. In the **Items** list, select **XML Document** and click **OK**.
   d. Name the file `adf-faces-skins.xml`, place it in the `<view_project_name>/WEB-INF` directory, and click **OK**.
   e. Replace the generated code with the code shown in Example 14–5.

Example 14–5 Default Code for an adf-faces-skins.xml File

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<skins xmlns="http://xmlns.oracle.com/adf/view/faces/skin">

  <skin>

  </skin>

</skins>
```

2. Register the new skin by defining the following for the `skin` element:
   - `<id>`: This value will be used if you want to reference your skin in an EL expression. For example, if you want to have different skins for different locales, you can create an EL expression that will select the correct skin based on its ID.
   - `<family>`: You configure an application to use a particular family of skins. Doing so allows you to group skins together for an application, based on the render kit used.
   - `<render-kit-id>`: This value determines which render kit to use for the skin. You can enter one of the following:
     - `oracle.adf.desktop`: The skin will automatically be used when the application is rendered on a desktop.
     - `oracle.adf.pda`: The skin will be used when rendered on a PDA.
   - `<style-sheet-name>`: This is the fully qualified path to the custom CSS file.
   - `<bundle-name>`: The resource bundle created for the skin. If you did not create a custom bundle, then you do not need to declare this element.

     Note: If you have created localized versions of the resource bundle, you only need to register the base resource bundle.

   Example 14–6 shows the entry in the `adf-faces-skins.xml` file for the SRDemo skin.

Example 14–6 Skins Entry for the SRDemo Skin in the adf-faces-skins.xml File

```
<skin>
  <id>
    srdemo.desktop
  </id>
  <family>
    srdemo
  </family>
  <render-kit-id>
    oracle.adf.desktop
  </render-kit-id>
  <style-sheet-name>
    skins/srdemo/srdemo.css
  </style-sheet-name>
</skin>
```

14.3.1.2 Configuring an Application to Use a Skin

You set an element in the `adf-faces-config.xml` file that determines which skin to use, and if necessary, under what conditions.

To configure an application to use a skin:

1. Open the `adf-faces-config.xml` file.
2. Replace the `<skin-family>` value with the family name for the skin you wish to use. Example 14–7 shows the configuration to use the `srdemo` skin family.

Example 14–7 Configuration to Use a Skin Family

```
<adf-faces-config xmlns="http://xmlns.oracle.com/adf/view/faces/config">
  <skin-family>srdemo</skin-family>
</adf-faces-config>
```

3. To conditionally set the value, enter an EL expression that can be evaluated to determine the skin to display.

   For example, if you want to use the German skin when the user's browser is set to the German locale, and use the English skin otherwise, you would have the following entry in the `adf-faces-config.xml` file:

   ```
   <skin-family>#{facesContext.viewRoot.locale.language=='de' ? 'german' :
   'english'}</skin-family>
   ```

4. To configure a component to dynamically change the skin, you must first configure the component on the JSF page to set a value in scope that can later be evaluated by the configuration file. You then configure the skin family in the `adf-faces-config` file to be dynamically set by that value.

   1. Open the JSF page that contains the component that will be used to set the skin family.
   2. Configure the component to set the skin family in `sessionScope`. Example 14–8 shows a `selectOneChoice` component that takes its selected value, and sets it as the value for the `skinFamily` attribute in `sessionScope`.

Example 14–8 Using a Component to Set the Skin Family

```
<af:selectOneChoice label="Select Skin"
                    value="#{sessionScope.skinFamily}"
                    onchange="form.submit();">
  <af:selectItem label="Simple" value="simple"/>
  <af:selectItem label="Minimal" value="minimal"/>
  <af:selectItem label="Oracle" value="oracle"/>
  <af:selectItem label="SRDemo" value="srdemo"/>
</af:selectOneChoice>
```

The `onchange` event handler will perform a form `POST` when ever a skin is selected in the `selectOneChoice` component. Alternative you can add a command button to the page that will re-submit the page. Every time there is a `POST` the EL expression will be evaluated, and if there is a new value redraw the page with the new skin.

3. In the `adf-faces-config` file, use an EL expression to dynamically evaluate the skin family:

```
<skin-family>#{sessionScope.skinFamily}</skin-family>
```

14.4 Internationalizing Your Application

When your application will be viewed by users in more than one country, you can configure your application to use different locales so that it displays the correct language for the language setting of a user's browser. For example, if you know your application will be viewed in Germany, you can localize your application so that when a user's browser is set to use the German language, text strings in the application will appear in German.

ADF Faces components provide automatic translation. The resource bundle used for the components' skin (which determines look and feel, as well as the text within the component) is translated into 28 languages. If a user sets the browser to use the German (Germany) language, any text contained within the components will automatically display in German. For more information on skins and this resource bundle, see Section 14.3.1, "How to Use Skins". For a complete list of all text included in ADF Faces components, see the "Reference: Keys for Resource Bundle Used by Skins" topic of the JDeveloper online help.

For any text you add to the application, you need to provide a resource bundle that holds the actual text, and you need to load that bundle into the page using the JSF `loadBundle` tag. Then, instead of directly entering the text on the JSF page or entering the text as a value for the `Text` attribute of a component, you bind that attribute to a key in the resource bundle. You then create a version of the resource bundle for each locale.

Note: Any text retrieved from the database is not translated. This document covers how to localize static text, not text that is stored in the database.

Figure 14–5 shows the SRList page from the SRDemo application in a browser set to use the English (United States) language.

Figure 14–5 The SRList Page in English

Although the title of this page is "My Service Requests," instead of having "My Service Requests" as the value for the `title` attribute of the `PanelPage` component, the value is bound to a key in the `UIResources` resource bundle. The `UIResources` resource bundle is loaded into the page using the `loadBundle` tag, as shown in Example 14–9. The resource bundle is given a variable name (in this case `res`) that can then be used in EL expressions. The `title` attribute of the `panelPage` component is then bound to the `srlist.pageTitle` key in that resource bundle.

Example 14–9 Resource Bundles Used in a JSF Page

```
<f:view>
  <f:loadBundle basename="oracle.srdemo.view.resources.UIResources"
                var="res"/>
  <af:document title="#{res['srdemo.browserTitle']}"
               initialFocusId="viewButton">
    <af:form>
      <af:panelPage title="#{res['srlist.pageTitle']}">
```

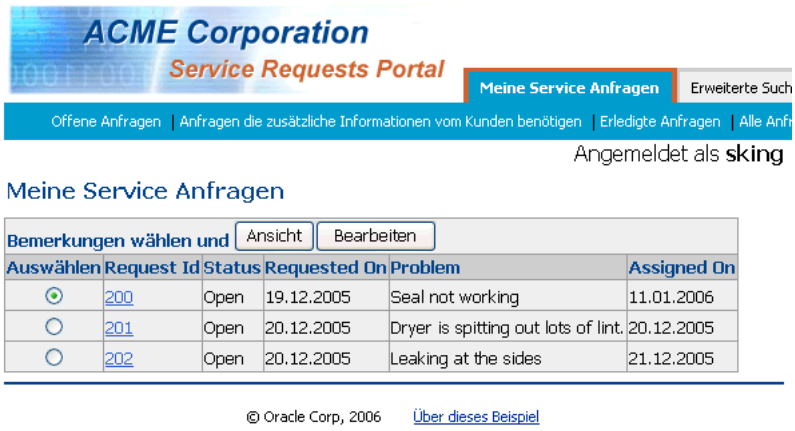
The `UIResources` resource bundle has an entry in the English language for all static text displayed on each page in the SRDemo application, as well as text for messages and global text, such as generic labels. Example 14–10 shows the keys for the SRList page.

Example 14–10 Resource Bundle Keys for the SRList Page Displayed in English

```
#SRList Screen
srlist.pageTitle=My Service Requests
srlist.menubar.openLink=Open Requests
srlist.menubar.pendingLink=Requests Awaiting Customer
srlist.menubar.closedLink=Closed Requests
srlist.menubar.allRequests=All Requests
srlist.menubar.newLink=Create New Service Request
srlist.selectAnd=Select and
srlist.buttonbar.view=View
srlist.buttonbar.edit=Edit
```

Figure 14–6 also shows the SRList page, but with the browser set to use the German (Germany) locale.

Figure 14–6 The SRList Page in German



Note: The column headings were not translated because the values are bound to the label property of the binding object. This value is set as a control hint in the entities structure XML file. Whenever any values are set for the label, a resource bundle is automatically created. To translate these, you must create localized versions of these resource bundles.

Example 14–11 shows the resource bundle version for the German (Germany) language, `UIResource_de`. Note that there is not an entry for the selection facet's title, yet it was translated from "Select" to "Auswahlen" automatically. That is because this text is part of the ADF Faces table component's selection facet.

Example 14–11 Resource Bundle Keys for the SRList Page Displayed in German

```
#SRList Screen
srlist.pageTitle=Meine Service Anfragen
srlist.menubar.openLink=Offene Anfragen
srlist.menubar.pendingLink=Anfrage wartet auf Kunden
srlist.menubar.closedLink=Geschlossene Anfragen
srlist.menubar.allRequests=Alle Anfragen
srlist.menubar.newLink=Erstelle neue Service Anfrage
srlist.selectAnd=Kommentare wählen und
srlist.buttonbar.view=Ansich
```

The resource bundles for the application can be either Java classes or property files. The abstract class `ResourceBundle` has two subclasses: `PropertyResourceBundle` and `ListResourceBundle`. A `PropertyResourceBundle` is stored in a property file, which is a plain-text file containing translatable text. Property files can contain values only for `String` objects. If you need to store other types of objects, you must use a `ListResourceBundle` instead. The contents of a property file must be encoded as ISO 8859-1. Any characters not in that character set must be stored as escaped Unicode.

To add support for an additional locale, you simply replace the values for the keys with localized values and save the property file appending a language code (mandatory), and an optional country code and variant as identifiers to the name, for example, `UIResources_de.properties`. The SRDemo application uses property files.

Note: Property files must contain characters in the IS0 8859-1 character set. If you need to use other characters, use a `ListResourceBundle` class instead.

All non-8859-1 character sets must be converted to escaped UTF-8 characters, or they will not display correctly.

The `ListResourceBundle` class manages resources in a name, value array. Each `ListResourceBundle` class is contained within a Java class file. You can store any locale-specific object in a `ListResourceBundle` class. To add support for an additional locale, you subclass the base class, save it to a file with an locale / language extension, translate it, and compile it into a class file.

The `ResourceBundle` class is flexible. If you first put your locale-specific `String` objects in a `PropertyResourceBundle` file, you can still move them to a

`ListResourceBundle` class later. There is no impact on your code, since any call to find your key will look in both the `ListResourceBundle` class as well as the `PropertyResourceBundle` file.

The precedence order is class before properties. So if a key exists for the same language in both a class file and in a property file, the value in the class file will be the value presented to the user. Additionally, the search algorithm for determining which bundle to load is as follows:

1. (baseclass)+(specific language)+(specific country)+(specific variant)
2. (baseclass)+(specific language)+(specific country)
3. (baseclass)+(specific language)
4. (baseclass)+(default language)+(default country)+(default variant)
5. (baseclass)+(default language)+(default country)
6. (baseclass)+(default language)

For example, if a user's browser is set to the German (Germany) locale and the default locale of the application is US English, the application will attempt to find the closest match, looking in the following order:

1. de_Germany
2. de
3. en_US
4. en
5. The base class bundle

> **Tip:** The `getBundle` method used to load the bundle looks for the default locale classes before it returns the base class bundle. If it fails to find a match, it throws a `MissingResourceException` error. A base class with no suffixes should always exist in order to avoid throwing this exception

14.4.1 How to Internationalize an Application

To internationalize your application, you need to do the following:

> **Tip:** These procedures will allow the application to display the correct language based on the browser settings of the user. You may also want to create your application in a way that allows the user to manually set the locale they wish to use. The current locale is stored in the viewRoot of `FacesContext`.

1. Create a base resource bundle that contains all the text strings that are not part of the components themselves. This bundle should be in the default language of the application.

Tips:

- Instead of creating one resource bundle for the entire application, you can create multiple resource bundles. For example, in a JSF application, you must register the resource bundle that holds error messages with the application in the `faces-config.xml` file. For this reason, you may want to create a separate bundle for messages.
- Create your resource bundle as a Java class instead of a property file if you need to include values for objects other than Strings, or if you need slightly enhanced performance.
- The `getBundle` method used to load the bundle looks for the default locale classes before it returns the base class bundle. However if it fails to find a match, it throws a `MissingResourceException` error. A base class with no suffixes should always exist in order to avoid throwing this exception

2. Use the base resource bundle on the JSF pages by loading the bundle and then binding component attributes to keys in the bundle.
3. Create a localized resource bundle for each locale supported by the application.
4. Register the locales with the application.
5. Register the bundle used for application messages.

Note: If you use a custom skin and have created a custom resource bundle for the skin, you must also create localized versions of that resource bundle. Similarly if your application uses control hints to set any text, you must create localized versions of the generated resource bundles for that text.

Detailed procedures for each step follow.

To create a resource bundle as a property file:

1. In JDeveloper, create a new simple file.
   1. In the Application Navigator, right-click where you want the file to be placed and choose **New** to open the New Gallery.

Note: If you are creating a localized version of the base resource bundle, save the file to the same directory as the base file.

---

   2. In the **Categories** tree, select **Simple Files**, and in the Items list, select **File**.
   3. Enter a name for the file, using the extension `.properties`.

> **Note:** If you are creating a localized version of a base resource bundle, you must append the ISO 639 lowercase language code to the name of the file. For example, the German version of the `UIResources` bundle is `UIResources_de.properties`. You can add the ISO 3166 uppercase country code (for example de_DE) if one language is used by more than one country. You can also add an optional non standard variant (for example, to provide platform or region information).
>
> If you are creating the base resource bundle, no codes should be appended.

2. Create a key and value for each string of static text for this bundle. The key is a unique identifier for the string. The value is the string of text in the language for the bundle. If you are creating a localized version of the base resource bundle, any key not found in this version will inherit the values from the base class.

   > **Note:** All non-ASCII characters must be either UNICODE escaped or the encoding must be explicitly specified when compiling, for example:
   >
   > ```
   > javac -encoding ISO8859_5 UIResources_de.java
   > ```

   For example the key and value for the title of the SRList page is:

   ```
   srlist.pageTitle=My Service Requests
   ```

   > **Note:** All non-8859-1 character sets must be converted to escaped UTF-8 characters, or they will not display correctly.

To create a resource bundle as a Java Class:

1. In JDeveloper, create a new simple Java class:
   - In the Application Navigator, right-click where you want the file to be placed and choose **New** to open the New Gallery.

     > **Note:** If you are creating a localized version of the base resource bundle, this must reside in the same directory as the base file.

   - In the **Categories** tree, select **Simple Files**, and in the **Items** list, select **Java Class**.
   - Enter a name and package for the class. The class must extend `java.util.ListResourceBundle`.

> **Note:** If you are creating a localized version of a base resource bundle, you must append the ISO 639 lowercase language code to the name of the class. For example, the German version of the `UIResources` bundle might be `UIResources_de.java`. You can add the ISO 3166 uppercase country code (for example de_DE) if one language is used by more than one country. You can also add an optional non standard variant (for example, to provide platform or region information).
>
> If you are creating the base resource bundle, no codes should be appended.

2. Implement the `getContents()` method, which simply returns an array of key-value pairs. Create the array of keys for the bundle with the appropriate values. Example 14–12 shows a sample base resource bundle java class.

> **Note:** Keys must be `Strings`. If you are creating a localized version of the base resource bundle, any key not found in this version will inherit the values from the base class.

Example 14–12 Base Resource Bundle Java Class

```
package sample;

import java.util.ListResourceBundle;

public class MyResources extends ListResourceBundle {
  public Object[][] getContents() {
    return contents;
  }
  static final Object[][] contents = {
    {"button_Search", "Search"},
    {"button_Reset", "Reset"},
  };
}
```

To use a base resource bundle on a page:

You need to load only the base resource bundle on the page. The application will automatically use the correct version based on the user's locale setting in their browser.

1. Set your page encoding and response encoding to be a superset of all supported languages. If no encoding is set, the page encoding defaults to the value of the response encoding set using the `contentType` attribute of the page directive. Example 14–13 shows the encoding for the SRList page.

Example 14–13 Page and Response Encoding

```
<?xml version='1.0' encoding='windows-1252'?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.0"
          xmlns:h="http://java.sun.com/jsf/html"
          xmlns:f="http://java.sun.com/jsf/core"
          xmlns:af="http://xmlns.oracle.com/adf/faces"
          xmlns:afc="http://xmlns.oracle.com/adf/faces/webcache">
  <jsp:output omit-xml-declaration="true" doctype-root-element="HTML"
              doctype-system="http://www.w3.org/TR/html4/loose.dtd"
              doctype-public="-//W3C//DTD HTML 4.01 Transitional//EN"/>
  <jsp:directive.page contentType="text/html;charset=UTF-8"/>
  <f:view>
```

Tip: By default JDeveloper sets the page encoding to `windows-1252`. To set the default to a different page encoding:

1. From the menu, choose **Tools** > **Preferences**.
2. In the left-hand pane, select **Environment** if it is not already selected.
3. Set **Encoding** to the preferred default.

2. Load the base resource bundle onto the page using the `loadBundle` tag, as shown in Example 14–14. The `basename` attribute specifies the fully qualified name of the resource bundle to be loaded. This resource bundle should be the one created for the default language of the application. The `var` attribute specifies the name of a request scope attribute under which the resource bundle will be exposed as a `Map`, and will be used in the EL expressions that bind component attributes to a key in the resource bundle.

Example 14–14 The loadBundle Tag

```
<f:loadBundle basename="oracle.srdemo.view.resources.UIResources"
              var="res"/>
```

3. Bind all attributes that represent strings of static text displayed on the page to the appropriate key in the resource bundle, using the variable created in the previous step. Example 14–15 shows the code for the View button on the SRList page.

Example 14–15 Binding to a Resource Bundle

```
<af:commandButton text="#{res['srlist.buttonbar.view']}"
                  . . . />
```

To register locales:

1. Open the `faces-config.xml` file and select the **Overview** tab in the editor window. The `faces-config.xml` file is located in the `<View_Project>/WEB-INF` directory.
2. In the JSF Configuration Editor, select **Application**.
3. If not already displayed, click the **Local Config**'s triangle to display the **Default Locale** and **Supported Locales** fields.
4. For **Default Locale**, enter the ISO locale identifier for the default language to be used by the application. This identifier should represent the language used in the base resource bundle.

5. Add additional supported locales by clicking **New**. Click **Help** or press F1 for additional help in registering the locales.

To register the message bundle:

1. Open the `faces-config.xml` file and click on the **Overview** tab in the editor window. The `faces-config.xml` file is located in the `<View_Project>/WEB-INF` directory.
2. In the window, select **Application**.
3. For **Message Bundle**, enter the fully qualified name of the base bundle that contains messages to be used by the application.

14.4.2 How to Configure Optional Localization Properties for ADF Faces

Along with providing text translation, ADF Faces also automatically provides other types of translation, such as text direction and currency codes. The application will automatically display appropriately based on the user's selected locale. However, you can also manually set the following localization settings for an application in the `adf-faces-config.xml` file.

- `<currency-code>`: Defines the default ISO 4217 currency code used by `oracle.adf.view.faces.converter.NumberConverter` to format currency fields that do not specify a currency code in their own converter.
- `<number-grouping-separator>`: Defines the separator used for groups of numbers (for example, a comma). ADF Faces automatically derives the separator from the current locale, but you can override this default by specifying a value in this element. If set, this value is used by `oracle.adf.view.faces.converter.NumberConverter` while it parses and formats.
- `<decimal-separator>`: Defines the separator (for example, a period or a comma) used for the decimal point. ADF Faces automatically derives the separator from the current locale, but you can override this default by specifying a value in this element. If set, this value is used by `oracle.adf.view.faces.converter.NumberConverter` while it parses and formats.
- `<right-to-left>`: ADF Faces automatically derives the rendering direction from the current locale, but you can explicitly set the default page rendering direction by using the values `true` or `false`.
- `<time-zone>`: ADF Faces automatically uses the time zone used by the client browser. This value is used by `oracle.adf.view.faces.converter.DateTimeConverter` while it converts `Strings` to `Date`.

To configure optional localization properties:

1. Open the `adf-faces-config.xml` file. The file is located in the `<View_Project>/WEB-INF` directory.
2. From the Component Palette, drag the element you wish to add to the file into the Structure window. An empty element is added to the page.
3. Enter the desired value.

Example 14–16 shows a sample `adf-faces-config.xml` file with all the optional localization elements set.

Example 14–16 Configuring Currency Code and Separators for Numbers and Decimal Point

```
<!-- Set the currency code to US dollars. -->
<currency-code>USD</currency-code>

<!-- Set the number grouping separator to period for German -->
<!-- and comma for all other languages -->
<number-grouping-separator>
 #{view.locale.language=='de' ? '.' : ','}
</number-grouping-separator>

<!-- Set the decimal separator to comma for German -->
<!-- and period for all other languages -->
<decimal-separator>
 #{view.locale.language=='de' ? ',' : '.'}
</decimal-separator>

<!-- Render the page right-to-left for Arabic -->
<!-- and left-to-right for all other languages -->
<right-to-left>
 #{view.locale.language=='ar' ? 'true' : 'false'}
</right-to-left>

<!-- Set the time zone to Pacific Daylight Savings Time -->
<time-zone>PDT</time-zone>
```

15

Optimizing Application Performance with Caching

This chapter describes how to add caching support to existing application pages.

This chapter includes the following sections:

- Section 15.1, "About Caching"
- Section 15.2, "Using ADF Faces Cache to Cache Content"

15.1 About Caching

For most Web-based applications, a large percentage of requests are made for identical or similar content. These repeated requests for both dynamic and static contents place a significant strain on application infrastructure.

Caching stores all or parts of a web page in memory for use in future responses. It significantly reduces response time to client requests by reusing cached content for future requests without executing the code that created it.

Oracle ADF Faces Cache provides a simple way for you to cache portions of a response generated by a request. You simply wrap the fragment content you want to cache with a beginning `<afc:cache>` and ending `</afc:cache>` tag. By caching both dynamic and static content, you can increase throughput and shorten response times.

You can add the `<afc:cache>` tag to cache the following fragment types:

- Page fragment—You make the `<afc:cache>` tag a direct child of the `<f:view>` tag, and enclose the page's content within it.
- Fragment within a page—You enclose only the fragment portion within the `<afc:cache>` tag. Caching fragments is useful when sections of a page must be created for each request.
- Included fragment that exists in its own subpage—You make the `<afc:cache>` tag a direct child of the `<f:subview>` tag, and enclose the fragment's content within it.

You can use the ADF Faces Cache library with any application developed with JavaServer Faces (JSF).

15.2 Using ADF Faces Cache to Cache Content

Consider using the `<afc:cache>` tag for the following types of content:

- Resource Intensive

  If rendering a particular JSF or ADF component requires resource-intensive operations like making database or network queries, caching can help to reduce the rendering cost by retrieving content from the cache as opposed to regenerating it.

- Shareable

  The cache can serve the same object to multiple users or sessions.

  The degree of sharing can be application wide or limited by certain properties, such as a bean property, user cookie, or request header.

- Changes infrequently

  Infrequently changing content is ideal to cache, because the cache can serve the content for a long period of time. The ADF Faces Cache expiration and invalidation mechanisms help to invalidate content in the cache. Use expiration when you can accurately predict when the source of the content will change; use invalidation for content that changes from a request.

  Because frequently changing content requires constant cache updates, this content is not ideal to cache.

Several of the pages in the SRDemo application use the `Cache` component to cache fragments. By analyzing how caching support was added to `SRCreate.jspx` and `SRFaq.jspx`, you can better understand how to cache fragments in your applications.

Figure 15–1 shows the `SRCreate.jspx` page. It contains these cacheable fragments:

- The first fragment contains content at the start of the page, including the text and link to the Frequently Asked Questions, the prompt to enter a basic description of your problem, and the `objectSeparator` component.

  This content is generic to all users.

- The second fragment contains the `panelForm` component for selecting an appliance, requiring a database query.

  This content varies by the user. The content is valid across all sessions for the same user.

- The third fragment contains the tabs, including the **New Service Request** tab.

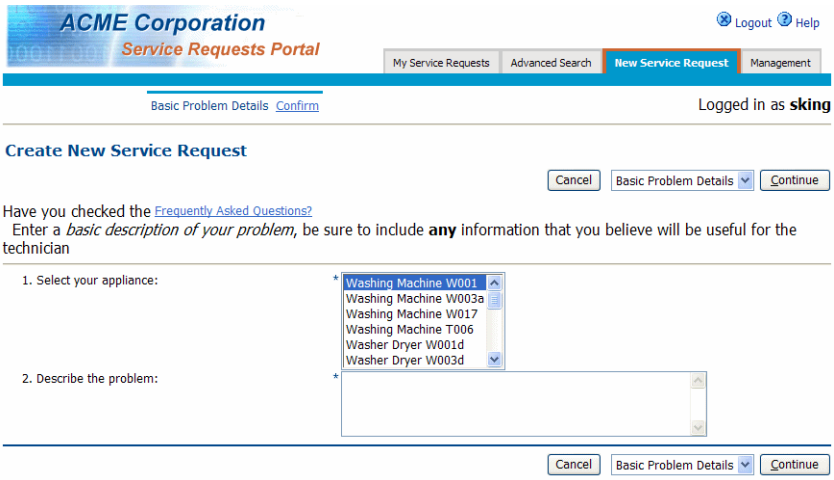
  This content varies by the user. The content is valid across all sessions for the same user.

- The forth fragment contains the **Logout** and **Help** menu item at the top of the page.

  This content is generic to all users.

Because these fragments are shareable by a given user across sessions or across all users, they are good caching candidates.

Figure 15–1 Create New Service Request Page in the SRDemo Application



Example 15–1 shows the code for the first fragment, the start of the page content.

Example 15–1 Start Page Content Fragment

```
<!--Page Content Start-->
<afc:cache duration="864000">
   <af:objectSpacer width="10" height="10"/>
   <af:panelHorizontal>
     <f:facet name="separator">
       <af:objectSpacer width="4" height="10"/>
     </f:facet>
     <af:outputText value="#{res['srcreate.faqText']}"/>
     <af:commandLink text=" #{res['srcreate.faqLink']}"
                              action="dialog:FAQ" useWindow="true"
                              immediate="true" partialSubmit="true"/>
   </af:panelHorizontal>
   <af:objectSpacer width="10" height="10"/>
   <af:outputFormatted value="#{res['srcreate.explainText']}"/>
   <af:objectSeparator/>
</afc:cache>
```

The attributes for the `<afc:cache>` tag specify the following:

- The `duration` attributes specifies 86,400 seconds before the fragment expires. When a fragment expires and client requests it, it is removed from the cache and then refreshed with new content.

Example 15–2 shows the code for the second fragment, the `panelForm` component for selecting your appliance.

Example 15–2 Appliance Selection Fragment

```
<af:panelForm>
  <afc:cache duration="86400"
             varyBy="userInfo.userId">
   <af:panelLabelAndMessage valign="top"
                            label="#{res['srcreate.info.1']}">
    <af:selectOneListbox id="navList1" autoSubmit="false"
                         value="#{bindings.findAllProduct1.inputValue}"
                         size="6" required="true">
      <f:selectItems value="#{bindings.findAllProduct1.items}"/>
    </af:selectOneListbox>
   </af:panelLabelAndMessage>
  </afc:cache>
```

The attributes for the `<afc:cache>` tag specify the following:

- The `duration` attributes specifies 86,400 seconds before the fragment expires.
- The `varyBy` attribute specifies which version of the fragment to display based on the `userInfo` bean. This attribute specifies to cache a version of the fragment for each user. The content is valid across sessions for the same user.

Example 15–3 shows the code for the third fragment, the tabs across the top of the page.

Example 15–3 Menu Tabs Fragment

```
<f:facet name="menu1">
    <afc:cache duration="864000"
               varyBy="userInfo.userId">
    <af:menuTabs var="menuTab" value="#{menuModel.model}">
      <f:facet name="nodeStamp">
        <af:commandMenuItem text="#{menuTab.label}"
                            action="#{menuTab.getOutcome}"
                            rendered="#{menuTab.shown and
menuTab.type=='default'}"
                           disabled="#{menuTab.readOnly}"/>
      </f:facet>
    </af:menuTabs>
  </afc:cache>
</f:facet>
```

The attributes for the `<afc:cache>` tag specify the following:

- The `duration` attribute specifies 86,400 seconds before the fragment expires.
- The `varyBy` attribute specifies which version of the fragment to display based on the `userInfo` bean.

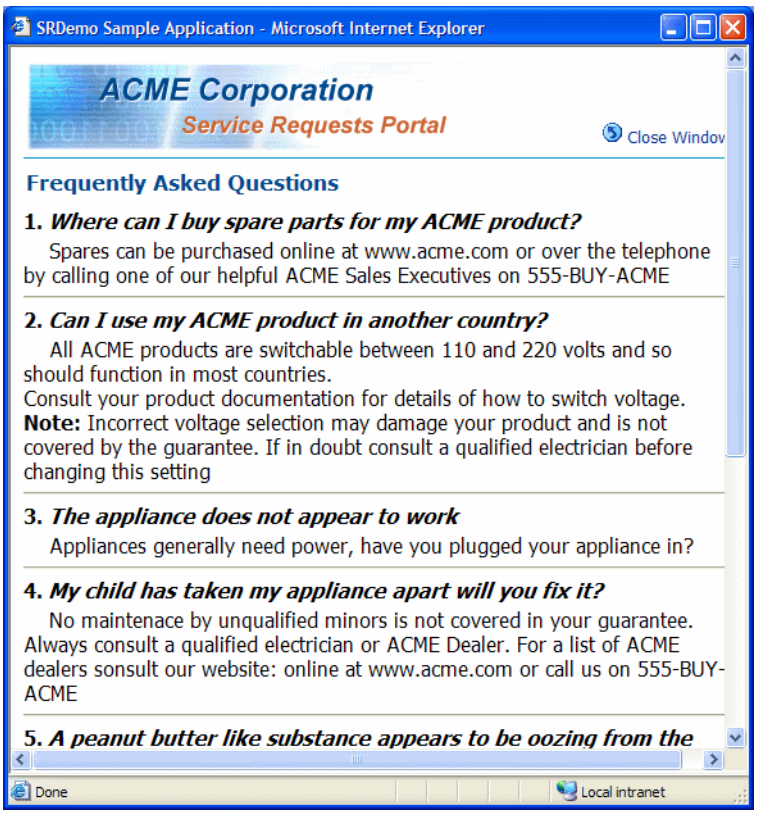
Example 15–4 shows the code for the last fragment, the **Logout** and **Help** menu items.

Example 15–4 Logout and Help Menu Fragment

```
<f:facet name="menuGlobal">
    <afc:cache duration="86400">
      <af:menuButtons>
        <af:commandMenuItem text="#{res['srdemo.menu.logout']}"
                            action="GlobalLogout"
                            immediate="true"
                            icon="/images/logout.gif"/>
        <af:commandMenuItem text="#{res['srdemo.menu.help']}"
                            action="GlobalHelp"
                            immediate="true"
                            icon="/images/help.gif"/>
      </af:menuButtons>
    </afc:cache>
</f:facet>
```

Figure 15–2 shows the `SRFaq.jspx` page. Its content is shareable among all users.

Figure 15–2 Frequently Asked Questions Dialog in the SRDemo Application



Example 15–5 shows the code for this page fragment.

Example 15–5 FAQ Fragment

```
<f:view>
    <afc:cache duration="86400"
               searchKeys="FAQ"
...FAQ Page Content...
   </afc:cache>
</f:view>
```

The attributes for the `<afc:cache>` tag specify the following:

- The `duration` attribute specifies 86,400 seconds before the fragment expires.
- The `searchKeys` attribute assigns this page fragment a search string of `FAQ`. You can invalidate this fragment using this search key.

  You use search keys to organize web pages and fragments into different groups. You can assign all the pages in a particular group with the same search key. For example, you can assign the search key `new_request` to all the pages that have something to do with creating a new service requests. To invalidate a group of objects, you submit an invalidation request that specifies the search key associated with that particular group. For example, if the invalidation request specifies the search key `new_request`, all the pages assigned the `new_request` search key will be invalidated. In the SRDemo application, the `SRFaq.jspx` page is the only page assigned a search key.

  When objects are marked as invalid and a client requests them, they are removed and then refreshed with new content.

15.2.1 How to Add Support for ADF Faces Cache

To use the `Cache` component, you add the ADF Faces Cache library to an application's project and apply the library to the specific JSP page.

To add the ADF Faces Cache library:

1. In the Application Navigator, select the project that you want to use the `Cache` component.
2. From the context menu, choose **Project Properties**.

   The Project Properties dialog opens.
3. Select the **Libraries** node.
4. On the Libraries page, click **Add Library**.
5. Locate the ADF Faces Cache library in the selection tree and click **OK**.
6. On the Libraries page, click **OK**.
7. For each JSP document or page, you plan to apply the `<afc:cache>` tag, add the following library syntax to the `<jsp:root>` tag:

   ```
   xmlns:afc="http://xmlns.oracle.com/adf/faces/webcache"
   ```

   You can now insert the `Cache` component from the Component Palette or use Code Insight to insert the `<afc:cache>` tag.

15.2.2 What Happens When You Cache Fragments

When you run an application containing the `<afc:cache>` tag, the content is not cached until there is an initial browser request for it. After the content is cached, the content is served from the cache. You can see when content is inserted into the cache and how many cache hits and misses result from fragment requests using a combination of the following tools:

- Logging
- AFC Statistics Servlet
- Visual Diagnostics

15.2.2.1 Logging

ADF Faces Cache leverages the Java Logging API (java.util.logging.Logger) to log events and error messages. These messages show the sequence of how objects are inserted and served from the cache.

Depending on the logging configuration specified in the `j2ee-logging.xml` file, logging information can display in the Log Window of JDeveloper and write to the `log.xml` file. The `j2ee-logging.xml` file specifies the directory path information for `log.xml`.

Example 15–6 shows log excerpts in which fragment `SRCreate.jspx` is initially requested and found not to be in the cache (`cache miss`) and inserted into the cache (`insert`). `SRCreate.jspx` is requested again, and served from the cache (`cache hit`).

Example 15–6 Log Sample

```
fragment is SRCreate.jspx:_id13
fragment (SRCreate.jspx:_id13) fetch: cache miss
fragment (SRCreate.jspx:_id13) insert: cached for 86400 secs
...
fragment is SRCreate.jspx:_id19
fragment (SRCreate.jspx:_id19) fetch: cache hit
...
```

> **See Also:** Section A.9 for further information about the `j2ee-logging.xml` file

15.2.2.2 AFC Statistics Servlet

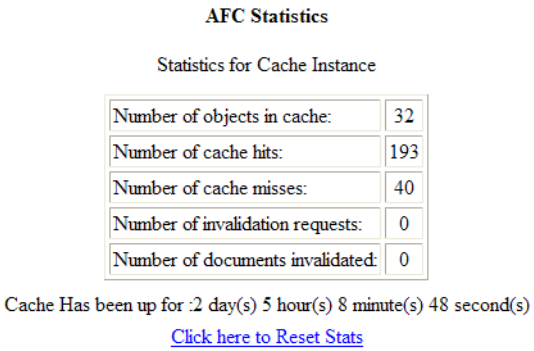
The AFC Statistics servlet, shown in Figure 15–3, displays the following cache statistics. These statistics can help to provide an overall picture of cache throughput:

- **Number of objects in cache**–The number of objects stored in the cache.
- **Number of cache hits**–The number of requests served by objects in the cache.
- **Number of cache misses**–The number of cacheable requests that were not served by the cache. This number represents initial requests and requests for invalidated or expired objects that have been refreshed.
- **Number of invalidation requests**–The number of invalidation requests serviced by the cache.
- **Number of documents invalidated**–The total number of objects invalidated by the cache.

The **Number of invalidation requests** and the **Number of documents invalidated** may not be the same. This difference can occur because one search key may apply to more than one object.

The **Click here to Reset Stats** link, shown in Figure 15–3, resets these statistics, except for **Number of objects in cache**.

Figure 15–3 AFC Statistics Servlet



To enable the servlet:

1. Create the following entry in the `web.xml` file in the `/WEB-INF` directory of the application:

```
<servlet>
   <servlet-name>AFCStatsServlet</servlet-name>
   <servlet-class>oracle.webcache.adf.servlet.AFCStatsServlet</servlet-class>
</servlet>
```

2. Point your browser to the following URL:

```
http://application_host:application_
port/application-context-root/servlet/AFCStatsServlet
```

> **See Also:** Topic "Viewing Cache Performance Statistics" in the JDeveloper online help for further information about the AFC Statistics servlet

15.2.2.3 Visual Diagnostics

The visual diagnostics feature enables you to visually display whether fragments are cache hits or cache misses. This feature demarcates fragment output with the HTML `<SPAN>` tag, using a class appropriate for its cache hit or cache miss status. By setting a distinct class style, you can visually determine whether fragments are stored in the cache.

While the SRDemo application does not use the visual diagnostics feature, you may find it useful for testing your applications.

> **See Also:** Topic "Using Visual Diagnostics" in the JDeveloper online help for further information

15.2.3 What You May Need to Know

When you use AFC Statistics servlet, you may encounter the following problems:

- HTTP `404 Page Not Found` error code

  If you receive this error when accessing the servlet, it is most likely the result of a configuration issue.

  To resolve this problem, ensure the following lines are present in the `web.xml` file:

  ```
  <servlet>
     <servlet-name>AFCStatsServlet</servlet-name>
     <servlet-class>oracle.webcache.adf.servlet.AFCStatsServlet</servlet-class>
  </servlet>
  ```

- `Cache instance is not running` error

  This error occurs because the servlet has not started to monitor the cache. The servlet only starts to monitor the cache after the first object has been inserted into the cache and the cache instance is created.

  To workaround this error, select **Click here to Reset Stats**.

16

Testing and Debugging Web Applications

This chapter describes the process of debugging your user interface project. It also supplies information about methods of the Oracle ADF Model API, which you can use to set breakpoints for debugging.

This chapter includes the following sections:

- Section 16.1, "Getting Started with Oracle ADF Model Debugging"
- Section 16.2, "Correcting Simple Oracle ADF Compilation Errors"
- Section 16.3, "Correcting Simple Oracle ADF Runtime Errors"
- Section 16.4, "Understanding a Typical Oracle ADF Model Debugging Session"
- Section 16.5, "Debugging the Oracle ADF Model Layer"
- Section 16.6, "Tracing EL Expressions"

16.1 Getting Started with Oracle ADF Model Debugging

Like any debugging task, debugging the web application's interaction with Oracle ADF is a process of isolating specific contributing factors. However, in the case of web applications, generally, this process does not involve compiling Java source code. Your web pages contain no Java source code, as such, to compile. In fact, you may not realize that a problem exists until you run and attempt to use the application. For example, these failures are only visible at runtime:

- Page not found servlet error
- Page is found but the components display without data
- Page fails to display data after executing a method call or built-in operation (like Next or Previous)
- Page displays but a method call or built-in operation fails to execute at all
- Page displays but unexpected validation errors occur

The failure to display data or to execute a method call arises from the interaction between the web page's components and the Oracle ADF Model layer. When a runtime failure is observed during ADF lifecycle processing, the sequence of preparing the model, updating the values, invoking the actions, and, finally, rendering the data failed to complete.

Fortunately, most failures in the web application's interaction with Oracle ADF result from simple and easy-to-fix errors in the declarative information that the application defines or in the EL expressions that access the runtime objects of the page's Oracle ADF binding container.

Therefore, in your Oracle ADF databound application, you should examine the declarative information and EL expressions as likely contributing factors when runtime failures are observed. Read the following sections to understand editing the declarative files:

- Section 16.2, "Correcting Simple Oracle ADF Compilation Errors"
- Section 16.3, "Correcting Simple Oracle ADF Runtime Errors"

The most useful diagnostic tool (short of starting a full debugging session) that you can use when running your application is the ADF Logger. You use this J2EE logging mechanism in JDeveloper to capture runtime traces messages from the Oracle ADF Model layer API. With ADF logging enabled, JDeveloper displays the application trace in the Message Log window. The trace includes runtime messages that may help you to quickly identify the origin of an application error. Read Section 16.4, "Understanding a Typical Oracle ADF Model Debugging Session" to configure the ADF Logger to display detailed trace messages.

If the error cannot be easily identified, you can utilize the debugging tools in JDeveloper to step through the execution of the application and the various phases of the Oracle ADF page lifecycle. This process will help you to isolate exactly where the error occurred. By using the debugging tools, you will be able to pause execution of the application on specific methods in the Oracle ADF API, examine the data that the Oracle ADF binding container has to work with, and compare it to what you expect the data to be. Read Section 16.5, "Debugging the Oracle ADF Model Layer" to understand debugging the Oracle ADF Model layer.

Occasionally, you may need help debugging EL expressions. While EL is not well-supported with a large number of useful exceptions, you can enable JSF trace messages to examine variable resolution. Read Section 16.6, "Tracing EL Expressions" to work with JSF trace messages.

16.2 Correcting Simple Oracle ADF Compilation Errors

When you create web pages and work with the ADF data controls to create the ADF binding definitions in JDeveloper, the Oracle ADF declarative files you edit must conform to the XML schema defined by Oracle ADF. When an XML syntax error occurs, the JDeveloper XML compiler immediately displays the error in the Structure window. Choose **Structure** from the JDeveloper **View** menu to open the Structure window for any Oracle ADF file you edit in the XML editor.

Currently a limitation of the JDeveloper compiler is the ability to resolve EL expressions. EL expressions in your web pages interact directly with various runtime objects in the web environment, including the web page's Oracle ADF binding container. At present, errors in EL expressions can be observed only at runtime. Thus, the presence of a single typing error in an object-access expression will not be detected by the compiler, but will manifest at runtime as a failure to interact with the binding container and a failure to display data in the page. For information about debugging runtime errors, see Section 16.3, "Correcting Simple Oracle ADF Runtime Errors".

> **Tip:** The JDeveloper Expression Builder is a dialog that helps you build EL expressions by providing lists of objects, managed beans, and properties. It is particularly useful when creating or editing ADF databound EL expressions because it provides a hierarchical list of ADF binding objects and their valid properties from which you can select the ones you want to use in an expression. Oracle recommends using the Expression Builder to avoid introducing typing errors. For details, see Section 5.6.2, "How to Use the Expression Builder".

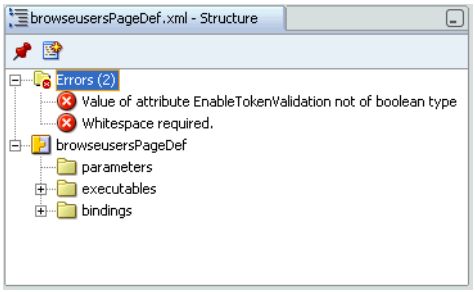
Example 16–1 illustrates simple compilation errors contained in the page definition file: "`fase`" instead of "`false`" and "`IsQueriable="false"/`" instead of "`IsQueriable="false"/>`" (missing a closing angle bracket).

Example 16–1 Sample Page Definition File with Two Errors

```
<?xml version="1.0" encoding="UTF-8" ?>
<pageDefinition xmlns="http://xmlns.oracle.com/adfm/uimodel"
                version="10.1.3.35.62" id="browseusersPageDef"
                Package="oracle.srdemo.view.pageDefs"
                EnableTokenValidation="fase"
               ...>
  <parameters/>
  <executables>
    <variableIterator id="variables">
      <variable Type="java.lang.String" Name="findUsersByName_name"
                IsQueriable="false"/
```

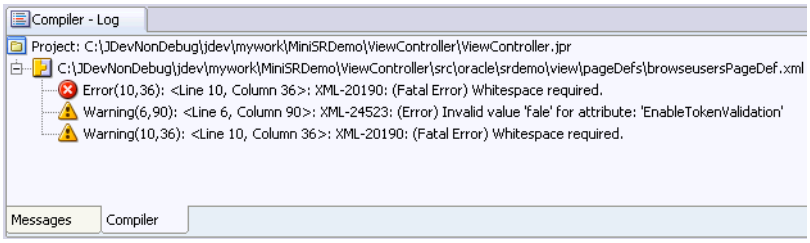
The Structure window for the above errors would display as shown in Figure 16–1.

Figure 16–1 Structure Window Displays XML Error



If you were to attempt to compile the application, the Compiler window would also display similar errors, as shown in Figure 16–2.

Figure 16–2 Compiler Window Displays XML Compile Error



To correct schema validation errors, in either the Structure window or the Compiler window, double-click the error to open the file. The file will open in the XML editor with the responsible line highlighted for you to fix.

After you correct the error, the Structure window will immediately remove the error from the window. Optionally, you may recompile the project using the make operation to recompile the changed file and view the empty Compiler window.

16.3 Correcting Simple Oracle ADF Runtime Errors

Failures of the Oracle ADF Model layer cannot be detected by the JDeveloper compiler, in part, because the page's data-display and method-execution behavior relies on the declarative Oracle ADF page definition files. The Oracle ADF Model layer utilizes those declarative files at runtime to create the objects of the Oracle ADF binding container.

To go beyond simple schema validation, you will want to routinely run and test your web pages to ensure that one of the following conditions does not exist:

- The project dependency between the data model project and the user interface project becomes disabled.

  By default, the dependency between projects is enabled whenever you create a web page that accesses a data control in the data model project. However, if the dependency is disabled and remains disabled when you attempt to run the application, an internal servlet error will be generated at runtime:

  ```
  oracle.jbo.NoDefException: JBO-25002: Definition
  model.DataControls.dcx of type null not found
  ```

  To correct the error, right-click the user interface project, choose **Project Properties**, and select **Dependencies** in the dialog. Make sure that the ***<ModelProjectName>*.jpr** option appears selected in the panel.

- The `DataBindings.cpx` file location changed but the `web.xml` file still references the original path for the file.

  By default, JDeveloper adds the `DataBindings.cpx` file to the package for your user interface project. If a change to the location of the file is made (for example, due to refactoring the application), an internal servlet error will be generated at runtime:

  ```
  oracle.jbo.NoXMLFileException: JBO-26001: XML File not found
  for the Container /oracle/<path>/DataBinding.cpx
  ```

  To correct the error, open the `web.xml` file and edit the path that appears in the `<context-param>` element `CpxFileName`.

- Page definition files have been renamed but the `DataBindings.cpx` file still references the original page definition filenames.

  While JDeveloper does not permit these files to be renamed within the IDE, if a page definition file is renamed outside of JDeveloper and the references in the `DataBindings.cpx` file are not also updated, an internal servlet error will be generated at runtime:

  ```
  oracle.jbo.NoDefException: JBO-25002: Definition
  oracle.<path>.pageDefs.<pagedefinitionName> of type Form
  Binding Definition not found
  ```

  To correct the error, open the `DataBindings.cpx` file and edit the page definition filenames that appear in the `<pageMap>` and `<pageDefinitionUsages>` elements.

- The web page file (`.jsp` or `.jspx`) has been renamed but the `DataBindings.cpx` file still references the original filename of the same web page.

The page controller uses the page's URL to determine the correct page definition to use to create the ADF binding container for the web page. If the page's name from the URL does not match the `<pageMap>` element of the `DataBindings.cpx` file, an internal servlet error will be generated at runtime:

```
javax.faces.el.PropertyNotFoundException: Error testing
property <propertyname>
```

To correct the error, open the `DataBindings.cpx` file and edit the web page filenames that appear in the `<pageMap>` element.

- Bindings have been renamed in the web page EL expressions but the page definition file still references the original binding object names.

  The web page may fail to display information that you expect to see. To correct the error, compare the binding names in the page definition file and the EL expression responsible for displaying the missing part of the page. Most likely the mismatch will occur on a value binding, with the consequence that the component will appear but without data. Should the mismatch occur on an iterator binding name, the error may be more subtle and may require deep debugging to isolate the source of the mismatch.

- Bindings in the page definition file have been renamed or deleted and the EL expressions still reference the original binding object names.

  Because the default error-handling mechanism will catch some runtime errors from the ADF binding container, this type of error can be very easy to find. For example, if an iterator binding (named `findUsersByNameIter`) was renamed in the page definition file, yet the page still refers to the original name, this error will display in the web page:

  ```
  JBO-25005: Object name findUsersByNameIter for type Iterator
  Binding Definition is invalid
  ```

  To correct the error, right-click the name in the web page and choose **Go to Page Definition** to locate the correct binding name to use in the EL expression.

- EL expressions were written manually instead of using the Expression Picker dialog and invalid object names or property names were introduced.

  This error may not be easy to find. Depending on which EL expression contains the error, you may or may not see a servlet error message. For example, if the error occurs in a binding property with no runtime consequence, such as displaying a label name, the page will function normally but the label will not be displayed. However, if the error occurs in a binding that executes a method, an internal servlet error `javax.faces.el.MethodNotFoundException: <methodname>` will display. Or, in the case of an incorrectly typed property name on the method expression, the servlet error `javax.faces.el.PropertyNotFoundException: <propertyname>` will display. For information about displaying JSF trace messages to help debug these exception, see Section 16.6, "Tracing EL Expressions".

If the above list of typical errors does not help you to find and fix a runtime error, you can initiate debugging within JDeveloper in order to isolate the contributing factor. This process involves pausing the execution of the application as it proceeds through the phases of the Oracle ADF page lifecycle, examining the data received by the lifecycle, and determining whether that data is expected or not. To inspect the data of your application, you will work with source code breakpoints and Data window, as described in Section 16.4, "Understanding a Typical Oracle ADF Model Debugging Session".

16.4 Understanding a Typical Oracle ADF Model Debugging Session

If you are not able to easily find the error in your web page or its corresponding page definition file, you can use the JDeveloper debugging tools to investigate where your application failure occurs. Specifically, the goal for debugging the interaction between the web page and the Oracle ADF Model layer is to pause the application by setting breakpoints on the execution of the Oracle ADF page lifecycle and to examine the data loaded at runtime. When the objects of the Oracle ADF Model layer do not contain the data that you expect to see, this observation will help you to identify the probable contributing factor.

Generally, the process for debugging proceeds like this:

1. Run the application and look for missing or incomplete data, actions and methods that are ignored or incorrectly executed, or other unexpected results.
2. Create a debugging configuration that will enable the ADF Log and send Oracle ADF Model messages to the JDeveloper Log window. For more information, see Section 16.4.2, "Creating an Oracle ADF Debugging Configuration".
3. Choose **Go to Java Class** from the **Navigate** menu (or press Ctrl + -) and use the dialog to locate the Oracle ADF class that represents the entry point for the processing failure.

   Tip: JDeveloper will locate the class from the user interface project that has the current focus in the Application Navigator. If your workspace contains more than one user interface project, be sure the one with the current focus is the one that you want to debug.

4. Open the class file in the Java editor and find the Oracle ADF method call that will enable you to step into the statements of the method.
5. Set a breakpoint on the desired method and run the debugger.
6. When the application stops on the breakpoint, use the Data window to examine the local variables and arguments of the current context.

Once you have set breakpoints to pause the application at key points, you can proceed to view data in the JDeveloper Data window. To effectively debug your web page's interaction with the Oracle ADF Model layer, you need to understand:

- The Oracle ADF page lifecycle and the method calls that get invoked
- The local variables and arguments that the Oracle ADF Model layer should contain during the course of application processing

Awareness of Oracle ADF processing, as described in Section 16.5, "Debugging the Oracle ADF Model Layer", will give you the means to selectively set breakpoints, examine the data loaded by the application, and isolate the contributing factors.

Note: JSF web pages may also use backing beans to manage the interaction between the page's components and the data. Debug backing beans by setting breakpoints as you would any other Java class file.

16.4.1 Turning on Diagnostic Logging

Even before you use the actual debugger, running with framework diagnostics logging turned on can be helpful to see what happened when the problem occurs. To turn on diagnostic logging set the Java System property named `jbo.debugoutput` to the value `console`. Additionally, the value `ADFLogger` lets you route diagnostics through the standard J2SE Logger implementation, which can be controlled in a standard way through the OC4J `j2ee-logging.xml` file.

The easiest way to set this system property while running your application inside JDeveloper is to edit your project properties and in the Run/Debug panel, select a run configuration and click **Edit** to edit it. Then add the string `-Djbo.debugoutput=console` to the **Java Options** field.

16.4.2 Creating an Oracle ADF Debugging Configuration

ADF Faces leverages the Java Logging API (`java.util.logging.Logger`) to provide logging functionality when you run a debugging session. Java Logging is a standard API that is available in the Java Platform, starting with JDK 1.4. For the key elements, see the section "Java Logging Overview" at `http://java.sun.com/j2se/1.4.2/docs/guide/util/logging/overview.html`.

Because standard Java Logging is used, you can edit the `j2ee-logging.xml` file to control the level of diagnostics you receive in the Log window:

- When you conduct a debugging session within JDeveloper, you will use JDeveloper embedded-OC4J and will want to modify the file in your JDeveloper install here:

  <JDev_Install>`/jdev/system/oracle.j2ee.10.1.3.xx.xx/embedded-oc4j/config`

- Similarly, when you want to conduct a remote debugging session on Oracle Application Server, you can modify the file here:

  <OAS_Home>`/j2ee/<OC4J_INSTANCE>/config`

- Or, when you want to conduct a remote debugging session on standalone OC4J, you can modify the file here:

  <OC4J_Home>`/j2ee/home/config`

To edit ADF package-level logging in the j2ee-logging.xml file:

If you want to change the logging level for Oracle ADF, you can edit the `<logger>` elements of the configuration file.

> **Note:** By default the level is set to `INFO` for all packages of Oracle ADF. However, Oracle recommends `level="FINE"` for detailed logging diagnostics.

For the packages `oracle.adf.view.faces` and `oracle.adfinternal.view.faces`, edit:

```
<logger name="oracle.adf" level="INFO"/>
<logger name="oracle.adfinternal" level="INFO"/>
```

For the Oracle ADF Model layer packages, edit these elements:

```
<logger name="oracle.adf" level="INFO"/>
<logger name="oracle.jbo" level="INFO"/>
```

Alternatively, you can create a debug configuration in JDeveloper that you can choose when you start a debugging session.

To create an Oracle ADF Model debugging configuration:

1. In the Application Navigator, double-click the user interface project.
2. In the Project Properties dialog, click **Run/Debug** and create a new run configuration, for example, named ADF debugging.
3. Double-click the new run configuration to edit the properties.
4. In the Edit Run Configuration dialog, for **Launch Settings**, enter the following **Java Options** for the default **ojvm** virtual machine:

   ```
   -Djbo.debugoutput=adflogger -Djbo.adflogger.level=FINE
   ```

   Oracle recommends the `level=FINE` for detailed diagnostic messages.

16.4.3 Understanding the Different Kinds of Breakpoints

You first need to understand the different kinds of breakpoints and where to create them.

To see the Debugger Breakpoints window, use the **View | Debugger > Breakpoints** menu choice from the main JDeveloper menu, or optionally the key accelerator for this: [`Ctrl`]+[`Shift`]+[`R`].

You can create a new breakpoint by selecting the **New Breakpoint** menu choice from the right-mouse menu anywhere in the breakpoints window. The **Breakpoint Type** dropdown list controls what kind of breakpoint you will create. The valid choices are:

- **Exception** — break whenever an exception of this class (or a subclass) is thrown.

  This is great when you don't know where the exception occurs, but you know what kind of exception it is (e.g. `java.lang.NullPointerException`, `java.lang.ArrayIndexOutOfBoundsException`, `oracle.jbo.JboException`, etc.) The checkbox options allow you to control whether to break on caught or uncaught exceptions of this class. The (Browse...) button helps you find the fully-qualified class name of the exception. The Exception Class combobox remembers most recently used exception breakpoint classes. Note that this is the default breakpoint type when you create a breakpoint in the breakpoints window.

- **Source** — break whenever a particular source line in a particular class in a particular package is run.

  You rarely create a source breakpoint in the New Breakpoint window. This is because it's much easier to create it by first using the **Navigate | Go to Class** menu (accelerator [`Ctrl`]+[`Shift`]+[`Minus`]), then scrolling to the line number you want -- or using **Navigate | Go to Line** (accelerator [`Ctrl`]+[`G`]) -- and finally clicking in the breakpoint margin at the left of the line you want to break on. This is equivalent to creating a new source breakpoint, but it means you don't have to type in the package, class, and line number by hand.

- **Method** — break whenever a method in a given class is invoked.

  This is handy to set breakpoints on a particular method you might have seen in the call stack while debugging a problem. Of course, if you have the source you can set a source breakpoint wherever you want in that class, but this kind of breakpoint lets you stop in the debugger even when you don't have source for a class.

- **Class** — break whenever any method in a given class is invoked.

  This can be handy when you might only know the class involved in the problem, but not the exact method you want to stop on. Again, this kind of breakpoint does not require source. The **Browse** button helps you quickly find the fully-qualified class name you want to break on.

- **Watchpoint** — break whenever a given field is accessed or modified.

  This can be super helpful to find a problem if the code inside a class modifies a member field directly from several different places (instead of going through setter or getter methods each time). You can stop the debugger in its tracks when any field is modified. You can create a breakpoint of this type by using the **Toggle Watchpoint** menu item on the right-mouse menu when pointing at a member field in your class' source.

16.4.4 Editing Breakpoints to Improve Control

After creating a breakpoint you can edit the breakpoint in the breakpoints window by selecting **Edit** in the context menu on the desired breakpoint.

Some really interesting features you can use by editing your breakpoint are:

- Associate a logical "breakpoint group" name to group this breakpoint with others having the same breakpoint group name. Breakpoint groups make it easy to enable/disable an entire set of breakpoints in one operation.
- Associate a debugger action to occur when the breakpoint is hit. The default action is to just stop the debugger so you can inspect things, but you can add a beep, write something to a log file, and enable or disable group of breakpoints.
- Associate a conditional expression with the breakpoint so that it the debugger only stops when that condition is met. In 10.1.3, the expressions can be virtually any boolean expression, including:
  - `expr ==value`
  - `expr.equals("value")`
  - `expr instanceoffully.qualified.ClassName`

  Note: Use the debugger watch window to evaluate the expression first to make sure its valid.

16.4.5 Filtering Your View of Class Members

An excellent but often overlooked feature of the JDeveloper debugger is the ability to filter the members you want to see in the debugger window for any class. In the debugger's Data window, pointing at any item and selecting Object Preferences from the right-mouse context menu brings up a dialog that lets you customize which members appear in the debugger and (more importantly sometimes) which members *don't* appear.

These preferences are set by class type and can really simplify the amount of scrolling you need to do in the debugger data window. This is especially useful while debugging when you might only be interested in a handful of a class' members.

16.4.6 Communicating Stack Trace Information to Someone Else

If you are unable to determine what the problem is and resolve it yourself, typically your next step is to ask someone else for assistance. Whether you post a question in the OTN JDeveloper Discussion Forum or open a Service Request on Metalink, including the stack trace information in your posting is extremely useful to anyone who will need to assist you further to understand exactly where the problem is occurring.

JDeveloper's Stack window makes communicating this information easy. Whenever the debugger is paused, you can view the Stack window to see the program flow as a stack of method calls that got you to the current line. Using the right-mouse Preferences menu on the Stack window background, you can set the Stack window preference to include the Line number information as well as the class and method name that will be there by default. Finally, the other useful context menu option Export lets you save the current stack information to an external text file whose contents you can then post or send to whomever might need to help you diagnose the problem.

16.5 Debugging the Oracle ADF Model Layer

The processing of your JSF page in combination with Oracle ADF Model is controlled by two classes:

- `oracle.adf.controller.faces.lifecycle.FacesPageLifecycle` class
- `oracle.adf.controller.v2.lifecycle.PageLifecycleImpl` class

`FacesPageLifecycle` implements certain methods of `PageLifecycleImpl` to provide customized error-handling behavior for ADF Faces applications. Generally, however, you will set breakpoints on `PageLifecycleImpl`, as this class provides the starting point for creating the objects of the Oracle ADF binding context.

> **Tip:** The `FacesPageLifecycle` class provides the default implementation of the phase of the ADF Lifecycle. A good place to set a breakpoint is on the `prepareModel()` method, as it initiates the first phase of the ADF lifecycle. For details about the Oracle ADF lifecycle, see Section 6.2.3, "What Happens at Runtime: The JSF and ADF Lifecycles".

The successful interaction between the web page and these objects of the Oracle ADF binding context ensures that the page's components display with correct and complete data, that methods and actions produce the desired result, and that the page renders properly with the appropriate validation errors.

16.5.1 Correcting Failures to Display Pages

At runtime, several things must happen before the ADF lifecycle can prepare the model and display the web page. When the first request for an ADF databound web page occurs, the servlet registers the Oracle ADF servlet filter `ADFBindingFilter`, named in the `web.xml` file. The method `ADFBindingFilter.doFilter()` sets up the ADF processing state, and the method `ADFBindingFilter.initializeBindingContext()` creates an instance of `oracle.adf.model.BindingContext` by reading the `CpxFileName` init parameter from the `web.xml` file.

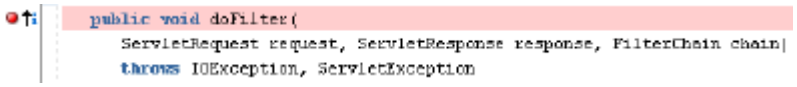
16.5.1.1 Fixing Binding Context Creation Errors

Immediately after `ADFBindingFilter.initializeBindingContext()` is called, `BindingContext` is an empty container object that will define a hierarchy of the Oracle ADF Model layer objects. However, as the container object, `BindingContext` must exist in order for the page's binding to be created. If it does not, an internal servlet error for the Container `/oracle/<path>/DataBinding.cpx` will be thrown:

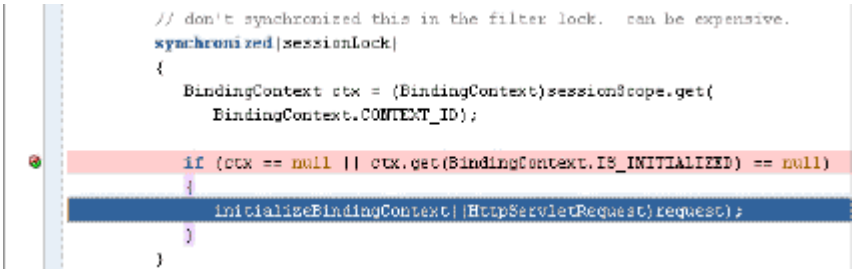
```
oracle.jbo.NoXMLFileException: JBO-26001: XML File not found
```

To debug creating the binding context for the web application:

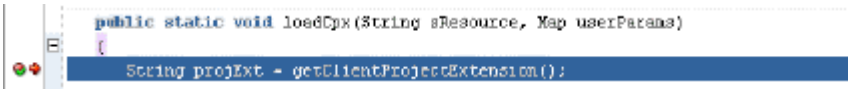
1. In the `oracle.adf.model.servlet.ADFBindingFilter` class, set a break on `chain.doFilter()` and step into this method.



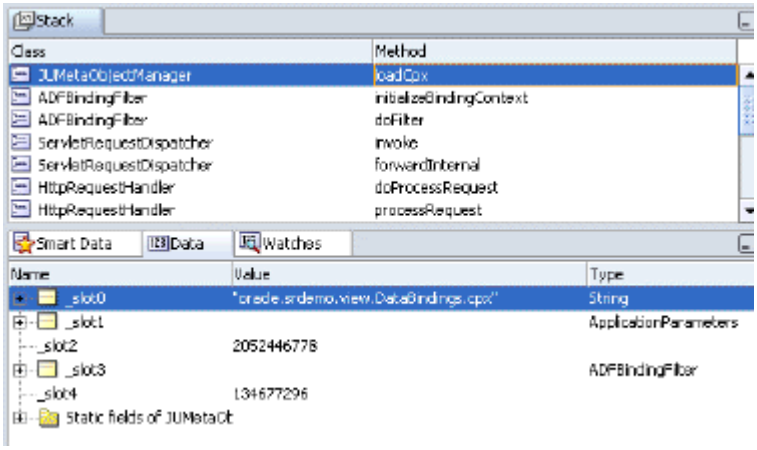
2. Set another break on `ctx.get(BindingContext.IS_INITIALIZED)` and step into this method.



3. In the `oracle.jbo.uicli.mom.JUMetaObjectManager` class, set a break on `chain.getClientProjectExtension()` and step into this method.



4. When processing pauses, look in **slot0** for a file with the expected package name in the Data window.



If the `DataBindings.cpx` file is not found, then check that the servlet context parameter element correctly defines the fully qualified name for the `.cpx` file and verify that the file exists in your project in the location specified by the qualified name path. Example 16–2 shows the context parameter for the SRDemo application.

> **Tip:** The name specified in the `param-value` element of the context parameter must be the fully qualified name of the `.cpx` file.

Example 16–2 Sample web.xml Servlet Context Parameter

```
<context-param>
        <param-name>CpxFileName</param-name>
        <param-value>oracle.srdemo.view.DataBindings</param-value>
</context-param>
```

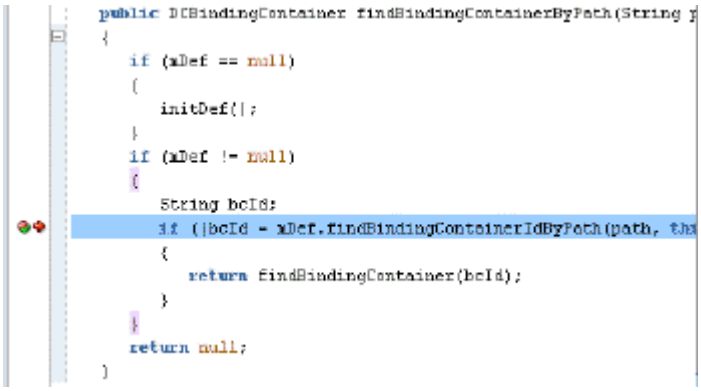
16.5.1.2 Fixing Binding Container Creation Errors

After `BindingContext` is created by `ADFBindingFilter`, the method `PageLifeCycle.xXX()` passes the request's web page URL to the method `BindingContext.findBindingContainer()` to find a page definition from the `<pageMap>` element in the `DataBindings.cpx` file that matches the web page. This becomes the `BindingContainer`. This `BindingContainer` object is the runtime instance object with all bindings created on it. If page definition file is not found, an internal servlet error will be thrown:

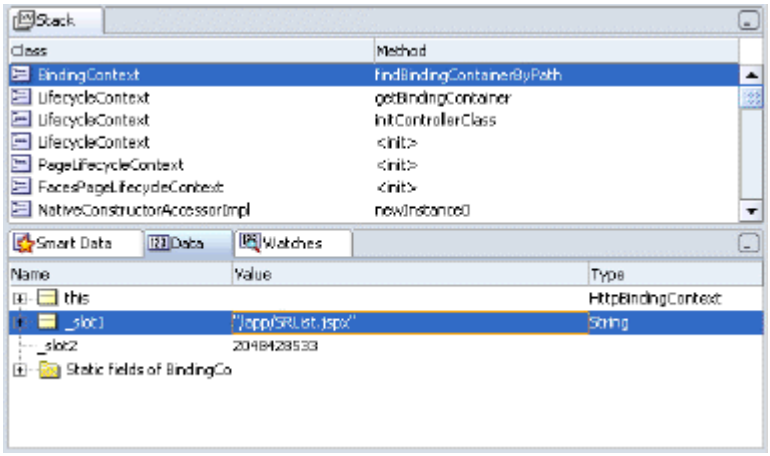
```
oracle.jbo.NoDefException: JBO-25002: Definition
oracle.<path>.pageDefs.<pagedefinitionName> of type Form Binding
Definition not found
```

To debug creating the binding container for the web page:

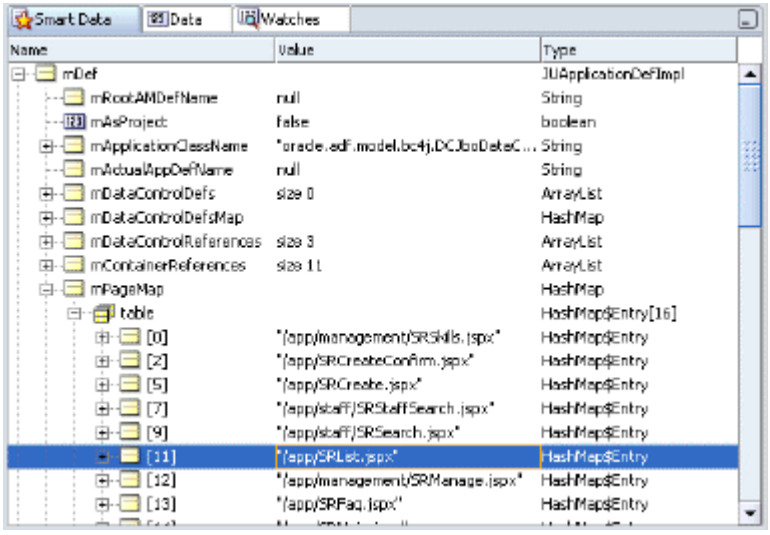
1. In the `oracle.adf.model.BindingContext` class, set a break on `findBindingContainerIdByPath()` and step into this method.



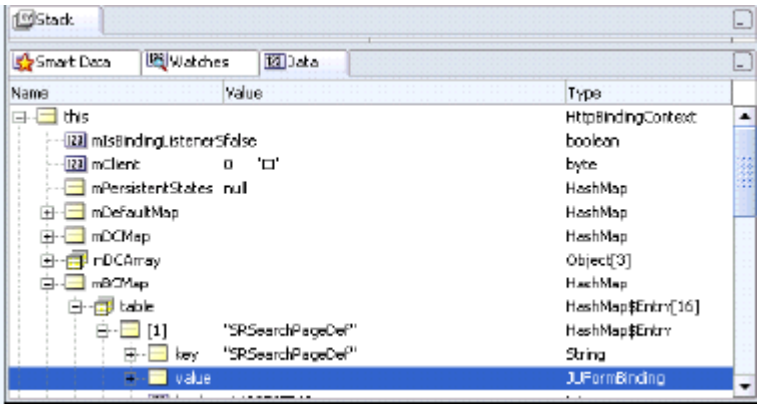
2. Look for the name of the databound web page associated with the binding container in the Data window.



3. In the Smart Data window, look for a matching entry for the expected databound web page file name.



4. In the Data window, there should be a matching page definition entry for the databound web page.



If the *<pagename>*`PageDef.xml` file is not found, then check that the `<pageMap>` element in the `DataBindings.cpx` file specifies the correct name and path to the web page in your project. Example 16–3 shows a sample `DataBindings.cpx` file for the SRDemo application. Notice that the `<pageMap>` element maps the JSF page to its page definition file

---

CAUTION: If you change the name of a JSF page or a page definition file, the `.cpx` file is *not* automatically refactored. You must manually update the page mapping in the `.cpx` to reflect the new page name.

Example 16–3 Sample Databinding.cpx Page Definitions

```
<?xml version="1.0" encoding="UTF-8" ?>
<Application xmlns="http://xmlns.oracle.com/adfm/application"
             version="10.1.3.34.12" id="DataBindings" SeparateXMLFiles="false"
             Package="oracle.srdemo.view" ClientType="Generic">
  <pageMap>
    <page path="/app/SRList.jspx" usageId="app_SRListPageDef"/>
    ...>
  </pageMap>
  <pageDefinitionUsages>
    <page id="SRListPageDef" path="oracle.srdemo.view.pageDefs.
          app_SRListPageDef"/>
    ...
  </pageDefinitionUsages>
  <dataControlUsages>
    <dc id="SRDemoFAQ" path="oracle.srdemo.faq.SRDemoFAQ"/>
    <dc id="SRAdminFacade" path="oracle.srdemo.model.SRAdminFacade"/>
    <dc id="SRPublicFacade" path="oracle.srdemo.model.SRPublicFacade"/>
  </dataControlUsages>
</Application>
```

16.5.2 Correcting Failures to Display Data

After `BindingContainer` is created by `BindingContext`, the ADF lifecycle initiates the Prepare Model and the Render Model phases before data can be displayed in the web page. Several things must happen before the bindings are resolved and data can appear in the web page:

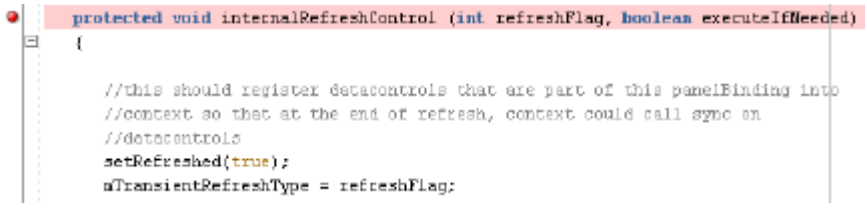
- Page parameters must be set.
- Iterator and Method executables must be get refreshed by executing named service methods and ADF iterator bindings.

16.5.2.1 Fixing Executable Errors

The ADF lifecycle enters the Prepare Model phase by calling `BindingContainer.refresh(PREPARE_MODEL)`. During the Prepare Model phase, `BindingContainer` page parameters get prepared and then evaluated. Next, `BindingContainer` executables get refreshed based on the order of entry in the `pagedef.xml` file's `<executables>` section and on the evaluation of their `Refresh` and `RefreshCondition` properties (if present). When an executable leads to an iterator binding refresh, the corresponding data control will be executed, and that leads to execution of one or more collections in the service objects. If an iterator binding fails to refresh, a JBO exception will be thrown and the data will not be available to display.

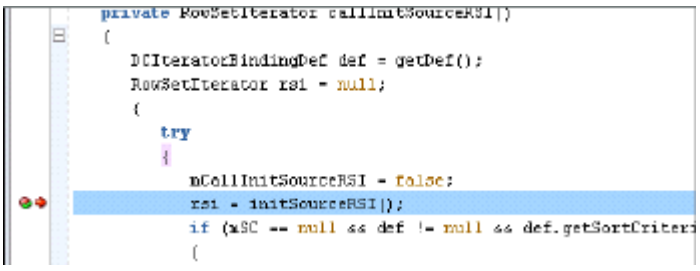
To debug all executables for the binding container:

1. In the `oracle.adf.model.binding.DCBindingContainer` class, set a break on `internalRefreshControl(int, boolean)` as the entry point to debug the executables.

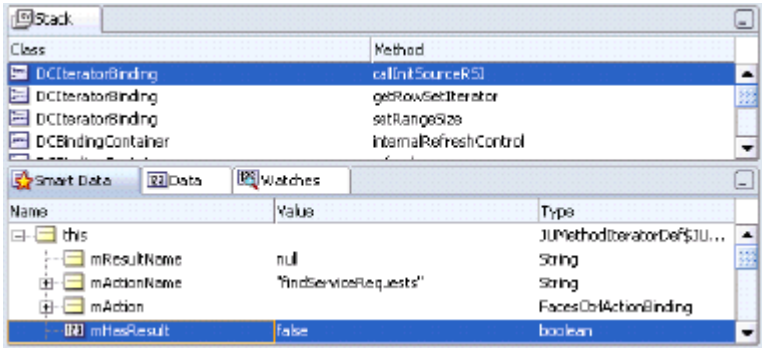


Tip: In the `DCBindingContainer.internalRefreshControl()` method, you can determine whether the executable will be refreshed by checking the outcome of the condition `if (/*execute ||*/ execDef == null || execDef.isRefreshable(this, iterObj, refreshFlag))`. If the condition evaluates to true, then the executable is refreshed and processing will continue to `initSourceRSI()`.

2. In the `oracle.adf.model.binding.DCIteratorBinding` class, set a break on `callInitSourceRSI()` to halt processing and step into the method.



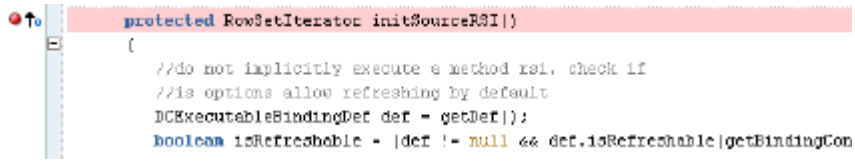
3. When processing pauses, look for `callInitSourceRSI()` in the Stack window. The result displayed in the Smart Data window should show the result that you expect.



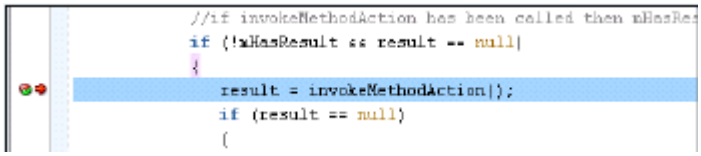
When your web page fails to display data from a method iterator binding, you can drill down to the entry point in `JUMethodIteratorDef.java` and its nested class `JUMethodIteratorBinding` to debug its execution.

To debug the method iterator executable for the binding container:

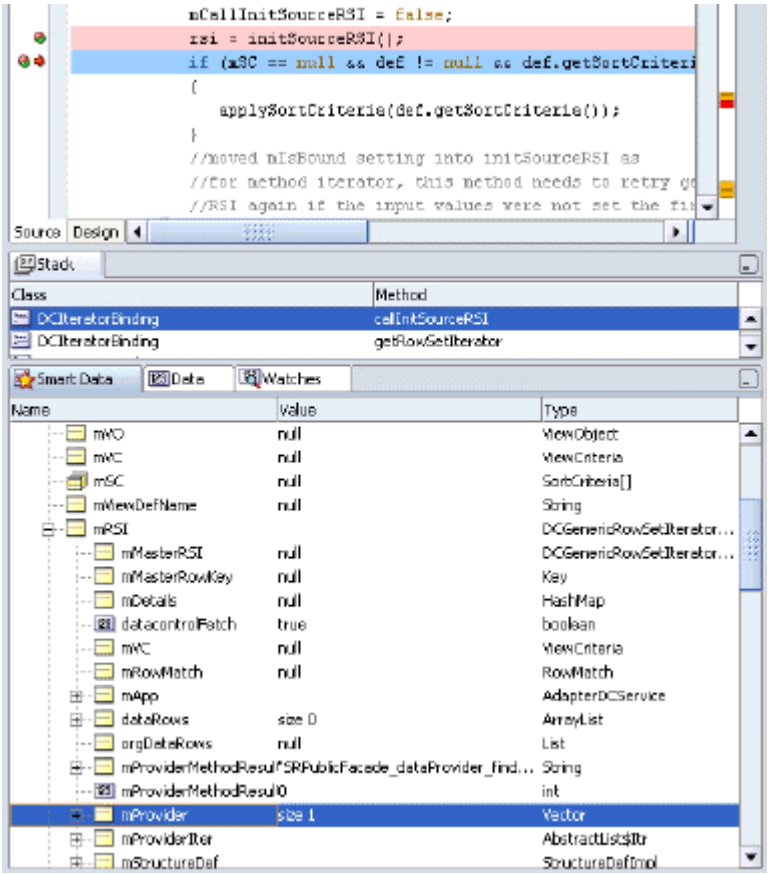
1. In the `oracle.jbo.uicli.binding.JUMethodIteratorDef` class, set a break on `initSourceRSI()` as the entry point to debug a method iterator binding executable.



2. Set a break on `invokeMethodAction()` to halt processing and step into the method.



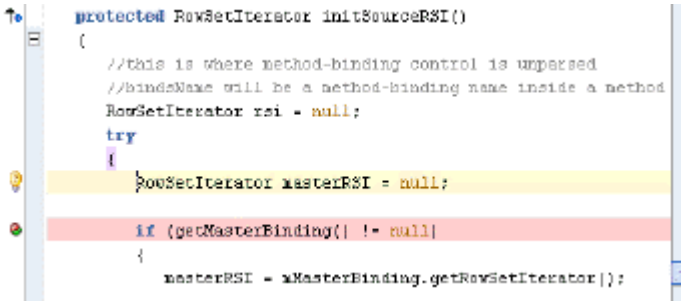
3. When `initSourceRSI()` returns a rowset iterator, pause processing and look for **mProvider** in the Smart Data window. The mProvider variable is the datasource fetched for this rowset iterator. If the method returned successfully, it should show a collection bound to an iterator or a bean.



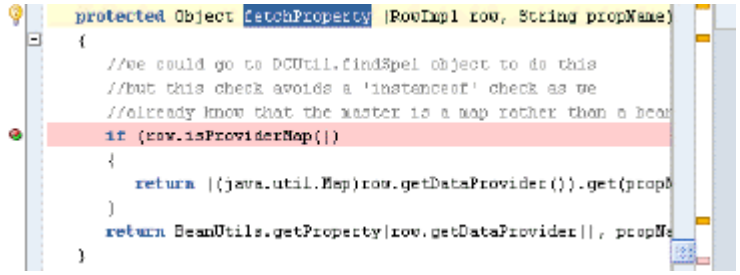
When your web page fails to display the detail data from an accessor binding, you can drill down to the entry point in `JUAccessorIteratorDef.java` to debug its execution.

To debug only the accessor binding executable for the binding container:

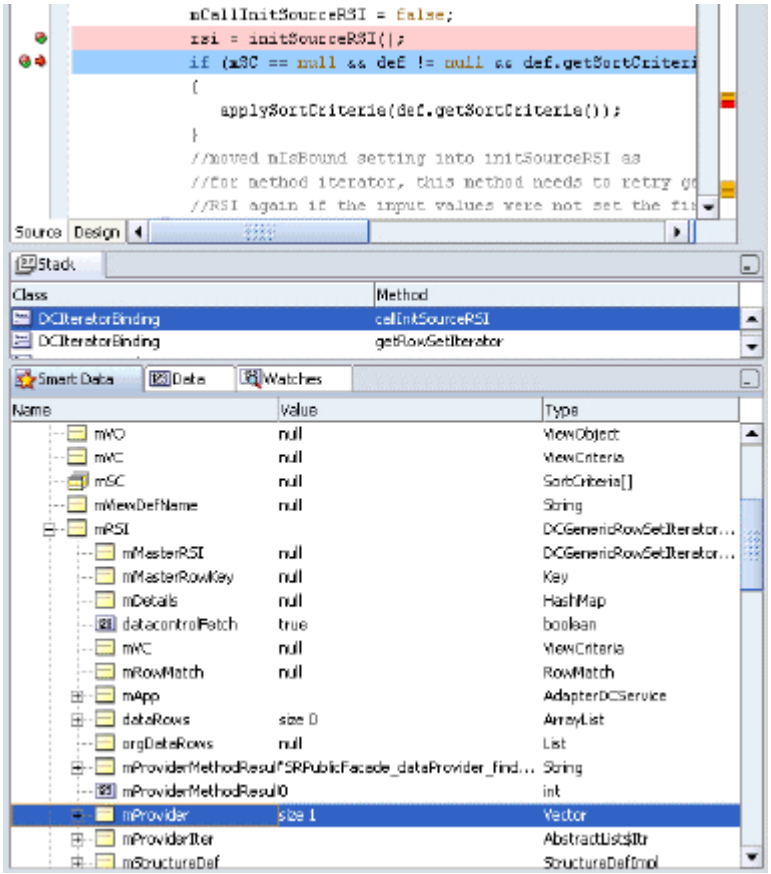
1. In the `oracle.jbo.uicli.binding.JUAccessorIteratorDef` class, set a break in `initSourceRSI()` as the entry point to debug an accessor executable.



2. In the `oracle.adf.model.generic.DCGenericDataControl` class, set a break in `fetchProperty(RowImpl row, String propName)` to halt processing before looking into the Data window. Check if the method returns any property that is a collection, iterator, or a bean.



3. When `initSourceRSI()` returns a rowset iterator, pause processing and look for `callInitSourceRSI()` in the Smart Data window. The result should show the collection that you expect.



Tip: If the debugger does not reach a breakpoint that you set on an executable in the binding container, then the error is most likely a result of the way the executable's `Refresh` and `RefreshCondition` attribute was defined. Examine the attribute definition. For details about the `Refresh` and `RefreshCondition` attribute values, see Section A.7.1, "PageDef.xml Syntax".

When the executable that produced the exception is identified, check that the `<executables>` element in the `pagedef.xml` file specifies the correct attribute settings.

Whether the executable is refreshed during the Prepare Model phase, depends on the value of `Refresh` and `RefreshCondition` (if they exist). If `Refresh` is set to `prepareModel`, or if no value is supplied (meaning it uses the default, `ifneeded`), then the `RefreshCondition` attribute value is evaluated. If no `RefreshCondition` value exists, the executable is invoked. If a value for `RefreshCondition` exists, then that value is evaluated, and if the return value of the evaluation is `true`, then the executable is invoked. If the value evaluates to `false`, the executable is not invoked. The default value always enforces execution.

Example 16–4 shows a sample `pagedef.xml` file from the SRDemo application. Notice that the `<executables>` element lists the executables in the order in which they should be executed, with the accessor iterator positioned after its master binding iterator.

Example 16–4 Sample Page Definition Master and Detail Executables

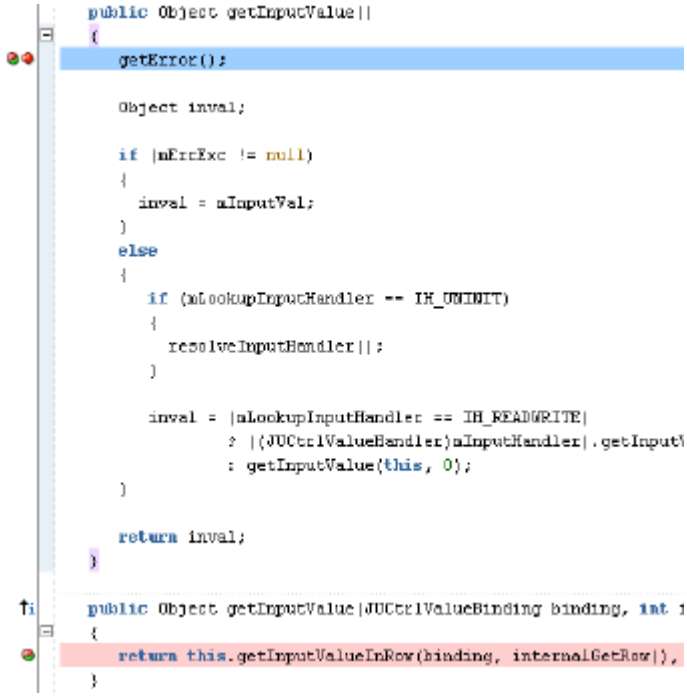
```
<executables>
      <methodIterator id="findAllServiceRequestIter"
                      Binds="findAllServiceRequest.result"
                      DataControl="SRPublicFacade" RangeSize="10"
                      BeanClass="oracle.srdemo.model.ServiceRequest"/>
      <accessorIterator id="serviceHistoryCollectionIterator" RangeSize="10"
                        Binds="serviceHistoryCollection"
                        DataControl="SRPublicFacade"
                        BeanClass="oracle.srdemo.model.ServiceHistory"
                        MasterBinding="findAllServiceRequestIter"/>
</executables>
```

16.5.2.2 Fixing Render Value Errors Before Submit

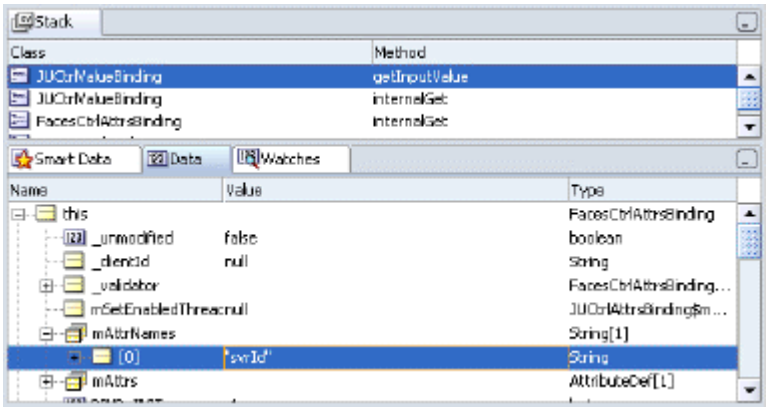
During the prepareRender phase of the ADF lifecycle, the bindings determine the data to display, and properties on the bindings determine the conditions in which to display the data. When the web page is rendered the first time, each EL expression that points to a binding gets resolved by the `BindingContainer` instance for that page. Based on the expression appropriate values like `format`, `isEnabled`, and `isViewable`, the data value for a binding is returned from `BindingContainer`. If the binding is unable to return the data, a JBO exception is thrown.

To debug the binding resolution for the binding container:

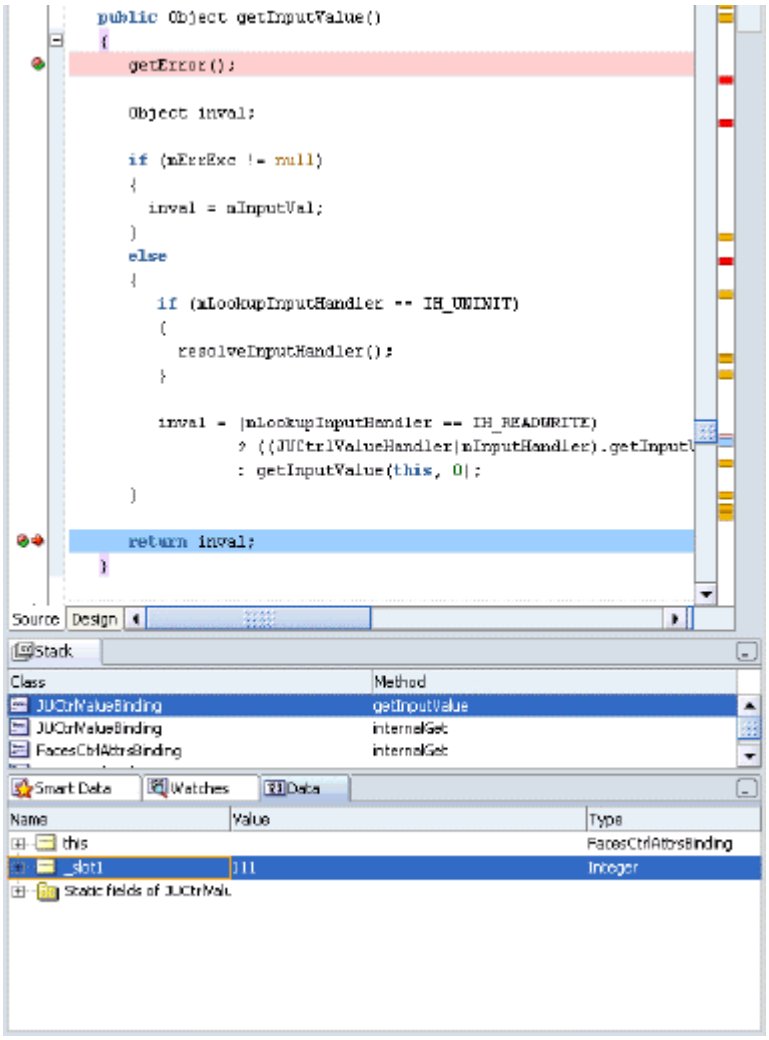
1. In the `oracle.jbo.uicli.binding.JUCtrlValueBinding` class, set a break in `getInputValue()` and step into the method.



2. If `getInputValue()` returns an error, pause processing and look for the binding name in the Data window.



3. Continue stepping into `getInputValue()`, and look for a return value in the Data window that you expect for the current row that this binding represents.



When the binding that produced the exception is identified, check that the `<bindings>` element in the `pagedef.xml` file specifies the correct attribute settings. Example 16–5 shows a sample `pagedef.xml` file for the SRDemo application.

Example 16–5 Sample Page Definition Value Bindings

```
<bindings>
    ...
    <attributeValues id="name" IterBinding="variables">
      <AttrNames>
        <Item Value="findUsersByName_name"/>
      </AttrNames>
    </attributeValues>
    <attributeValues id="email" IterBinding="findUsersByNameIter">
      <AttrNames>
        <Item Value="email"/>
      </AttrNames>
    </attributeValues>
    <attributeValues id="lastName" IterBinding="findUsersByNameIter">
      <AttrNames>
        <Item Value="lastName"/>
      </AttrNames>
    </attributeValues>
    <table id="UserexpertiseAreas" IterBinding="expertiseAreasIterator">
      <AttrNames>
        <Item Value="expertiseLevel"/>
        <Item Value="product"/>
      </AttrNames>
    </table>
</bindings>
```

In case of submit, again, the lifecycle first looks up and prepares the `BindingContainer` instance. If the lifecycle finds a state token that was persisted for this `BindingContainer`, it asks the `BindingContainer` to process this state token. Processing the state token restores the variable values that were saved out in previous the render. If you need to debug processing the state token, break in `DCIteratorBinding.processFormToken()` and `DCIteratorBinding.buildFormToken()`.

After this, all posts are applied to the bindings through `setInputValue()` on the value bindings.

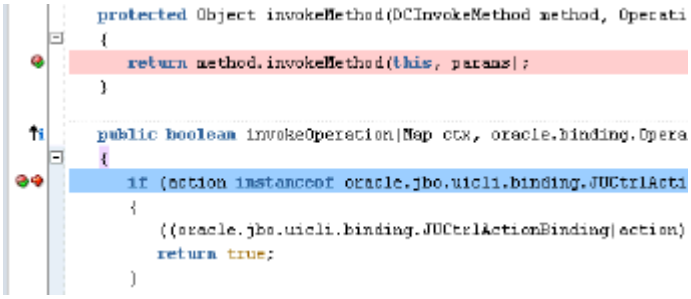
16.5.3 Correcting Failures to Invoke Actions and Methods

When the executables are refreshed, actions and custom methods may be invoked on the page. At this stage, the corresponding action or method binding is refreshed. If an executable or its target binding is not executed, the action will be ignored.

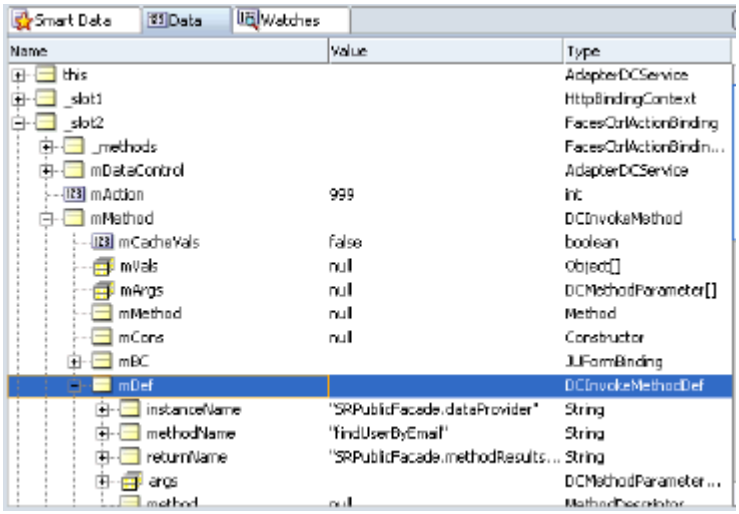
The entry point for action and method execution is the `DCDataControl.invokeOperation()` method. Although `JUCtrlActionBinding.invoke()` is another potential entry point, method iterator bindings also use it to invoke methods implicitly. Instead, debugging on `DCDataControl.invokeOperation()` allows you to work with the same method that the data control uses to invoke the method. This is preferred because some adapter data controls can interpret the method name in a custom way rather than leave it to ADF to call the method.

To debug the action or method invocation for the binding container:

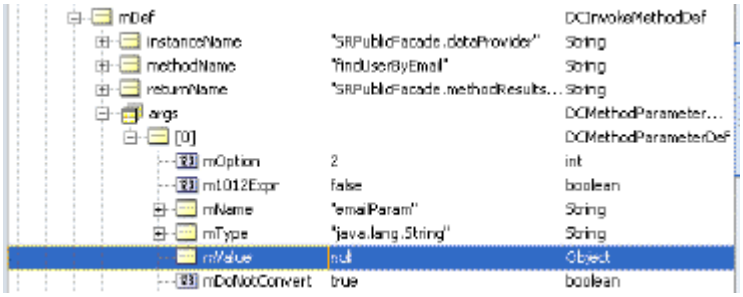
1. In the `oracle.adf.model.binding.DCDataControl` class, set a break on `invokeOperation()` as the entry point to debug an action or method invocation.



2. When processing pauses, step though the method to verify **instanceName** in the Data window shows the method being invoked is the intended method on the desired object.

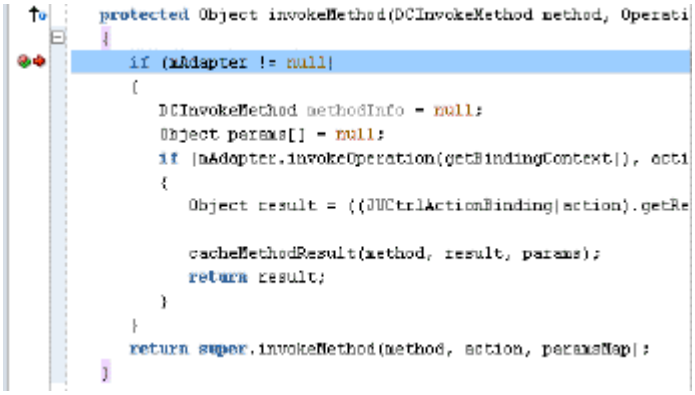


3. Verify **args** in the Data window shows the parameter value for each parameter being passed into your method is as expected. The parameter value below shows null.

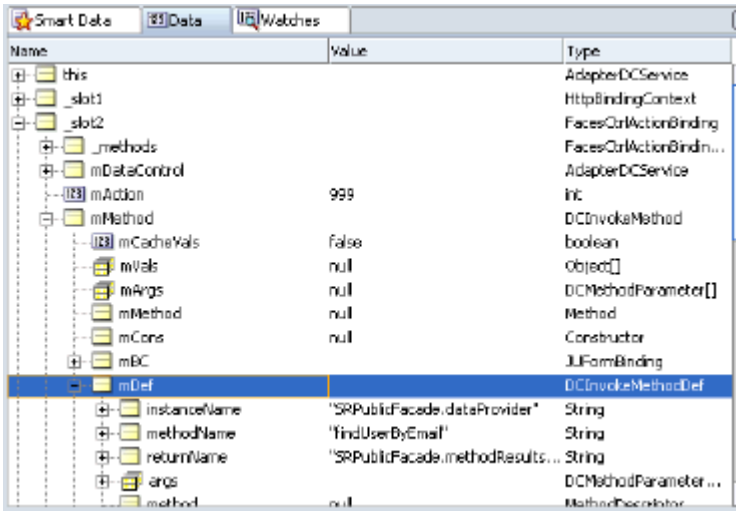


To debug a custom method invocation for the binding container:

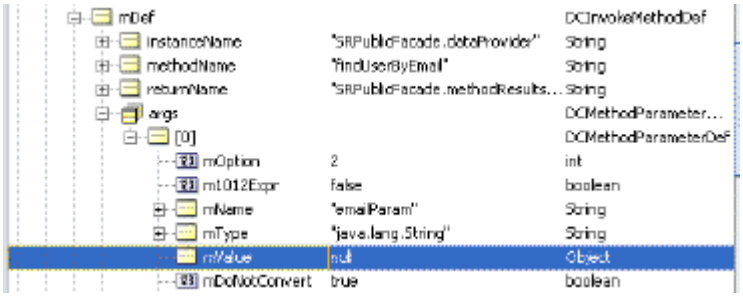
1. In your class, set a breakpoint on the desired custom method.
2. In `oracle.adf.model.generic.DCGenericDataControl` class, set a break on `invokeMethod()` to halt processing before looking into the Data window.



3. When processing pauses, step though the method to verify **instanceName** in the Data window shows the method being invoked is the intended method on the desired object.



4. Verify **args** in the Data window shows the parameter value for each parameter being passed into your method is as expected. The parameter value below shows null.



When the ignored action or custom method is identified, check that the `<invokeAction>` definitions in `<executables>` element and their corresponding `<action>` and `<methodAction>` definitions in the `<bindings>` element of the `pagedef.xml` file specifies the correct attribute settings.

> **Tip:** If the debugger does not reach a breakpoint that you set on an action in the binding container, then the error is most likely a result of the way the executable's `Refresh` and `RefreshCondition` attribute was defined. Examine the attribute definition. For details about the `Refresh` and `RefreshCondition` attribute values, see Section A.7.1, "PageDef.xml Syntax".

Whether the `<invokeAction>` executable is refreshed during the Prepare Model phase, depends on the value of `Refresh` and `RefreshCondition` (if they exist). If `Refresh` is set to `prepareModel`, or if no value is supplied (meaning it uses the default, `ifneeded`), then the `RefreshCondition` attribute value is evaluated. If no `RefreshCondition` value exists, the executable is invoked. If a value for `RefreshCondition` exists, then that value is evaluated, and if the return value of the evaluation is `true`, then the executable is invoked. If the value evaluates to `false`, the executable is not invoked. The default value always enforces execution.

Example 16–6 shows a sample of the action and custom method binding definition in the `pagedef.xml` file for the SRDemo application.

Example 16–6 Sample Page Definition Executables and Action Bindings

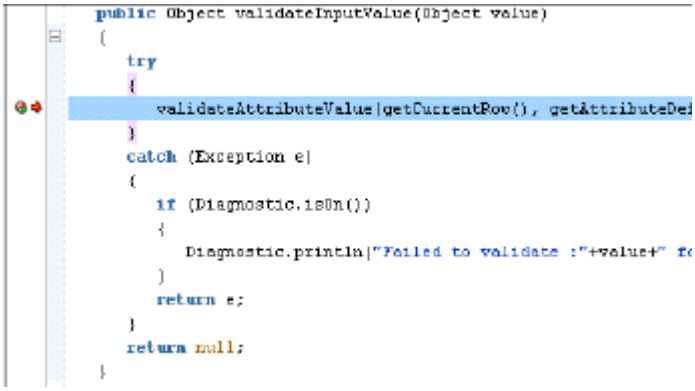
```
<executables>
    ...
    <invokeAction Binds="findServiceRequests" id="tableRefresh"
                  Refresh="ifNeeded"
                  RefreshCondition="${(userState.refresh) and
                    (!adfFacesContext.postback)}"/>
    ...
</executables>
<bindings>
    <methodAction id="findServiceRequests"
                  InstanceName="SRPublicFacade.dataProvider"
                  DataControl="SRPublicFacade"
                  MethodName="findServiceRequests" RequiresUpdateModel="true"
                  Action="999"
                  ReturnName="SRPublicFacade.methodResults.SRPublicFacade_
                                  dataProvider_findServiceRequests_result">
       <NamedData NDName="userIdParam" NDValue="#{userInfo.userId}"
                  NDType="java.lang.Integer"/>
       <NamedData NDName="statusParam" NDValue="#{userState.listMode}"
                  NDType="java.lang.String"/>
   </methodAction>
    ...
</bindings>
```

16.5.4 Correcting Page Validation Failures

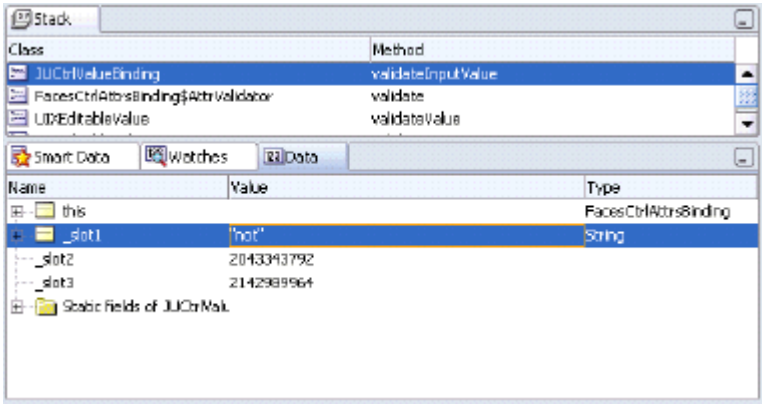
The method `validate()` on the `BindingContainer` gets called, which calls `validateInputValue()` on each of the bindings referred to in this `BindingContainer`. If the validation set on an input field fails to behave as expected, then no validation error message will be displayed in the web page.

To debug validation-checking failures for the binding container:

1. In `oracle.jbo.uicli.binding.JUCtrlValueBinding` class, set a break in `validateInputValue(Object value)` to halt processing before looking into the Data window.



2. When processing pauses, look for **slot1** in the Data window and confirm that the validation is performed. The value **not** shown below indicates validation was not performed.



When the validation that failed is identified, check that the validation rule for the value binding is correctly defined and that the input field component's `<af:validator>` tag is bound to the same attribute defined by the value binding. Example 16–7 shows a sample validation rule in the `pagedef.xml` file for the SRDemo application.

Notice that the ADF Model validation rule should appear on the attribute binding. For details about working with validation rules, see Section 12.3, "Adding Validation".

> **Tip:** To process ADF Model layer validation, the Faces validator tag must be bound to the associated attribute's validator property. For example:
>
> `<af:validator binding="#{bindings.<someattribute>.validator}"/>`
>
> where *`<someattribute>`* would be `createProducts_description` to work with the sample validation rule shown in Example 16–7.

Example 16–7 Reference to Validation Rule in Page Definition File

```
<attributeValues id="description" IterBinding="variables" ApplyValidation="true">
  <LengthValidationBean xmlns="http://xmlns.oracle.com/adfm/validation"
            OnAttribute="createProducts_description"
            DataType="CHARACTER" CompareType="LESSTHAN"
            ResId="description_Rule_0" Inverse="false"
            CompareLength="20"/>
  <AttrNames>
     <Item Value="createProducts_description"/>
  </AttrNames>
</attributeValues>
```

16.6 Tracing EL Expressions

EL is not well supported with exceptions to inform you of specific failures. However, Example 16–8 shows one common exception you are likely to see when the resolver is unable to completely evaluate the expression.

Example 16–8 Expression Evaluation PropertyNotFound Exception

```
javax.faces.el.PropertyNotFoundException:
   Error setting property 'resultsTable' in bean of type null
at com.sun.faces.el.PropertyResolverImpl.setValue
   (PropertyResolverImpl.java:153)
```

You can check your web page's source code for problems in the expression, such as mistyped property names. When no obvious error is found, you will want to configure the `logging.properties` file in the *<JDeveloper_Install>*`/jre/lib` directory to display messages from the EL resolver.

To trace EL expression variables:

1. Open *<JDeveloper_Install>*`/jre/lib/logging.properties` in your text editor.
2. Set `java.util.logging.ConsoleHandler.level=FINE`.
3. Add the line:

   ```
   com.sun.faces.level=FINE
   ```

4. Run your application and view the variable resolution in the JDeveloper Log window.

For example, the SRDemo application defines a backing bean `backing_SRSearch.java`. Example 16–9 shows the `SRSearch.jspx` page, which relies on the ADF table binding `resultsTable` to create a databound table component.

Example 16–9 Reference to Backing Bean in Table Binding

```
<af:table rows="#{bindings.findAllServiceRequests1.rangeSize}"
                      ...
                      binding="#{backing_SRSearch.resultsTable}"
                      id="resultsTable"
                      width="100%"
                      rendered="#{(bindings.hideResultsParam!='true') and
                        (bindings.findAllServiceRequests1.estimatedRowCount >0)}">
```

Example 16–10 shows the messages that appear in the JDeveloper Log window when you run the application with EL trace messages enabled. In this case, the resolver is not able to resolve the value binding `resultsTable` from the backing bean and the `PropertyNotFound` exception will appear in the browser.

Example 16–10 JDeveloper Log with EL Trace Enabled

```
02-Dec-2005 09:41:28 com.sun.faces.el.ValueBindingImpl getValue
FINE: getValue(ref=backing_SRSearch.resultsTable)
02-Dec-2005 09:41:28 com.sun.faces.application.ApplicationAssociate
   createAndMaybeStoreManagedBeans
FINE: Couldn't find a factory for backing_SRSearch
02-Dec-2005 09:41:28 com.sun.faces.el.VariableResolverImpl resolveVariable
FINE: resolveVariable: Resolved variable:null
02-Dec-2005 09:41:28 com.sun.faces.el.ValueBindingImpl getValue
FINE: getValue Result:null
02-Dec-2005 09:41:28 com.sun.faces.application.ApplicationAssociate
   createAndMaybeStoreManagedBeans
FINE: Couldn't find a factory for backing_SRSearch
02-Dec-2005 09:41:28 com.sun.faces.el.VariableResolverImpl resolveVariable
FINE: resolveVariable: Resolved variable:null
02-Dec-2005 09:41:28 com.sun.faces.el.ValueBindingImpl setValue
FINE: setValue Evaluation threw exception:
javax.faces.el.PropertyNotFoundException
```

The message `FINE: Couldn't find a factory for backing_SRSearch` indicates that the backing bean was never created. To fix the error, check the `faces-config.xml` file and make sure that the backing bean is listed. Example 16–11 shows the correct listing for the file.

Example 16–11 faces-config.xml Managed Bean Description

```
<!-- Page backing beans -->
 <managed-bean>
   <managed-bean-name>backing_SRSearch</managed-bean-name>
   <managed-bean-class>oracle.srdemo.view.backing.SRSearch</managed-bean-class>
   <managed-bean-scope>request</managed-bean-scope>
   <!--oracle-jdev-comment:managed-bean-jsp-link:1SRSearch.jspx-->
 </managed-bean>
```

In summary, when you encounter a `PropertyNotFound` exception and the property is one that appears in an EL expression, you may check the syntax of your web page for simple errors. Then, rerun the application with the JSF trace messages enabled and examine the variable resolution messages for clues.

Part III

Implementing Projects With Oracle ADF

Part III contains the following chapters:

- Chapter 17, "Working Productively in Teams"
- Chapter 18, "Adding Security to an Application"
- Chapter 19, "Advanced TopLink Topics"
- Chapter 20, "Creating Data Control Adapters"
- Chapter 21, "Working with Web Services"
- Chapter 22, "Deploying ADF Applications"

17

Working Productively in Teams

The source control system used for the SRDemo application was CVS. This chapter contains advice for using CVS with ADF projects, and general advice for using CVS with JDeveloper.

This chapter includes the following sections:

- Section 17.1, "Using CVS with an ADF Project"
- Section 17.2, "General Advice for Using CVS with JDeveloper"

17.1 Using CVS with an ADF Project

This section contains advice specifically for using CVS with an ADF project, for example the SRDemo application.

17.1.1 Choice of Internal or External CVS Client

A CVS client lets you import your work into CVS or check it out from CVS control. The CVS client can be a standalone program, or it can be integrated into an IDE, as it is with JDeveloper. The SRDemo application was created using the JDeveloper internal CVS client.

17.1.2 Preference Settings

You set up JDeveloper to use CVS by ensuring that Support for CVS n.n is checked on the Extensions preferences page (**Tools** > **Preferences** | **Extensions** | **Versioning Support n.n** | **Configure**) and that CVS is selected from the dropdown list on the Versioning preferences page (**Tools** > **Preferences** | **Versioning**).

Preferences for using CVS are set by selecting **Tools** > **Preferences** | **Extensions** | **Versioning** | **CVS** and its subpages.

The SRDemo application was created using the default preferences for CVS, although you may want to consider setting the timeout to ten minutes (Operation Timeout on the General subpage), especially if you have a slow connection to a remote server.

17.1.3 File Dependencies

JDeveloper will work with the CVS version control system to keep files within a multi-file component synchronized, for example, by automatically checking out all the files that are dependent on a file that you expressly check out. However, when working with Oracle ADF-base JSP pages, you should be conscious of the dependencies between the various, related artifacts.

For example, when you commit a JSP page like `SomeName.jsp`, if changes you made in JDeveloper have caused the associated `SomeNamePageDef`.xml file to be modified, it will also appear in the **Outgoing** page of the Pending Changes window. On the other hand, if `SomeName.jsp` is a *new* JSP page on which you've dropped some databound controls, its associated `SomeNamePageDef.xml` file will also appear in the **Candidates** page of the Pending Changes window, and the `DataBindings.cpx` file will appear as a modified file in the **Outgoing** page. By understanding these relationships, you can better decide which files need to be committed together as part of the same CVS transaction to ensure that other developers who update their project from the source control server receive a consistent set of related files.

17.1.4 Use Consistent Connection Definition Names

Most JDeveloper and ADF objects will be created only once per project and will by definition have the same name regardless of who sees or uses them. However, some objects like database connections could theoretically be left to the creativity of each team member in their own JDeveloper environment, even though they map to the same connection details. Avoid such naming differences for otherwise common connection definitions when working with ADF under version control since the discrepancy will cause unnecessary differences in your `data-sources.xml` files. Team members should agree up front on a common, case-sensitive connection name and that should be used by every member of the team.

17.1.5 General Advice for Committing ADF Work to CVS

In general, you should commit your work after it has been tested and are satisfied that it is working. The longer you work on a set of components without testing the changes and checking them in, the greater the chances that other developers will have modified them too, resulting in merge conflicts and the need to resolve them.

17.1.5.1 Other Version Control Tips and Techniques

Make sure to have an active CVS connection open in the CVS navigator when you are performing any kind of renaming or refactoring operations. If you do so, these will be automatically handled as appropriate file deletes and adds in the source control system. If you are not in the context of a CVS connection when you make these kinds of changes, then the next time you connect to source control, your renamed files may inadvertently show up as new files.

When renaming files (for example, through refactoring), you should commit the files as soon as practicable after you have renamed them. This is because renaming a file through JDeveloper involves a CVS delete operation and a CVS add operation, and an added file needs to be committed to make it available to other developers. However, you should still test the changes before committing them. A typical scenario would be to refactor the files, then rerun the unit tests, then commit the files.

When developing new features, you may have to depart from the normal rule of unit testing files before committing them, if other members of the team need to work on the files in order to complete the unit of work that is to be tested. In this case, the files will need to be committed before testing so that other members of the team can obtain them from the CVS repository.

When committing work to CVS, always add comments describing the changes you have made. You add comments in the Comments box of the Commit to CVS dialog (**Versioning** > **Commit**).

17.1.6 Check Out or Update from the CVS Repository

It is preferable to perform, at regular intervals, a clean checkout from the CVS repository to a fresh directory (using **Versioning** > **Check Out Module**). Simply updating your working copy from the repository (using **Versioning** > **Update**) can hide problems such as incomplete commits.

You could use Apache Ant, which is integrated into JDeveloper, to create a script that will automatically check out the full source and build it. If the build completes successfully, this will be confirmation that everyone has committed all the changes required to make the system perform correctly. Otherwise, the build will break and problems will be signalled. To find out how to use Apache Ant to create build scripts, search for "About Ant Integration in JDeveloper" in the JDeveloper online help.

17.1.7 Special Consideration when Manually Adding Navigation Rules to the faces-config.xml File

If you manually add navigation rules to the `faces-config.xml` file (using the XML view or the Overview screen), you must switch to the visual diagram view of `faces-config` before checking in the `faces-config.xml` file. Doing so will cause the diagram file (`faces-config.oxd_faces`) to register the metadata change and force it to reflect the rule change. It also ensures that the `faces-config.oxd_faces` file is marked for commit and that the two files will not get out of synchronization.

If you don't do this, the diagram file will no longer be in step with the XML metadata and will give errors. If this happens, the solution is to manually delete the diagram file and let JDeveloper re-create it when it next attempts to open the file. That file is `\model\public_html\WEB-INF\faces-config.oxd_faces` under the `userinterface/viewcontroller` project.

17.2 General Advice for Using CVS with JDeveloper

This section contains advice for using CVS with JDeveloper generally.

17.2.1 Team-Level Activities

Divide the development work between several projects.

Consider using a code formatter, possibly as part of an Apache Ant build script. JDeveloper's code formatter is available from the Code Style page of the Preferences dialog (**Tools** > **Preferences** | **Code Style**). You can use this to create and export a standard format that all team members can import, thus allowing them to share the same built-in code formatting rules.

Build the code before checking it into CVS and before doing a CVS update.

Consider running a continuous integration tool. The tool should rebuild the whole project whenever someone commits changes to the CVS repository and should notify developers when code they have committed breaks the build by requesting that the code be fixed. Running a continuous integration tool will improve confidence in the quality of the code in the CVS repository, encourage developers to update more often, and lead to smaller updates and fewer conflicts. An example of a continuous integration tool is Apache Gump (`http://gump.apache.org/`).

Before importing modules, configure the CVS repository to import binary file types as binary (rather than as text), to prevent them from being corrupted.

17.2.2 Developer-Level Activities

This section contains advice for developers working with files under CVS control.

17.2.2.1 Typical Workflow When Checking Your Work Into CVS

Always perform an update (**Versioning** > **Update**) or module checkout (**Versioning** > **Check Out Module**) before you start editing files to make sure that you are working with the most recent versions.

While you can commit your work one file at a time using the **Versioning > Commit** menu option, Oracle recommends using the Pending Changes window. To show this window, choose **Versioning > Pending Changes** from JDeveloper's main menu. When working in a team, before committing the files you've been working on, you will typically use the Pending Changes window in the following sequence:

- Use the Outgoing Page to add new files to source control.

  First, use the **Outgoing** page to see all of the new files you've created in the current workspace. To be sure the list is as up to date as possible, click the **Refresh** icon in the page toolbar. Decide which of the new files should be added to source control, and select all of these. Finally, use the **Add** option on the context menu to add the selected files to source control. The longer you work on a set of components without testing the changes and checking them in, the greater the chances that other developers will have modified them too thereby resulting in merge conflicts and the need to resolve them.

  > **Tip:** Do not commit the `WEB-INF\temp` directory because this is a directory containing cached images that ADF Faces generates once on demand at runtime.

- Use the Incoming Page to update workspace files from other team members.

  Second, use the **Incoming** panel to review whether any changes made by other developers on your team might affect the work you're about to check in. If other team members may have created files in new directories that you do not yet have in your copy of the project, use the **Update Project Folders** option on the context menu of the workspace or on an individual project to ensure your local working area reflects those new directories. Again, you should click the **Refresh** button to ensure that you're seeing the most up-to-date list of incoming files. If team members have changed files *unrelated* to your work, you can choose to update your copies of those files if useful to you for testing. If they have changed files that are the *same* as ones you have modified, then JDeveloper will show the incoming status as **conflicts on merge**. You need to update the files and address any merge conflicts before the CVS server will allow you to check in.

- Resolve any merge conflicts if necessary.

  After performing an update that encountered merge conflicts, JDeveloper displays an exclamation point next to each conflicting file in the Application Navigator. Also, in the Pending Changes window's **Outgoing** page the outgoing status will be shown as **conflicts**. You can resolve the conflicts using JDeveloper's built-in merge tool. Right-click the file and choose **Resolve Conflicts** from the context menu. Three versions of the file will be shown: on the left will be the version in the CVS repository, on the right will be the current local version, and in the middle will be an editable version that represents the result of the merge. Symbols in the margin between the three panels indicate the suggested action for resolving each conflict.

By selecting an appropriate icon in the margin and using the context menu, you can insert changes from the file on the left side or the right side after the adjacent difference.

Tooltips explain the suggested action of each conflict. You can accept the suggested actions or edit the text directly. To complete the merge, you must save the changes that have been made, using the **Save** button in the merge window's toolbar. If this is not enabled, you may need to use the **Mark as Resolved** or **Mark All As Resolved** options in the context menu. Once you've saved the merged version of the file, the merge tool window becomes blank and JDeveloper removes the conflict symbol from the navigator icon and you will be able to commit the merged file to the CVS repository. You can close the merge tool window and proceed to the next conflict, if any.

- Use the Outgoing Page to commit your changes.

  Finally, use the **Outgoing** page of the Pending Changes window to commit your changes to source control. There may be some files that are modified but which you don't want to commit. For example, each time you run your application on the embedded OC4J server, JDeveloper may refresh the contents of your project's `data-sources.xml` and/or `jazn-data.xml` file. You may not want to keep checking in modified versions of these each time. In addition, there may be files you modified, but whose changes you don't wish to keep. As you can do at any time, you can choose **Versioning** > **Undo Changes** from the context menu for such a file in the Application Navigator. This will revert the file to the latest checked-in version in source control. Finally, select the files you want to check in, and choose **Commit** on the context menu.

  > **Tip:** Be aware that the **Commit All** button on the toolbar of the Pending Changes window will commit all files in the **Outgoing** list. Use the technique described above to commit selected files.

17.2.2.2 Handling CVS Repository Configuration Files

To prevent accidental corruption of the CVS repository, do not change repository configuration files manually. If you need to change a CVS configuration file, check out CVSROOT as a module, modify the specific configuration file locally, and then commit it to the repository.

18

Adding Security to an Application

This chapter describes how to use Oracle ADF Security in your web application to handle authentication and authorization on the Oracle Application Server. It also describes how to bypass Oracle ADF Security when you want to work strictly with container-managed security.

This chapter includes the following sections:

- Section 18.1, "Introduction to Security in Oracle ADF Web Applications"
- Section 18.2, "Specifying the JAZN Resource Provider"
- Section 18.3, "Configuring Authentication Within the web.xml File"
- Section 18.4, "Creating a Login Page"
- Section 18.5, "Creating a Logout Page"
- Section 18.6, "Implementing Authorization Using Oracle ADF Security"
- Section 18.7, "Implementing Authorization Programmatically"

18.1 Introduction to Security in Oracle ADF Web Applications

Web application security can be provided by Oracle ADF Security. The Oracle ADF Security implementation is built upon a pluggable architecture that implements the Oracle Application Server Java Authentication and Authorization (JAAS) Provider for authentication and authorization:

- Authentication provides a way to determine who the current user is. Oracle ADF Security can authenticate users against data within various resource providers.
- Authorization provides a way to restrict access to the application or parts of the application (called resources) based on the user attempting to access the resource. Oracle ADF Security allows you to set authorization on ADF Model layer objects.

First, you must configure the application to use a resource provider. The user data against which the login and passwords are authenticated is stored within a resource provider, such as a database or LDAP director. By editing the `jazn.xml` file, you choose an identity management provider for the OracleAS JAAS Provider. Read the following section to understand editing the `jazn.xml` file:

- Section 18.2, "Specifying the JAZN Resource Provider"

Then, you can configure the application's container to use Oracle ADF Security. This will allow you to use Oracle ADF Security for authentication and authorization. Alternatively, you can bypass Oracle ADF Security and use container-managed security.

Read the following sections to understand how to configure authentication and create login and logout pages:

- Section 18.3, "Configuring Authentication Within the web.xml File"
- Section 18.4, "Creating a Login Page"
- Section 18.5, "Creating a Logout Page"

When you want to assign resources to particular users, you can work with Oracle ADF Model layer to enable authorization. If you choose not to use ADF authorization, you can still work with ADF authentication. Alternatively, you can integrate standard J2EE authorization with the Oracle ADF Model layer to restrict resources. The SRDemo application uses the latter approach. Read the following section to understand both approaches to implementing authorization:

- Section 18.6, "Implementing Authorization Using Oracle ADF Security"
- Section 18.7, "Implementing Authorization Programmatically"

> **Note:** When you want to understand the security features of OC4J, see the *Oracle Containers for J2EE Security Guide* in the Oracle Application Server documentation library. For example, the "Standard Security Concepts" chapter provides a useful overview of the JAAS security model.

18.2 Specifying the JAZN Resource Provider

If you wish to use the JAZN realm from either the lightweight XML resource provider (`system-jazn-data.xml`) or through the Oracle Internet Directory, you need to edit the `jazn.xml` file to select one of those providers.

Note: If you are working with another JAAS-compliant security provider, see your security provider's documentation.

18.2.1 How To Specify the Resource Provider

To use the JAZN realm from either the lightweight XML resource provider (`system-jazn-data.xml`) or through the Oracle Internet Directory (LDAP provider), you need to specify which provider you want your application to work with.

To specify the resource provider, you edit the provider environment descriptor in `jazn.xml`, located in the following directories.

- For JDeveloper's embedded OC4J:

  <JDEV_HOME>`/jdev/system/oracle.j2ee.10.1.3/embedded-oc4j/config` directory

- For JDeveloper's standalone OC4J:

  <JDEV_HOME>`/j2ee/home/config` directory

- For Oracle Application Server:

  <OC4J_HOME>`/j2ee/`*<instance_name>*`/config` directory

To work with the XML-based provider, comment out the environment descriptor for LDAP:

```
<jazn xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:noNamespaceSchemaLocation=
         "http://xmlns.oracle.com/oracleas/schema/jazn-10_0.xsd"
   schema-major-version="10"
   schema-minor-version="0"
   provider="XML"
   location="./system-jazn-data.xml"
   default-realm="jazn.com"
/>

<!--
<jazn
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:noNamespaceSchemaLocation=
         "http://xmlns.oracle.com/oracleas/schema/jazn-10_0.xsd"
   schema-major-version="10"
   schema-minor-version="0"
   provider="LDAP"
   location="ldap://myoid.us.oracle.com:389"
/>
-->
```

To work with the LDAP provider, comment out the environment descriptor for XML:

```
<!--
<jazn
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:noNamespaceSchemaLocation=
         "http://xmlns.oracle.com/oracleas/schema/jazn-10_0.xsd"
   schema-major-version="10"
   schema-minor-version="0"
   provider="XML"
   location="./system-jazn-data.xml"
   default-realm="jazn.com"
/>
-->

<jazn
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:noNamespaceSchemaLocation=
         "http://xmlns.oracle.com/oracleas/schema/jazn-10_0.xsd"
   schema-major-version="10"
   schema-minor-version="0"
   provider="LDAP"
   location="ldap://myoid.us.oracle.com:389"
/>
```

18.2.2 What You May Need to Know About Oracle ADF Security and Resource Providers

Because Oracle ADF Security uses OracleAS JAAS, it relies on the LoginContext to provide the basic methods for authentication. LoginContext uses Login Modules, which are pluggable bits of code that handle the actual authentication. Oracle ADF Security also uses OracleAS JAAS Provider `RealmLoginModule` login module to perform standard user name/password type of authentication.

Oracle ADF Security can authenticate users against a given resource provider. The resource provider, such as a database or LDAP directory, contains the data against which the login and passwords are authenticated.

Specifically, Oracle ADF Security supports the use of Oracle Single Sign-On and Oracle Internet Directory (OID) to provide authentication. You should use OID (the LDAP-based provider) to provide identity management in production environments where scalability and manageability are important. In this case, you will need to administer the users through the LDAP administration tools provided with Oracle Containers for J2EE.

For more information on using OID, see the *Oracle Identify Management Guide to Delegated Administration* from the Oracle Application Server documentation library.

In addition, JDeveloper provides an XML-based resource provider (`system-jazn-data.xml`) that can be used for small scale applications or for development and testing purposes. This provider contains user, role, grant, and login module configurations.

18.3 Configuring Authentication Within the web.xml File

In many web-based applications, there may be a link to "protected" areas of the site that require knowing who the originator of the request is; in other words, access to the linked area requires an authenticated user. This can be accomplished dynamically with the adfAuthentication servlet or without ADF, using only J2EE container-managed authentication provided by OC4J. Either way, by configuring the container with security constraints, you prevent access to the server without an authenticated session.

> **Note:** The SRDemo application currently does not demonstrate Oracle ADF Security at the ADF Model layer. To understand how the SRDemo application handles authentication, see Section 18.3.1, "How to Enable J2EE Container-Managed Authentication".

Once the user is authenticated, the application can determine whether that user has privileges to access the resource as defined by any authorization constraint. You configure this constraint and set up users or roles for you application to recognize in the `web.xml` file.

For example, in the SRDemo application, three roles determine who gets access to perform what type of functions. Each user must be classified with one of the three roles: user, technician or manager. All of these criterion are implemented using container managed Form-based authentication supported by Oracle Application Server.

18.3.1 How to Enable J2EE Container-Managed Authentication

If your application contains pages that require a user to be authenticated against a data store in order to be accessed, you must declare the following in the `web.xml` configuration file:

- `<security-role>` defines valid roles in the security context.
- `<login-config>` defines the protocol for authentication, for example form-based or HTTPS.
- `<security-constraint>` defines the resources specified by URL patterns and HTTP methods that can be accessed only by authorized users or roles.

- `<servlet>` defines the servlet that provides authentication.
- `<servlet-mapping>` maps the servlet to a URL pattern. The
- `<filter>` defines the filter used to transform the content of the authentication request.
- `<filter-mapping>` maps the filter to the file extensions used by the application. For details about the ADF binding filter, see Configuring the ADF Binding Filter.

> **Note:** When you insert an ADF Faces component into a JSF page for the first time, JDeveloper updates the `web.xml` file to define the ADF Faces servlet filter and ADF Faces resources servlet. For more details about the these servlet settings, see What Happens When You First Insert an ADF Faces Component.

The security roles that you define in the `web.xml` file identify the logical names of groups of users that your application recognizes. You will create security constraints in order to restrict access to particular web pages based on whether the authenticated user belongs to the authorized role or not.

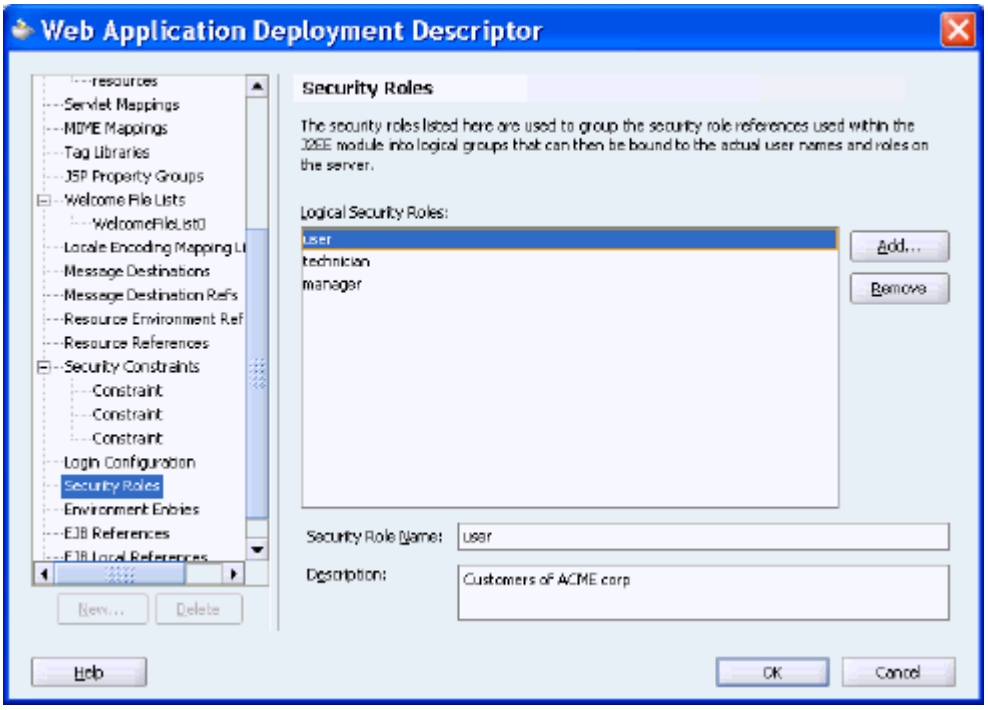
To specify security roles for J2EE container-managed security:

1. In the Navigator, expand your JSP project, right-click the **web.xml** file and choose **Properties**. The `web.xml` file resides in the WEB-INF folder of your project.
2. To add the security role definition, select **Security Roles** on the left panel of the Web Application Deployment Descriptor editor and click **Add**.

   The roles you enter here must match roles from your data store. For example, if you are using the XML-based provider (as defined with `system-jazn-data.xml`), you would enter the value of `<name>` for any of the defined `<roles>` that need to be authenticated. Additionally, if you configure OC4J to use security role mapping, the role names must also match the roles defined in the `<security-role-mapping>` element of the `orion-web.xml` configuration file.
3. Save all changes and proceed to create the login configuration, as described below.

Figure 18–1 shows the `web.xml` editor with the Security Roles definition displayed. In the SRDemo application, three security roles are defined.

Figure 18–1 Web Application Deployment Descriptor Dialog, Security Roles Panel



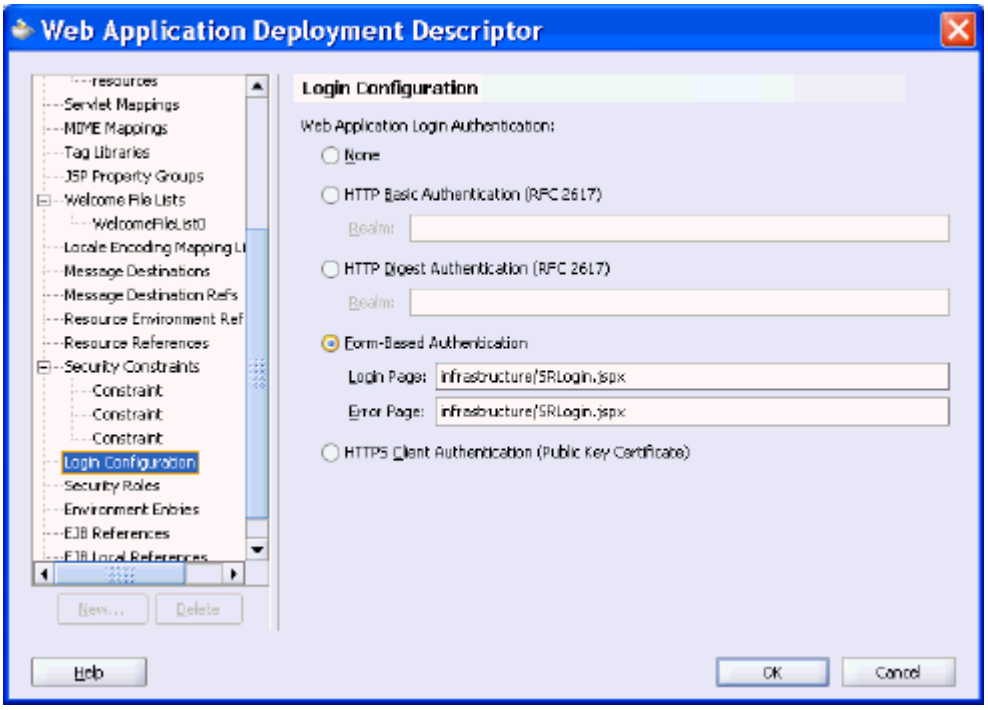
Before configuring the login configuration, you should already have created a login web page and the optional login error page. For details, see Section 18.4, "Creating a Login Page".

To create a login configuration for J2EE container-managed security:

1. In the Navigator, expand your JSP project, right-click the **web.xml** file and choose **Properties**. The `web.xml` file resides in the WEB-INF folder of your project.
2. To create a login configuration, select **Login Configuration** on the left panel of the editor. For example, to use form-based authentication, you would select **Form-Based Authentication**, and enter the name of the file used to render the login and login error page, for example `login.jspx` and `loginerror.jspx`. For further details, see Section 18.4.1, "Wiring the Login and Error Pages".
3. Save all changes and close the Web Application Deployment Descriptor editor.

Figure 18–2 shows the `web.xml` editor with the Login Configuration definition displayed.

Figure 18–2 Web Application Deployment Descriptor Dialog, Login Configuration Panel



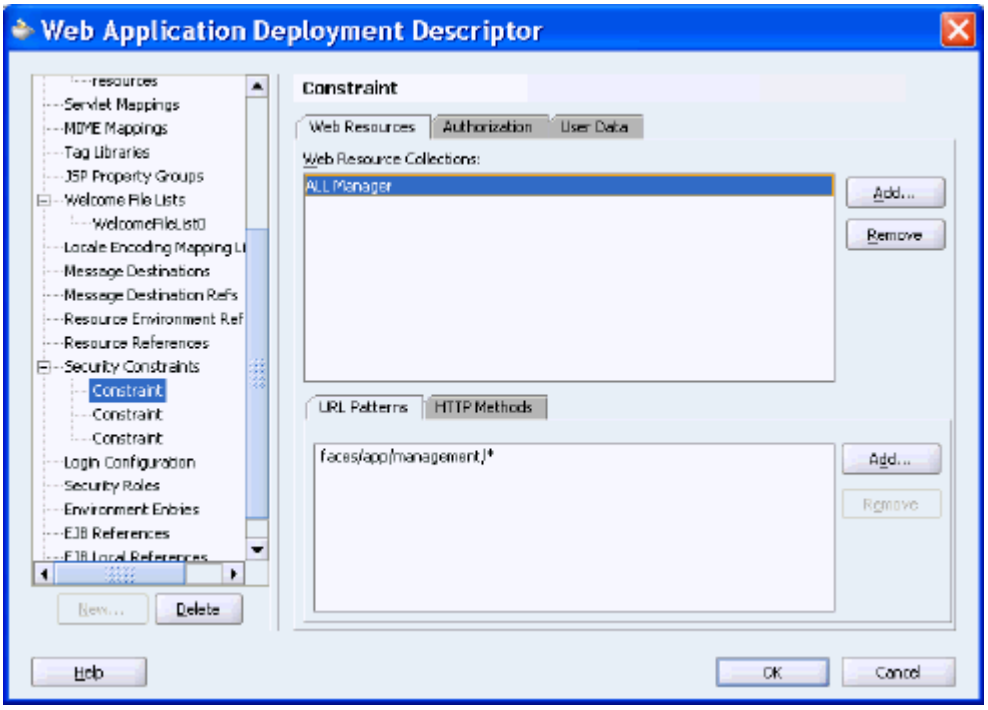
To create security constraints for J2EE container-managed security:

1. In the Navigator, expand your JSP project, right-click the **web.xml** file and choose **Properties**. The `web.xml` file resides in the WEB-INF folder of your project.
2. To add the security constraint definition, select **Security Constraints** on the left panel of the editor, and at the bottom of the panel click **New**.
3. To add a new Web Resource, on the Constraints page, click **Add**.

   Tip: Because the security constraint is specified as a URL, the web resource name you supply can be based on your application's database connection name. For example, if your database connection is `MyConnection`, then you might type `jdbc/MyConnection` for the web resource name.
4. To specify the URL pattern of your client requests, click the web resource name you just specified, select **URL Patterns**, and click **Add**. Type a forward slash (`/`) to reference a JSP login page located at the top level relative to the web application folder.
5. To specify authorized security roles, select the **Authorization** tab. Select the security roles that require authentication. The roles available are the roles you configured in step 2.
6. To specify transport guarantee, select the **User Data** tab. Select the type of guarantee to use.
7. Save all changes and close the Web Application Deployment Descriptor editor.

Figure 18–3 shows the `web.xml` editor with a Security Constraint definition displayed.

Figure 18–3 Web Application Deployment Descriptor Dialog, Security Constraints Panel



18.3.2 What Happens When You Use Security Constraints without Oracle ADF Security

Example 18–1 shows sample definitions similar to the ones that your `web.xml` file should contain when you have finished configuring J2EE container-managed security.

Example 18–1 J2EE Security Enabled in the SRDemo Application web.xml File

```
<security-constraint>
    <web-resource-collection>
      <web-resource-name>ALL Manager</web-resource-name>
      <url-pattern>faces/app/management/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
      <role-name>manager</role-name>
    </auth-constraint>
</security-constraint>
<security-constraint>
    <web-resource-collection>
      <web-resource-name>AllStaff</web-resource-name>
      <url-pattern>faces/app/staff/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
      <role-name>technician</role-name>
      <role-name>manager</role-name>
    </auth-constraint>
</security-constraint>
<security-constraint>
    <web-resource-collection>
      <web-resource-name>SRDemo Sample</web-resource-name>
      <url-pattern>faces/app/*</url-pattern>
    </web-resource-collection>
```

```
  <auth-constraint>
     <role-name>user</role-name>
        <role-name>technician</role-name>
        <role-name>manager</role-name>
     </auth-constraint>
</security-constraint>
<login-config>
   <auth-method>FORM</auth-method>
   <form-login-config>
      <form-login-page>infrastructure/SRLogin.jspx</form-login-page>
      <form-error-page>infrastructure/SRLogin.jspx</form-error-page>
   </form-login-config>
</login-config>
<security-role>
   <description>Customers of ACME corp</description>
   <role-name>user</role-name>
</security-role>
<security-role>
   <description>Employees of ACME corp</description>
   <role-name>technician</role-name>
</security-role>
<security-role>
   <description>The boss</description>
   <role-name>manager</role-name>
</security-role>
```

When the user clicks a link to a protected page, if they are not authenticated (that is, the authenticated user principal is not currently in SecurityContext), the OC4J security servlet is called and the web container invokes the login page defined by the deployment descriptor `<form-login-config>` element.

Once a user submits their user name and password, that data is compared against the data in a resource provider where user information is stored, and if a match is found, the originator of the request (the user) is authenticated. The user name is then stored in SecurityContext, where it can be accessed to obtain other security related information (such as the group the user belongs to) in order to determine authorization rights.

The `web.xml` deployment descriptor supports declarative security through `<security-constraints>` that specify the resources available to the authenticated users of the application. Whether or not the user is permitted to access a web page depends on its membership in a role identified in the `<auth_constraint>` element. The application calls the servlet method `isUserInRole()` to determine if a particular user is in a given security role. The `<security-role>` element defines a logical name of the roles based on the same names defined by the JAZN realm in the `system-jazn-data.xml` file.

18.3.3 How to Enable Oracle ADF Authentication

For web-based applications, you can configure a security constraint against the adfAuthentication servlet within the `web.xml` file. This constraint prevents access to the servlet without an authenticated session. As long as the link to the protected area contains the URL pattern defined in the constraint, the web container will invoke the login page if the user is not authenticated.

> **Note:** The adfAuthentication servlet is optional and allows dynamic authentication, that is, if the user has not yet logged in and the page being accessed needs authorization, then the user will be prompted to log in. The servlet take an optional parameter `success_url`. If `success_url` is specified, then after successfully logging in, the user is directed to the requested page. If `success_url` is not specified, then after successful login, the servlet directs the user back to the page from which the login was initiated.

To configure web.xml for Oracle ADF Security:

1. In the Navigator, expand your JSP project, right-click the `web.xml` file and choose **Properties**. The `web.xml` file resides in the WEB-INF folder of your project.
2. Define Security Roles, Login Configuration, and Security Constraints as you normally would. (See above procedures.)
3. To create the `<servlet>` element for the ADF authentication servlet, select **Servlets/JSP** on the left panel of the editor and click **New**. Enter the following:

   Servlet Name: `adfAuthentication`

   Servlet Class:
   `oracle.adf.share.security.authentication.AuthenticationServlet`

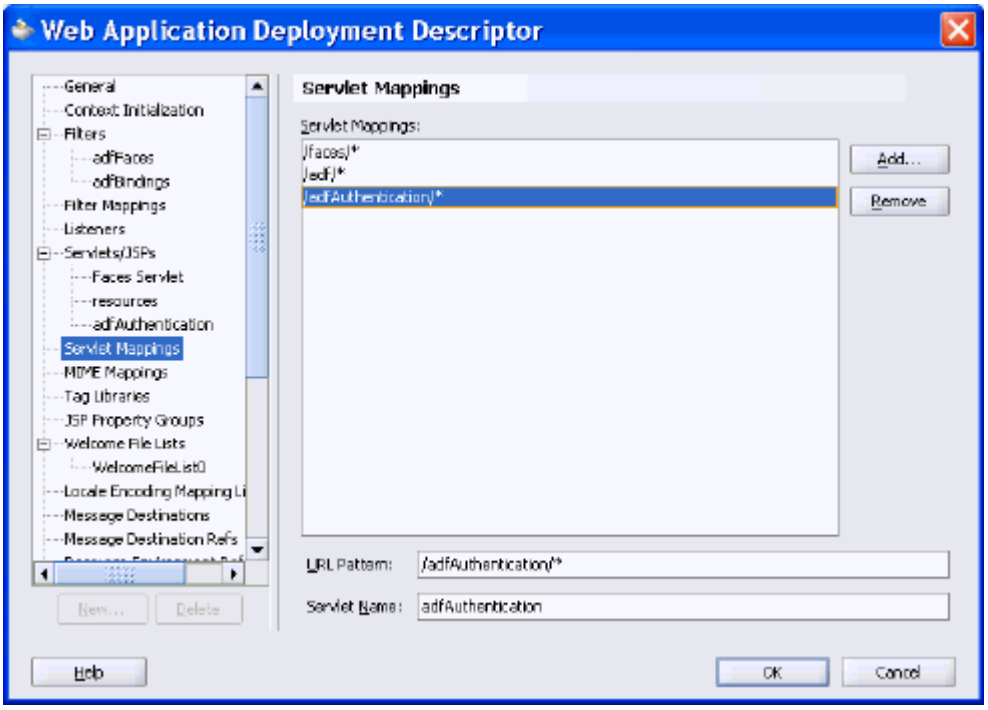
   To add an initialization parameter that contains the URL for the resulting page if authentication succeeds, select **Initialization Parameters** and click **Add**. If you do not enter a URL, the user will return to the current page.
4. To create a servlet mapping, select **Servlet Mapping** on the left panel of the editor, and click **Add**. Enter the following:

   URL Pattern: `/adfAuthentication/*`

   Servlet Name: `adfAuthentication`
5. Save all changes and close the Web Application Deployment Descriptor editor.

Figure 18–4 shows the `web.xml` editor with the Servlet Mapping definition displayed for the adfAuthentication servlet.

Figure 18–4 Web Application Deployment Descriptor Dialog, Servlet Mapping Panel



18.3.4 What Happens When You Use Security Constraints with Oracle ADF

Example 18–2 shows sample definitions similar to the ones that your `web.xml` file should contain.

Example 18–2 Oracle ADF Security Enabled in a Sample web.xml File

```
<servlet>
  <servlet-name>adfAuthentication</servlet-name>
<servlet-class>oracle.adf.share.security.authentication.
                            AuthenticationServlet</servlet-class>
  <init-param>
    <param-name>sucess_url</param-name>
    <param-value>inputForm.jsp</param-value>
  </init-param>
</servlet>
<servlet-mapping>
  <servlet-name>adfAuthentication</servlet-name>
  <url-pattern>/adfAuthentication/*</url-pattern>
</servlet-mapping>
<security-constraint>
  <web-resource-collection>
    <web-resource-name>adfAuthentication</web-resource-name>
    <url-pattern>/adfAuthentication</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>user</role-name>
  </auth-constraint>
</security-constraint>
```

```
<login-config>
  <auth-method>FORM</auth-method>
  <form-login-config>
    <form-login-page>login.jspx</form-login-page>
    <form-error-page>login.jspx</form-error-page>
  </form-login-config>
</login-config>
<security-role>
  <role-name>user</role-name>
</security-role>
```

When the user clicks a link to a protected page, if they are not authenticated (that is, the authenticated user principal is not currently in SecurityContext), the Oracle ADF Security Login Servlet is called and the web container invokes the login page.

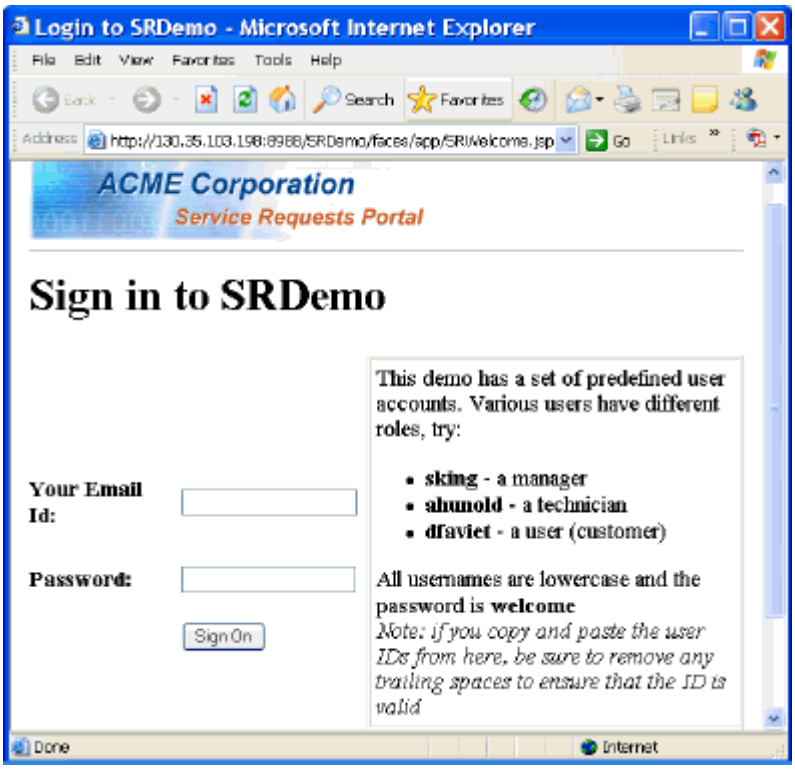
Once a user submits their user name and password, that data is compared against the data in a resource provider where user information is stored, and if a match is found, the originator of the request (the user) is authenticated. The user name is then stored in SecurityContext, where it can be accessed to obtain other security related information (such as the group the user belongs to) in order to determine authorization rights.

Because Oracle ADF Security implements OracleAS JAAS, authentication also results in the creation of a JAAS Subject, which also represents the originator of the request.

18.4 Creating a Login Page

The login page for a web application should use the J2EE security container login method `j_security_check` as a method that the form posts. Figure 18–5 shows a sample login page from the SRDemo application.

Figure 18–5 Sample Login Page from the SRDemo Application



CAUTION: When you create the login page, you may use JSP elements and JSTL tags. Your page can be formatted as a JSFX document, but due to a limitation in relation to JSF and container security, JSF components cannot be used.

To create a web page for the login form:

1. With the user interface project selected, open the New Gallery and select **JSP** from the **Web Tier - JSP** category. Do NOT select the Web Tier - JSF category to create a JSPX document as a login form.
2. In the Create JSP wizard, choose **JSPX Document** type for the JSP file type. The wizard lets you create a JSPX document without using managed beans.
3. On the Tag Libraries page of the wizard, select **All Libraries** and add **JSTL Format 1.1** and **JSTL Core 1.1** to the **Selected Libraries** list.
4. Click **Finish** to complete the wizard and add the JSPX file to the user interface project.
5. In the Component Palette, select the **JSTL 1.1 FMT** page, and drag **SetBundle** into the Structure window for the JSPX document so it appears above the title element.
6. In the Insert SetBundle dialog, set **BaseName** to the package that contains the resource bundle for the page. For example, in the SRDemo application, it is `oracle.srdemo.view.resources.UIResources`.
7. Optionally, drag **Message** onto the title element displayed in the Structure window. Double-click the Message element and set the **key** property to the resource bundle's page title key. For example, in the SRDemo application, the key is `srlogin.pageTitle`. Delete the string title leftover from the page creation.
8. In the Component Palette, select the **HTML Forms** page and drag **Form** inside the page body. In the Insert Form dialog, set the action to `j_security_check` and set the method to **post**.
9. Drag **Text Field** for the user name into the form and set the name to `j_username`.
10. Drag **Password Field** into the form and name it `j_password`.
11. Drag **Submit Button** into the form with label set to `Sign On`.
12. In the Component Palette, again select the **JSTL 1.1 FMT** page, and drag two **Message** tags into the form so they appear beside the input fields. Set their key properties. For example, in the SRDemo application, the resource keys are `srlogin.password and srlogin.username`.

Example 18–3 shows the source code from the SRDemo application's login page. This JSPX document uses only HTML elements and JSTL tags to avoid conflicts with the security container when working with JSF components. The security check method appears on the `<form>` element and the form contains input fields to accept the user name and password. These fields assign the values to the container's login bean attributes `j_username` and `j_password`, respectively.

Example 18–3 Sample Source from SRLogin.jspx

```
<html>
    <head>
      <meta http-equiv="Content-Type"
            content="text/html; charset=windows-1252"/>
      <fmt:setBundle basename="oracle.srdemo.view.resources.UIResources"/>
      <title>
        <fmt:message key="srdemo.login"/>
      </title>
    </head>
    <body>
        ... omitting the "number of attempts" checking logic ...
        <form action="j_security_check" method="post">
          <table cellspacing="3" cellpadding="2" border="0" width="100%">
            <tr>
              <td colspan="3">
                <img height="69" width="340"
                     src="/SRDemo/faces/images/SRBranding.gif"
                     alt="SRDemo Logo"/>
                <hr/>
              </td>
            </tr>
            <tr>
              <td colspan="3">
                <h1>
                  <fmt:message key="srlogin.pageTitle"/>
                </h1>
              </td>
            </tr>
            <tr>
              <td colspan="3">
                <c:if test="${sessionScope.loginAttempts >0}">
                  <h3><fmt:message key="srdemo.badLogin"/></h3>
                </c:if>
              </td>
            </tr>
            <tr>
              <td>&amp;nbsp;</td>
              <td> </td>
              <td rowspan="7">
                <table border="1" cellpadding="5">
                  <tr>
                    <td>
                      <fmt:message key="srlogin.info"/>
                    </td>
                  </tr>
                </table>
              </td>
            </tr>
            <tr>
              <td>&amp;nbsp;</td>
            </tr>
            <tr>
              <td width="120">
                <b><fmt:message key="srlogin.username"/></b>
              </td>
              <td>
                <input type="text" name="j_username"/>
              </td>
            </tr>
```

```
            <tr>
              <td width="120">
                <b><fmt:message key="srlogin.password"/></b>
              </td>
              <td>
                <input type="password" name="j_password"/
              </td>
            </tr>
            <tr>
              <td> </td>
              <td>
                <input type="submit" name="logon" value="Sign On"/>
              </td>
            </tr>
            <tr>
            </tr>
              <td>&amp;nbsp;</td>
            <tr>
              <td>&amp;nbsp;</td>
            </tr>
            <tr>
              <td>&amp;nbsp;</td>
            </tr>
            <tr>
              <td colspan="3">
                <hr/>
              </td>
            </tr>
          </table>
        </form>
      </c:if>
    </body>
</html>
```

18.4.1 Wiring the Login and Error Pages

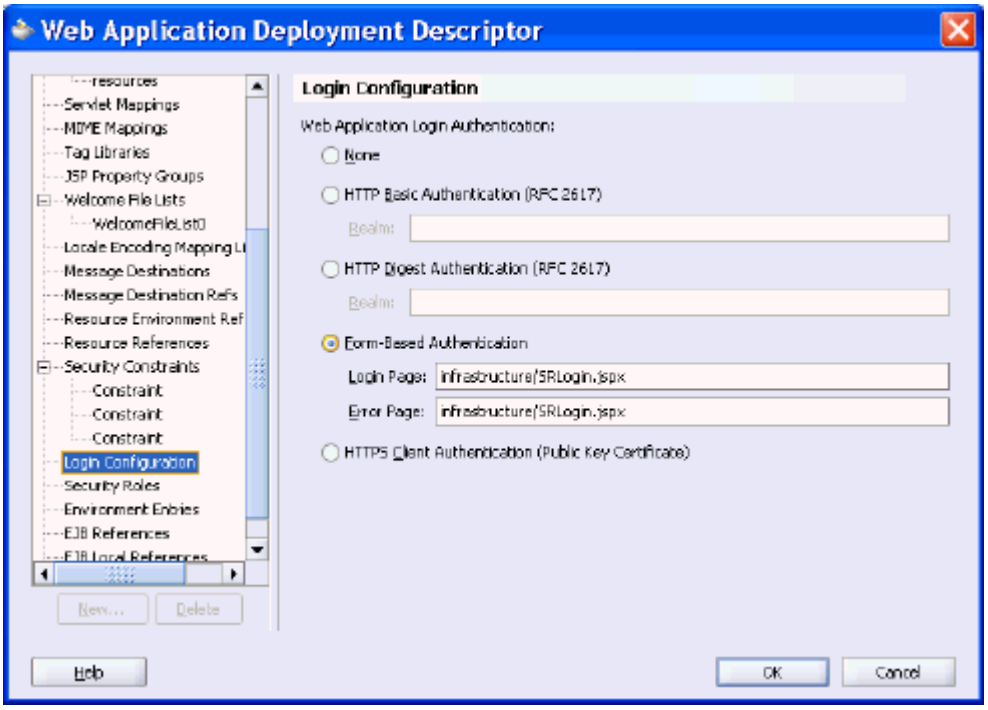
To allow the web container to perform authentication, the `web.xml` file must contain the login configuration information that specifies the page to display for log in and another page to display when log in fails because the user could not be authenticated.

To configure how login is to be handled:

1. In the Application Navigator, locate **web.xml** in the WEB-INF folder.
2. Right-click **web.xml** and choose **Properties**.
3. In the Web Application Deployment Descriptor dialog, select **Login Configuration**.
4. Choose **Form-Based Authentication** and enter the path name for both the login and error page. The path specified for the login page and error page is relative to the document root that will be used to authenticate the user. For example, in the SRDemo application, the path `infrastructure/SRLogin.jspx` is used for both the login and error page.

Figure 18–6 shows the `web.xml` editor with the Login Configuration definition displayed.

Figure 18–6 Web Application Deployment Descriptor Dialog, Login Configuration Panel



18.4.2 What Happens When You Wire the Login and Error Pages

When you define the `web.xml` login configuration information, JDeveloper creates these definitions:

```
<login-config>
    <auth-method>FORM</auth-method>
    <form-login-config>
      <form-login-page>infrastructure/SRLogin.jspx</form-login-page>
      <form-error-page>infrastructure/SRLogin.jspx</form-error-page>
    </form-login-config>
  </login-config>
```

Because you selected Form-based authentication, to specify user-written HTML Form for authentication, the page servlet will look for the JSP page you specified to authenticate the user. The JSP page must return an HTML page containing a Form that conforms to a specific naming convention. Similarly, when authentication fails, the servlet will look for a page to display. In the SRDemo application, the same page appears for both cases, though you could have defined different pages.

Example 18–3 shows the conventions of that permit the HTML Form to invoke the authentication servlet. Specifically, the form must specify three pieces of information:

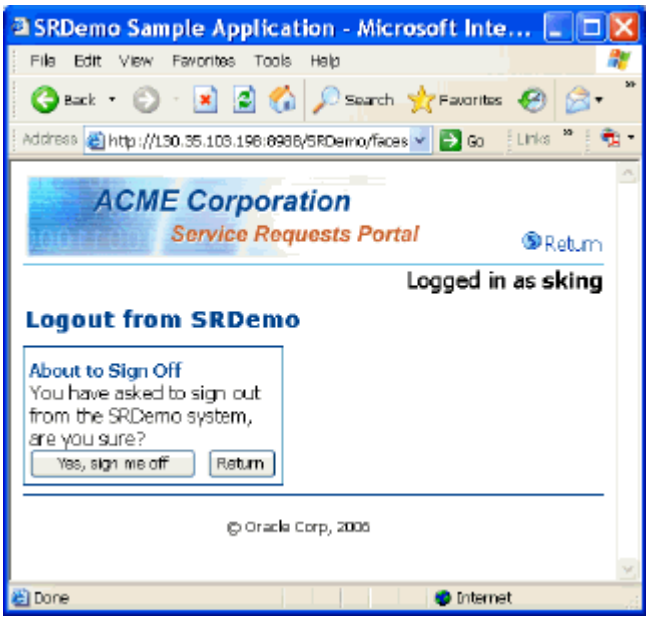
1. `<form action="j_security_check" method="post">` to invoke the security check method `j_security_check` on the container's login bean.
2. `<input type="text" name="j_username"/>` to assign the username value to the container's login bean attribute `j_username`.
3. `<input type="password" name="j_password"/>` to assign the password value to the container's login bean attribute `j_password`.

Please note that the value of the login bean attributes must be retuned by the HTML Form with the exact names shown. In a JSF JSP page, a JSF form does not guarantee this. Therefore, Oracle recommends that you use a JSP document page in order to use the HTML Form to preserve the login bean attribute names.

18.5 Creating a Logout Page

The logout page may be called from the global logout button that appears on any page that includes the global menu page. The purpose of the logout page is to provide a prompt for the user to confirm that they want to quit. If the user chooses to log out, their session is invalidated and then they are redirected back to the application's welcome page. They will have to log in again to continue the application. Figure 18–7 shows the logout page from the SRDemo application.

Figure 18–7 Sample Logout Page from SRDemo Application



To create the logout page:

1. With the user interface project selected, open the New Gallery and select **JSF JSP** from the **Web Tier - JSF** category. In this case, it is acceptable to use JSF components.
2. In the Create JSF JSP wizard, choose **JSP Document** type for the JSF JSP file type. In this case, you want to create a JSPX document that will use JSF components.
3. On the Component Binding page, do not create a managed bean.
4. On the Tag Libraries page of the wizard, add **ADF Faces Components** and **ADF Faces HTML** to the **Selected Libraries** list.
5. Click **Finish** to complete the wizard and add the JSPX file to the user interface project.
6. In the Component Palette, select the **ADF Faces Core** page, and drag the components **Document**, **Form**, and **PanelPage** so that PanelPage appears nested inside Form, and Form appears nested inside Document.

7. Next construct the PanelPage container for the command buttons by dragging the components **PanelBox**, **PanelHeader**, **PanelButtonBar** so that PanelButtonBar appears nested inside PanelHeader, and PanelHeader appears nested inside PanelBox. All should be nested inside PanelPage.
8. To create the buttons that give the user the choice whether to logout or not, drag two **CommandButton** components inside the PanelButtonBar.
9. The first button should provide the logout function. You can wire it separately by creating a managed bean. For details, see Section 18.5.1, "Wiring the Logout Action".
10. The second button should invoke an action **GlobalHome** to direct the user to the desired page. This action will be defined in the `faces-config.xml` file with a navigation rule.

Example 18–4 shows the source code from the SRDemo application's logout page. This JSPX document has no restriction on using JSF components because the page has no interaction with the security container. The action to invoke the logout function appears on the `<af:commandButton>` with the logout label.

Example 18–4 Sample Source from SRLogout.jspx

```
<af:form>
      <af:panelPage title="#{res['srlogout.pageTitle']}">
        <!--Page Content Start-->
        <af:panelBox>
          <af:panelHeader text="#{res['srlogout.subTitle']}"
                          messageType="warning">
            <af:outputText value="#{res['srlogout.text']}"/>
            <af:panelButtonBar>
              <af:commandButton text="#{res['srlogout.logout.label']}"
                                action="#{backing_SRLogout.logoutButton_action}"/>
              <af:commandButton text="#{res['srlogout.goBack.label']}"
                                action="GlobalHome"/>
            </af:panelButtonBar>
          </af:panelHeader>
        </af:panelBox>
        <!-- Page Content End -->
        ... omitting facets related to the visual design of the page ...
      </af:panelPage>
</af:form>
```

18.5.1 Wiring the Logout Action

To handle the logout action, the JSPX document can use a managed bean with properties that correspond to the logout page's logout command button.

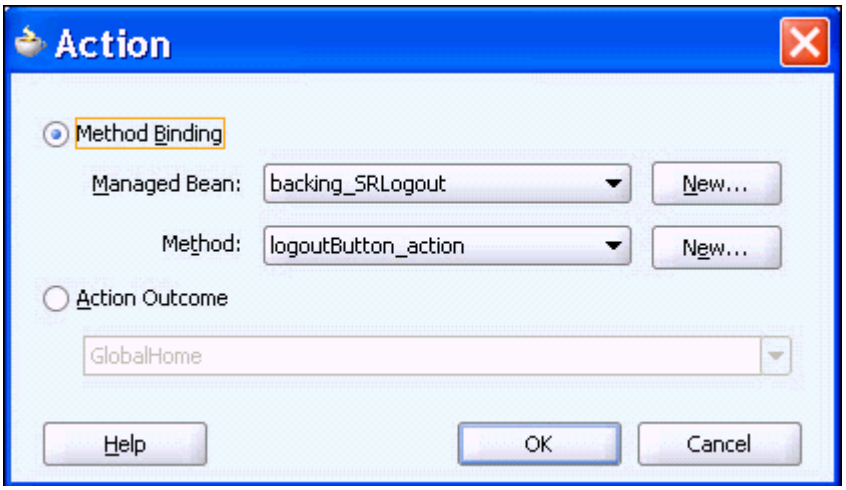
To handle the logout action:

1. In the open logout page, double-click the command button that you reserved for the logout action.
2. In the Action property dialog, leave **Method Binding** selected and click **New** to define the Managed Bean class.
3. In the Create Managed Bean dialog, specify the new class file name for the managed bean and enter the name of the managed bean to register with the `faces-config.xml` file.

4. In the Action property dialog, click **New** to name the method that you will implement in the managed bean class to return a string that sets the component's outcome value.

   Figure 18–8 shows the Action property dialog with the managed bean `backing_SRLogout` and the method `logoutButton_action()` entered.

Figure 18–8 Action Binding Dialog for Logout CommandButton



5. In the generated `.java` file, implement the method handler for the command button that will redirect the user back to an appropriate page. See Example 18–5 for a sample.

   Warning: If your application calls the `invalidate()` method on the HTTP Session to terminate the current session at logoff time, you *must* use a "Redirect" to navigate back to a home page to require accessing an ADF Model binding container. The redirect to a databound page ensures that the ADF Binding Context gets created again after invalidating the HTTP Session.

Example 18–5 shows the method handler from the SRDemo application logout page's managed bean. The `logoutButton_action()` method invalidates the session and redirects to the home page. The security container will prompt the user to reauthenticate automatically.

Example 18–5 Sample Source from SRLogout.java

```
public String logoutButton_action() throws IOException{
  ExternalContext ectx = FacesContext.getCurrentInstance().getExternalContext();
  HttpServletResponse response = (HttpServletResponse)ectx.getResponse();
  HttpSession session = (HttpSession)ectx.getSession(false);
  session.invalidate();

  response.sendRedirect("SRWelcome.jspx");
  return null;
}
```

18.5.2 What Happens When You Wire the Logout Action

When you define the `action` property for the command button, JDeveloper updates the `Logout.jspx` page source code with the name of the managed bean and bean method to invoke:

```
<af:commandButton text="#{res['srlogout.logout.label']}"
                  action="#{backing_SRLogout.logoutButton_action}"/>
```

and, JDeveloper updates the `faces-config.xml` file to define the managed bean:

```
<managed-bean>
     <managed-bean-name>backing_SRLogout</managed-bean-name>
     <managed-bean-class>oracle.srdemo.view.backing.SRLogout</managed-bean-class>
     <managed-bean-scope>request</managed-bean-scope>
</managed-bean>
```

Once a user clicks the logout button, the JSF controller identifies the corresponding class file from the Faces configuration file and passes the name of the action handler method to the managed bean. In turn, the action handler, shown previously in Example 18–5, invalidates the session and redirects to the home page.

18.6 Implementing Authorization Using Oracle ADF Security

Authorization provides a way to restrict access to a resource based on the user attempting access. Oracle ADF Security implements OracleAS JAAS for authorization of security-aware resources.

Oracle ADF Security provides another level of granularity, allowing object instance access control based on Java Permissions using JAAS. Specifically, certain Oracle ADF Model layer objects are "security-aware," meaning that there are pre-defined component-specific permissions that a developer can grant for a given resource.

Note: The SRDemo application currently does not demonstrate Oracle ADF Security at the ADF Model layer. To understand how the SRDemo application handles authorization, see Section 18.7, "Implementing Authorization Programmatically".

The following Oracle ADF objects are security-aware as defined by the page definition file associated with each databound web page:

- Binding container
- Iterator binding
- Attribute binding
- MethodAction binding

You set grants on these objects by defining which authenticated users or roles have permission to perform a given action on the object (called a resource). Grantees, which are roles, users, or groups defined as principals are mapped to permissions. Permissions are permission to execute a specific action against a resource, as defined by Oracle ADF Security classes (see the Oracle ADF Javadoc for details). Grants are aggregated. That is if a group's role is granted permissions, and a user is a member of that group, then the user also has those permissions. If no grant is made, then access by the role, user, or group is denied.

Table 18–1 shows permissions you can grant on binding containers, iterator bindings, attribute-level bindings (for example, table, list, boolean, and attribute-value bindings), and method bindings. You use the Authorization Editor to grant permissions for users on the Oracle ADF objects created at runtime from the page definition file.

Table 18–1 Oracle ADF Security Authorization Permissions

ADF Model Object	Defined Actions	Affect on Components in the User Interface
Binding Container for a web page	**grant** - can administer the permissions on the page	On pages that allow runtime customization, any link or button configured to set access controls will be disabled for users not granted this permission.
	edit - can edit content on the page	If a user is granted permission for the view action, but not for the edit action, then any data in input text boxes will display as read only.
	personalize - allows the user customization of the page	On pages that allow runtime customization, any link or button configured to put the page into personalization mode will be disabled for users not granted this permission.
	view - can view the page	A user not granted this permission will be shown an authorization error.
Iterator Binding	**read** - can read the returned rows	All rows of data will be returned. However, you can limit what can be displayed or updated by placing grants on the individual attribute bindings.
	update - can update data in a row	If the Commit operation is dropped as a command button from the Data Control Palette, the button will be disabled for users who were not granted this permission. Instead of limiting updates to an entire row, you can instead limit the ability to update individual attributes.
	create - can create a new row	If the Create operation is dropped as a command button from the Data Control Palette, the button will be disabled for any users that were not granted this permission.
	delete - can delete a row	If the Delete operation is dropped as a command button from the Data Control Palette, the button will be disabled for any users that were not granted this permission.
Method Action Binding	**invoke** - the method can execute	If the method is bound to a command button, that button will be disabled for any users that were not granted this permission. If the method is invoked implicitly, the method will only execute for users granted this permission.
Attribute-level Bindings	**read** - can read the attribute's value	The value for the attributes will be displayed.
	update - can update the attribute 's value	Any data in input text boxes will display as read only for users who were not granted this permission.

Before you can implement Oracle ADF authorization, you must first:

- Configure authentication for the ADF Authentication servlet. For details, see Section 18.3.3, "How to Enable Oracle ADF Authentication".
- Configure your application to use Oracle ADF Security authorization. For details, see Section 18.6.1, "Configuring the Application to Use Oracle ADF Security Authorization".

18.6.1 Configuring the Application to Use Oracle ADF Security Authorization

You must first configure the application to use Oracle ADF Security before you can work with ADF authorization in your application.

18.6.1.1 How to Configure Oracle ADF Security Authorization

To enable Oracle ADF Security authorization, you create a configuration file named `adf-config.xml` that sets the application's container to use Oracle ADF Security. The file initializes the ADFContext and SecurityContext.

To configure an application to use Oracle ADF Security:

1. Right-click on the project for which security is needed and choose **New**.
2. In the New Gallery, select the **XML** category.

   If XML is not displayed, use the **Filter By** list at the top to select **All Technologies**.
3. In the Items list, select **XML Document** and click **OK**.
4. Name the file `adf-config.xml`, save it in the *<application_name>*`/.adf/META-INF` directory, and click **OK**.

   The file opens in the source editor.
5. Replace the generated code with the following:

```
<?xml version="1.0" encoding="windows-1252" ?>
<adf-config xmlns:xsi=" http://www.w3.org/2001/XMLSchema-instance "
    xsi:schemaLocation=" http://xmlns.oracle.com/adf/config
    ../../../../../bc4jrt/src/oracle/adf/share/config/schema/config.xsd"
    xmlns=" http://xmlns.oracle.com/adf/config "
    xmlns:sec=" http://xmlns.oracle.com/adf/security/config ">
  <sec:adf-config-child xmlns=" http://xmlns.oracle.com/adf/security/config ">
     <JaasSecurityContext
        initialContextFactoryClass="oracle.adf.share.security.
             JAASInitialContextFactory"
        authorizationEnforce="true"
        jaasProviderClass="oracle.adf.share.security.providers.jazn.
             JAZNSecurity Context" >
    </JaasSecurityContext>
   </sec:adf-config-child>
</adf-config>
```

6. Save and close the file.

18.6.1.2 What Happens When You Configure An Application to Use Oracle ADF Security

The `authorizationEnforce` parameter in the `<JaasSecurityContext>` element set to true will allow the authenticated user principals to be placed into ADF SecurityContext once the user is authenticated.

> **Tip:** If you want to run the application without using Oracle ADF Security, simply set the `authorizationEnforce` parameter to false.

18.6.1.3 What You May Need to Know About the Authorization Property

Because security can be turned on and off, it is recommended that an application should determine this property setting before invoking an authorization check. The application can check if Oracle ADF Security is enabled by checking the authorization property setting. This is exposed through the `isAuthorizationEnabled()` method of the SecurityContext under the ADFContext. For example:

```
if (ADFContext.getCurrent().getSecurityContext().isAuthorizationEnabled())
  {
    Permission p = new RegionPermission("view.pageDefs.page1PageDef", "Edit");
    AccessController.checkPermission(p);
    // do the protected action
  } catch (AccessControlException ace) {
    // do whatever's appropriate on an access denied
}
```

18.6.2 Setting Authorization on ADF Binding Containers

You use the Authorization Editor to grant permissions for users on the binding container as it is defined by the entire page definition. See Table 18–1 for details about available Oracle ADF permissions.

To grant permissions on the binding container using the Authorization Editor:

1. Create your web page. From the Visual Editor, right-click the page and choose **Go to Page Definition**.
2. In the Structure window, right-click the root node, **PageDef**, and choose **Edit Authorization**.
3. The Authorization Editor shows the pre-defined permissions for the binding container, along with the principals (roles and users) as defined by your resource provider.

   Click **Help** or press F1 for more help on using this dialog.

18.6.3 Setting Authorization on ADF Iterator Bindings

You use the Authorization Editor to grant permissions for users on iterator bindings. See Table 18–1 for details about available Oracle ADF permissions.

To grant permissions on iterators using the Authorization Editor:

1. Create your web page. From the Visual Editor, right-click the page and choose **Go to Page Definition**.
2. In the Structure window, expand the **executables** node.
3. Right-click on the iterator you wish to grant a permission for and choose **Edit Authorization**.
4. The Authorization Editor shows the pre-defined permissions for the iterator, along with the principals (roles and users) as defined by your resource provider.

   Click **Help** or press F1 for more help on using this dialog.

18.6.4 Setting Authorization on ADF Attribute and MethodAction Bindings

You use the Authorization Editor to grant permissions for users on attribute and method action bindings.

Note that permissions granted on an attribute reflect the ability to execute operations such as Create, Delete, and Commit. Therefore, do not set authorization on the operations, but instead on the attribute or iterator. See Table 18–1 for details about Oracle ADF permissions.

To grant permissions on attribute and method bindings using the Authorization Editor:

1. Create your web page. From the Visual Editor, right-click the page and choose **Go to Page Definition**.
2. In the Structure window, expand the **bindings** node.
3. Right-click on the attribute or method action binding you wish to grant a permission for and choose **Edit Authorization**.
4. The Authorization Editor shows the pre-defined permissions for the attribute or method action binding, along with the principals (roles and users) as defined by your resource provider.

   Click **Help** or press F1 for more help on using this dialog.

18.6.5 What Happens When Oracle ADF Security Handles Authorization

When a user attempts to execute an action against a resource which has a defined grant, Oracle ADF Security checks to see if the user is a principal defined in the grant. If the user is not yet authenticated, the application displays the login page or form. If the user has been authenticated, and does not have permission, a security error is displayed.

Example 18–6 shows grants for the attribute binding and method binding if you are using the Oracle JAZN lightweight XML provider, these grants are written in the `system-jazn-data.xml` file. Note that in these grants, the role `users` has been granted a `MethodPermission` to invoke the `deleteDepartments()` method, and also an `AttributePermission` to read the `DepartmentID` attribute value.

Example 18–6 Sample system-jazn-data.xml File Oracle ADF Permissions

```
<grant>
  <grantee>
    <principals>
      <principal>
        <realm-name>jazn.com</realm-name>
        <type>role</type>
        <class>oracle.security.jazn.spi.xml.XMLRealmRole</class>
        <name>jazn.com/users</name>
      </principal>
    </principals>
  </grantee>
  <permissions>
    <permission>
      <class>oracle.adf.share.security.authorization.MethodPermission</class>
      <name>SessionEJB.dataProvider.deleteDepartments</name>
      <actions>invoke</actions>
    </permission>
```

```
    <permission>
      <class>oracle.adf.share.security.authorization.AttributePermission</class>
      <name>EmployeesView1.DepartmentId</name>
      <actions>read</actions>
    </permission>
  </permissions>
</grant>
```

Users or roles are those already defined in your resource provider.

18.7 Implementing Authorization Programmatically

You can set authorization policies against resources and users. For example, you can allow only certain groups of users the ability to view, create, or change certain data or invoke certain methods. Or, you can prevent components from rendering based on the group a user belongs to. Because the user has been authenticated, the application can determine whether or not to allow that user access to any object that has an authorization restraint configured against it.

The application can reference roles programmatically to determine whether a specific user belongs to a role. In the SRDemo application this is accomplished using the method `isUserInRole()` defined by the `FacesContext` interface (and also available from the `HttpServletRequest` interface).

The SRDemo application uses three core roles to determine who will have access to perform specific functions. Each user is classified with by the roles: user, technician, or manager. The `remoteUser` value (obtained from the Faces Context through the `userid` property) matches the email address in the SRDemo application's USERS table. These criteria are implemented using container-managed, Form-based authentication provided by Oracle Application Server as described in Section 18.3.1, "How to Enable J2EE Container-Managed Authentication".

18.7.1 Making User Information EL Accessible

Once the security container is set up, performing authorization is a task of:

- Reading the container security attributes the first time the application references it
- Making the key security information available in a form that can be accessed through the expression language

To accomplish this, the JSF web application can make use of a managed bean that is registered with session scope. The managed beans are Java classes that you register with the application using the `faces-config.xml` file. When the application starts, it parses this configuration file and the beans are made available and can be referenced in an EL expression, allowing access by the web pages to the bean's content.

For detailed information about working with managed beans, see Section 10.2, "Using a Managed Bean to Store Information".

This sample from `SRList.jspx` controls whether the web page will display a button that the manager uses to display an edit page.

```
<af:commandButton text="#{res['srlist.buttonbar.edit']}"
           action="#{backing_SRList.editButton_action}"
           rendered="#{userInfo.manager}"/>
```

This sample from `SRCreateConfirm.jspx` controls whether the web page will display a user name based on the user's authentication status.

```
<f:facet name="infoUser">
     <!-- Show the Logged in user -->
     <h:outputFormat value="#{res['srdemo.connectedUser']}"
                     rendered="#{userInfo.authenticated}" escape="false">
            <f:param value="#{userInfo.userName}"/>
     </h:outputFormat>
</f:facet>
```

18.7.1.1 Creating a Class to Manage Roles

The managed bean's properties allow you to invoke methods in a class that contains the code needed to validate users and to determine the available roles. This class should be created before you create the managed bean so you know the property names to use when you define the managed bean.

To create the Java class:

1. In the New Gallery select the **General** category and the **Java Class** item.
2. In the Create Java Class dialog, enter the name of the class and accept the defaults to create a public class with a default constructor.

Example 18–7 shows the key methods that the SRDemo application implements:

Example 18–7 SRDemo Application UserInfo.java Sample

```
/**
 * Constructor
 */
public UserInfo() {

        FacesContext ctx = FacesContext.getCurrentInstance();
        ExternalContext ectx = ctx.getExternalContext();

        //Only allow Development mode functions if security is not active
        _devMode = (ectx.getAuthType() == null);

        //Ask the container who the user logged in as
        _userName = ectx.getRemoteUser();

        //Default the value if not authenticated
        if (_userName == null || _userName.length()==0) {
            _userName = "Not Authenticated";
        }

        //Set the user role flag...
        //Watch out for a tricky bug here:
        //We have to evaluate the roles Most > Least restrictive
        //because the manager role is assigned to the technician and user roles
        //thus checking if a manager is in "user" will succeed and we'll stop
        //there at the lower level of priviledge
        for (int i=(ROLE_NAMES.length-1);i>0;i--)  {
            if (ectx.isUserInRole(ROLE_NAMES[i])){
                _userRole = i;
                break;
            }
        }
    }
```

```
/*
 * Function to take the login name from the container and match that
 * against the email id in the USERs table.
 * Note this is NOT an authentication step, the user is already
 * authenticated at this stage by container security. The binding
 * container is injected from faces-config.xml and refers to a special
 * pageDef "headless_UserInfoPageDef.xml" which only contains the definition
 * of this method call,
 */
private Integer lookupUserId(String userName) {
    if (getBindings() != null) {
        OperationBinding oper =
       (OperationBinding)getBindings().getOperationBinding("findUserByEmail");
        //now set the argument to the function with the username we want
        Map params = oper.getParamsMap();
        params.put("emailParam",userName);
        // And execute
        User user = (User)oper.execute();
        setUserobject(user);
        //It is possible that the data in the database has changed and
        //there is no match in the table for this ID - return an appropriate
        //Error in that case
        if (user != null){
            return user.getUserId();
        }
        else{
            FacesContext ctx = FacesContext.getCurrentInstance();
            ctx.addMessage(null,JSFUtils.getMessageFromBundle
             ("srdemo.dataError.userEmailMatch",FacesMessage.SEVERITY_FATAL));
            return -1;
        }
    }
    else {
        //This can happen if the ADF filter is missing from the web.xml
        FacesContext ctx = FacesContext.getCurrentInstance();
        ctx.addMessage(null,JSFUtils.getMessageFromBundle
           ("srdemo.setupError.missingFilter",FacesMessage.SEVERITY_FATAL));
        return -1;
    }
}

/**
 * @return the String role name
 */
public String getUserRole() {
    return ROLE_NAMES[_userRole];
}

/**
 * Get the security container user name of the current user.
 * As an additional precaution make it clear when we are running in
 * Dev mode
 * @return users login name which in this case is also their email id
 */
public String getUserName() {
  StringBuffer name = new StringBuffer(_userName);
  if (_devMode) {
    name.append(" (Development Mode)");
  }
  return name.toString();
```

```
    }

    /**
     * Function designed to be used from Expression Language
     * for swiching UI Features based on role.
     * @return boolean
     */
    public boolean isCustomer() {
        return (_userRole==USER_ROLE);
    }

    /**
     * Function designed to be used from Expression Language
     * for switching UI Features based on role.
     * @return boolean
     */
    public boolean isTechnician() {
        return (_userRole==TECHNICIAN_ROLE);
    }

    /**
     * Function designed to be used from Expression Language
     * for switching UI Features based on role.
     * @return boolean
     */
    public boolean isManager() {
        return (_userRole==MANAGER_ROLE);
    }

    /**
     * Function designed to be used from Expression Language
     * for switching UI Features based on role.
     * This particular function indicates if the user is either
     * a technician or manager
     * @return boolean
     */
    public boolean isStaff() {
        return (_userRole>USER_ROLE);
    }

    /**
     * Function designed to be used from Expression Language
     * for switching UI Features based on role.
     * This particular function indicates if the session is actually authenticated
     * @return boolean
     */
    public boolean isAuthenticated() {
        return (_userRole>NOT_AUTHENTICATED);
    }
}
```

18.7.1.2 Creating a Managed Bean for the Security Information

The UserInfo bean is registered as a managed bean named `userInfo` in the JSF `faces-config.xml` file. The managed bean uses expressions for managed properties which the `UserInfo.java` class implements.

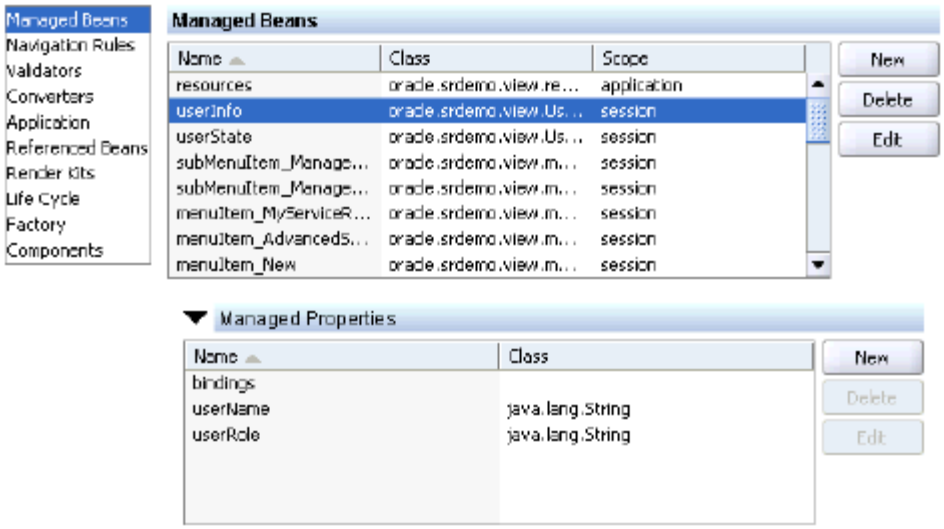
For example, in the SRDemo application the following expressions appear in the UserInfo managed bean:

- `#{userInfo.userName}` either returns the login Id or the String "Not Authenticated"
- `#{userInfo.userRole}` returns the current user's role in its String value, for example, manager
- `#{userInfo.staff}` returns true if the user is a technician or manager
- `#{userInfo.customer}` returns true if the user belongs to the role user
- `#{userInfo.manager}` returns true if the user is a manager

To define the managed bean properties and expressions:

1. In the Application Navigator, open the `faces-config.xml` file in the user interface WEB-INF folder.
2. In the window, select the **Overview** tab.
3. In the element list on the left, select **Managed Beans** and click **New**.
4. In the Create Managed Bean dialog specify the class information for the managed bean. If you have not created the class, see Section 18.7.1.1, "Creating a Class to Manage Roles".
5. To permit the security information defined by the managed bean to accessible by multiple web pages, set **Scope** to **Session**. For example, the SRDemo application defines the managed bean name `userInfo`, corresponding to the `UserInfo.java` class.
6. In the Overview window, click the arrow to the left of the **Managed Properties** bar (appears below the managed bean list) to display properties of the bean.
7. Click **New** to create a unique managed bean property **bindings** with the value `#{data.<ManagedBeanName+PageDefID}`. In the Oracle ADF model, the variable **bindings** makes the binding objects accessible to EL expressions. In the SRDemo application defines the bindings property as UserInfoPageDef. The importance of this expression is described in Section 18.7.2.3, "Create a Page Definition to Make the Method an EL Accessible Object".
8. Optionally, click **New** to create the security properties that your application will access. For example, the SRDemo application defines the **userName** and **userRole** properties as Strings. Figure 18–9 shows the managed bean overview created for the SRDemo application.

Figure 18–9 Overview of userInfo Managed Bean in the Faces Configuration File



Example 18–8 shows the portion of the `faces-config.xml` file that defines the managed bean `userInfo` to hold security information for the SRDemo application. Note that the managed bean also defines the managed property `bindings`. Note that the values shown for managed property `userName` and `userRole` are ignored by the SRDemo application and were included for test purposes only.

Example 18–8 Managed Beans in the SRDemo faces-config.xml File

```
<!-- The managed bean used to hold security information -->
  <managed-bean>
    <managed-bean-name>userInfo</managed-bean-name>
    <managed-bean-class>oracle.srdemo.view.UserInfo</managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
    <managed-property>
      <property-name>bindings</property-name>
      <value>#{data.UserInfoPageDef}</value>
    </managed-property>
    <!-- Test Data ignored if real security is in use-->
    <managed-property>
      <property-name>userName</property-name>
      <property-class>java.lang.String</property-class>
      <value>sking</value>
    </managed-property>
    <managed-property>
      <property-name>userRole</property-name>
      <property-class>java.lang.String</property-class>
      <value>manager</value>
    </managed-property>
  <!-- End Test Data -->
</managed-bean>
```

18.7.2 Integrating the Managed Bean with Oracle ADF Model

The managed bean does have some interaction with the Oracle ADF Model layer. Once the user logs in and the logon ID is obtained, the application needs to translate the login ID into the unique userid that permits the application to identify the user. This information can then be used throughout the application to determine what menus and functionality to display. For instance, in the SRDemo application, the SRList page will only display the Edit button when the user logged in belongs to the manager role. The same authorization restraints is applied to the SRCreate page.

To obtain a unique userid that databound web pages can use to perform authorization:

- Create a TopLink named query that will return a user object for a particular id (which is the value obtained from the container security).
- Create a method on the session bean facade that wraps this lookup method up.
- Create an ADF page definition file for the managed bean to describe its use of this lookup method on the session bean.
- Inject the binding information into the UserInfo bean to provide access to the ADF Model layer to invoke the method on the session bean.
- Execute the custom method from the UserInfo bean the first time the particular id is required for authorization.

18.7.2.1 Creating a TopLink Named Query To Return a User Object

You can identify the user who logs into the application through a named query. This query will return a user object for a unique identifier, such as a particular email id, received from the container security. The query is read-only and takes a String parameter containing the identifier.

To create a named query, use the descriptor of the TopLink SRMap file that corresponds to the USERS table. The query in SRDemo application is based on the email id and receives its value from the security container.

For more information about TopLink named queries, see Section 3.8, "Creating and Modifying Objects with a Unit of Work".

To create a named query for the User entity:

1. In the Application Navigator, expand the data model project and open SRMap to display the Mapping editor.
2. In the Structure window, expand the entities package.
3. In the Mapping editor, select the descriptor **User** and click **Add** to define a new TopLink named query. For example, SRDemo application uses `findUserByEmail`.
4. Use the General panel to add a parameter that identifies the unique attribute. For example, the SRDemo application uses `emailParam` of type `java.lang.String`.
5. Use the Format panel to define an expression for the named query. For example, the SRDemo application uses `email EQUAL emailParam`.
6. Save the query.

18.7.2.2 Create a Session Facade Method to Wrap the Named Query

Oracle recommends that you use a session facade to access entities and methods in order to expose services to clients. The session bean that implements the session facade design pattern, becomes your application's entry point for the Oracle ADF data control. Chapter three describes how to expose services with ADF data controls. Like other methods to be invoked at application runtime, the finder method for the named query must be registered with the Oracle ADF EJB data control in your project. This step begins the process of allowing the ADF Model layer to access the user security information for a uniquely identified user.

If you have not already created a session facade to wrap the TopLink queries, see Section 3, "Building and Using Application Services".

To add a finder method to an existing the session facade:

1. Expand the data model package that contains the session bean for which you created the ADF EJB data control.
2. Double-click the session bean `.java` file to open it in the source editor.
3. Add the new method. Example 18–9 shows the session facade finder method implemented in the SRDemo application.
4. In the Application Navigator, right-click the session bean and choose **Edit Session Facade**.
5. In the Application Navigator, add the new method to the remote interface.
6. Save the `.java` file and recompile.
7. In the Application Navigator, right-click the session bean and choose **Create Data Control**. The new method will appear on the Data Control Palette.

Example 18–9 SRDemo SRPublicFacadeBean.java Finder Method to Expose Unique ID

```
public User findUserByEmail(String emailParam) {
    Session session = getSessionFactory().acquireSession();
    Vector params = new Vector(1);
    params.add(emailParam);
    User result =
      (User)session.executeQuery("findUserByEmail", User.class, params);
    session.release();

    return result;
}
```

18.7.2.3 Create a Page Definition to Make the Method an EL Accessible Object

After the finder method used to return a unique id for the user has been registered with the ADF data control, the next step in exposing the finder methods to the Oracle ADF Model layer is to provide a page definition description, where it will be defined as a method action binding. Once the binding is exposed by the Oracle ADF Model, it can be used throughout the application pages.

Typically, each web page maps to a single page definition file. However, when the action binding is to be accessible throughout the application, the binding definition must belong to its own page definition—one that is "headless"—without a corresponding web page.

To create a headless page definition file for the user interface project:

1. In the Application Navigator, expand the user interface package that contains the page definition files.
2. Right-click the **pageDefs** package node and choose **New**.
3. In the New Gallery, create an XML document from the **General - XML** category.
4. In the Create XML File dialog, name the file for the managed bean that defines the security properties and append `PageDef`. For example, in the SRDemo application, the headless page definition is named `headless_UserInfoPageDef.xml`.
5. Open the XML file in the source editor and add the method binding definition. Example 18–10 shows the binding definition created for the SRDemo application.
6. Save the file.

The value `999` (or `CUSTOM`) set on the action property of the methodBinding specifies the method to be invoked is a custom method defined by the application service.

Example 18–10 SRDemo headless_UserInfoPageDef.xml Page Definition File

```
<?xml version="1.0" encoding="UTF-8" ?>
<pageDefinition xmlns="http://xmlns.oracle.com/adfm/uimodel"
                version="10.1.3.35.65" id="UserInfoPageDef"
                Package="oracle.srdemo.view.pageDefs">
  <bindings>
    <methodAction id="findUserByEmail"
                  InstanceName="SRPublicFacade.dataProvider"
                  DataControl="SRPublicFacade"
                  MethodName="findUserByEmail" RequiresUpdateModel="true"
                  Action="999"
                  ReturnName="SRPublicFacade.methodResults.
                        SRPublicFacade_dataProvider_findUserByEmail_result">
      <NamedData NDName="emailParam" NDType="java.lang.String"/>
    </methodAction>
  </bindings>
</pageDefinition>
```

The ADF Model layer loads the page definition from the path reference that appears in the `DataBinding.cpx` file. The new page definition file needs to have this reference to id "`UserInfoPageDef`" within `DataBindings.cpx`. This can be done from the Structure window for the CPX file.

To create a headless page definition file for the user interface project:

1. In the Application Navigator, expand the root user interface package and locate the `DataBindings.cpx` file. The packages appear in the Application Sources folder.
2. Double-click **DataBindings.cpx** and open the Structure window.
3. In the Structure window, select the **pageDefinitionUsages** node and choose **Insert Inside pageDefinitionUsages > page**.
4. Set **Id** to the name you specified for your headless page definition file (contains the single methodAction binding). For example, the SRDemo application uses `UserInfoPageDef`.
5. Set **path** to package that where you added the page definition file. For example, in the SRDemo application, the path is `oracle.srdemo.view.pageDefs.userInfo`.

At runtime, a reference to `data.<Headless_PageDefID>` will now resolve to this binding definition. Example 18–11 shows the id specified for the headless page definition file in the SRDemo application.

Example 18–11 SRDemo DataBindings.cpx Page Definition Reference

```
<?xml version="1.0" encoding="UTF-8" ?>
<Application xmlns="http://xmlns.oracle.com/adfm/application" ...
  <pageDefinitionUsages>
    <page id="SRListPageDef"
          path="oracle.srdemo.view.pageDefs.app_SRListPageDef"/>
    <page id="UserInfoPageDef"
          path="oracle.srdemo.view.pageDefs.headless_UserInfoPageDef"/>
     ...
</Application>
```

18.7.2.4 Executing the Session Facade Method from the UserInfo Bean

In the managed bean definition `userInfo`, you may have already defined a managed property `bindings` that has the value `#{data.UserInfoPageDef}`. For details, see Section 18.7.1.2, "Creating a Managed Bean for the Security Information".

To compliment the expression, the class that implements the security methods (`UseInfo.java`) requires a corresponding getter and setter method for the `bindings` property:

```
public void setBindings(BindingContainer bindings) {
   this._bindings = bindings;
 }

public BindingContainer getBindings() {
   return _bindings;
}
```

The first time the application requires the `UserId`, the session bean method is called. This is done using the `getUserId()` method in `UserInfo.java`. The `getUserId()` method checks to see if the `UserId` is currently populated. If not, it makes a call to a private method `lookupUserId()` that actually calls the session facade method:

```
public Integer getUserId() {
   if (_userId == null){
    _userId = lookupUserId(_userName);
   }
   return _userId;
}
```

The `lookupUserId()` method is responsible for invoking the methodAction binding which calls the session facade method defined to get the user ID:

```
private Integer lookupUserId(String userName) {
        if (getBindings() != null) {
            OperationBinding oper = (OperationBinding)getBindings().
                                getOperationBinding("findUserByEmail");
            //now set the argument to the function with the username we
            //are interested in
            Map params = oper.getParamsMap();
            params.put("emailParam",userName);
            // And execute
            User user = (User)oper.execute();
            setUserobject(user);
            return user.getUserId();
        }
}
```

The method uses `getBindings()` to get the injected binding container from the Faces configuration. Once the binding container is obtained, the method looks up the methodAction binding responsible for coordinating with the session facade method. For details about the session facade method, see Section 18.7.2.4, "Executing the Session Facade Method from the UserInfo Bean".

19

Advanced TopLink Topics

This chapter describes how to use the advanced TopLink functions in the Mapping editor.

This chapter includes the following sections:

- Section 19.1, "Introduction to Advanced TopLink Topics"
- Section 19.2, "Using Advanced Parameters (databindings.cpx)"
- Section 19.3, "Configuring Method Access for Relationship"
- Section 19.4, "Using sessions.xml with a TopLink Data Control"
- Section 19.5, "Using Multiple Maps with a TopLink Data Control"
- Section 19.6, "Compiling TopLink Classes with Specific JDK Versions"

19.1 Introduction to Advanced TopLink Topics

The TopLink mappings (introduced in Chapter 3, "Building and Using Application Services") allow you to map Java objects to your database. When creating TopLink mappings, there are some functions that are not available from the Mapping editor. You will need to implement these functions in your Java code. Refer to the ***Oracle TopLink Developer's Guide*** for additional information.

19.2 Using Advanced Parameters (databindings.cpx)

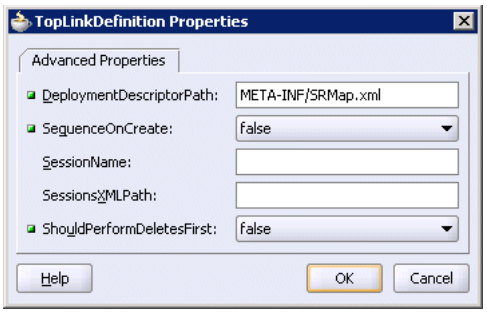
You can use the `databindings.cpx` file to override or modify the default TopLink data control behavior. This section includes information on the following options:

- Performing Deletes First
- Specifying the TopLink Session File
- Specifying the Sequencing

Refer to Appendix A, "Reference ADF XML Files" for additional information on parameters in the `databindings.cpx` file.

Use the TopLinkDefinitions Properties dialog (see Figure 19–1) define these parameters on the data control.

Figure 19–1 TopLinkDefinition Properties Dialog



19.2.1 Performing Deletes First

By default, the TopLink unit of work (see Section 3.8, "Creating and Modifying Objects with a Unit of Work") performs insert operations *before* delete operations. However, there may be instances in which you must perform the delete operation first.

For example, removing a row with a primary key of **1** and then creating a new row with the same primary key within the same transaction will result in a SQL exception indicating that the row already exists.

To eliminate this problem, use the **TopLinkShouldPerformDeletesFirst** parameter in the `databindings.cpx` file to force the unit of work to perform the delete operation first.

Example 19–1 Specifying the TopLinkShouldPerformDeletesFirst Option

```
...
<Parameter
   name="TopLinkShouldPerformDeletesFirst"
   value="True"
</Parameter
...
```

19.2.2 Specifying the TopLink Session File

By default, the TopLink session configuration file is named `sessions.xml`. You can create this file by using the Mapping editor in Oracle JDeveloper (refer to the Oracle JDeveloper online help for more information).

To specify a different sessions configuration file, use the **TopLinkSessionsXMLFileName** parameter in the `databindings.cpx` file.

Example 19–2 Specifying the TopLinkSessionsXMLFileName Option

```
...
<Parameter
   name="TopLinkSessionsXMLFileName"
   value="META-INF/sessions.xml"
</Parameter
...
```

19.2.3 Specifying the Sequencing

By default, the TopLink unit of work (see Section 3.8, "Creating and Modifying Objects with a Unit of Work") assigns sequence numbers during the commit operation. However, there may be instances in which the sequence number must be displayed in the user interface *before* the commit operation.

For example, if the sequence number is used as the value of an ID field in a form displayed to the user, you must have the sequence number before committing the transaction.

To eliminate this problem, use the **TopLinkSequenceOnCreate** parameter in the `databindings.cpx` file to disable the assigning of the sequence number during the commit operation of a create transaction

Example 19–3 Specifying the ToplinkSequenceOnCreate Option

```
...
<Parameter
   name="TopLinkSequenceOnCreate"
   value="False"
</Parameter
...
```

19.3 Configuring Method Access for Relationship

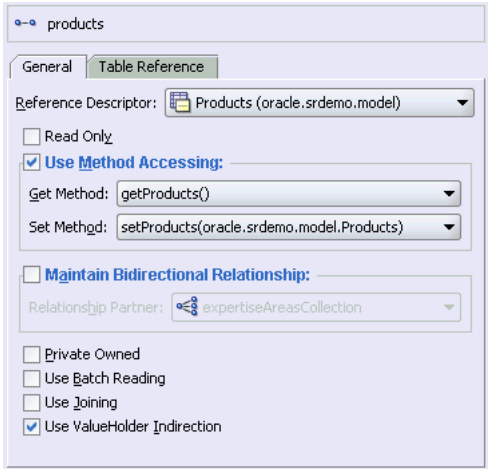
By default, TopLink mappings use direct access to access public attributes. Alternatively, you can use getter and setter methods to access object attributes when writing the attributes of the object to the database, or reading the attributes of the object from the database. This is known as method access.

Figure 19–2 shows a TopLink mapped attribute that uses method accessing.

To configure method accessing for a relationship:

1. Select a relationship mapping from a TopLink descriptor in the Structure window.
2. On the mapping's General tab, select the **Use Method Accessing** option.

Figure 19–2 General Tab of TopLink Mapping Editor



3. Select the specific getter and setter methods for the relationship.

19.4 Using sessions.xml with a TopLink Data Control

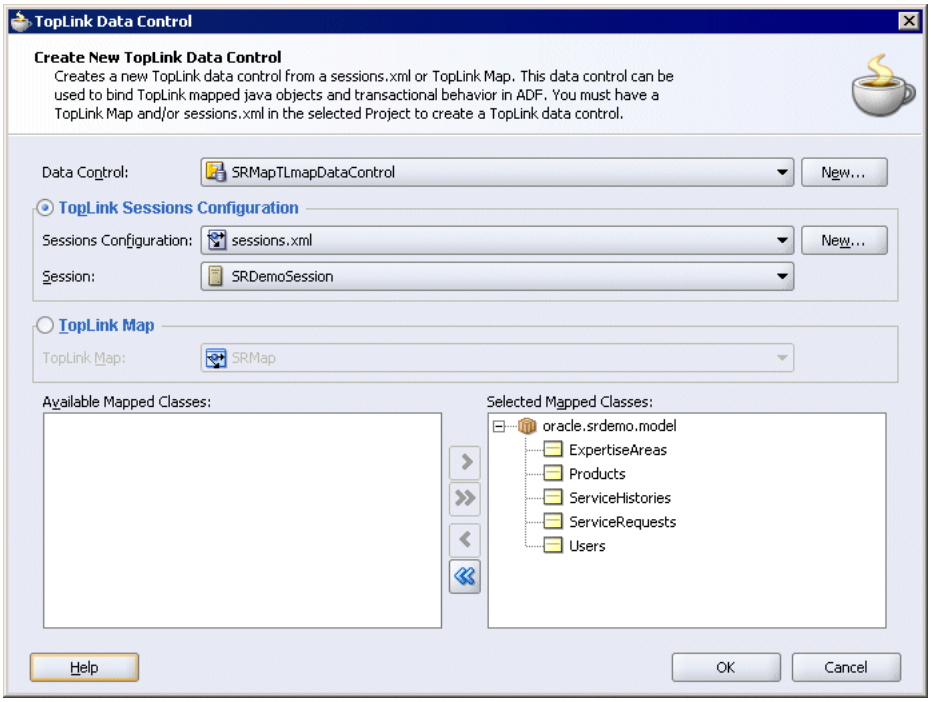
You can create a data control from a TopLink sessions configuration file (`sessions.xml`), similarly to creating a data control from a TopLink map (see Section 19.5, "Using Multiple Maps with a TopLink Data Control").

Use the TopLink Data Control dialog (as shown in Figure 19–3), select **TopLink Sessions Configuration**, and then select the specific sessions configuration file (`sessions.xml`) and session.

To create a TopLink data control from a sessions configuration file (sessions.xml):

1. Right-click the `sessions.xml` file in the Navigator window and select **Create Data Control**.
2. On the TopLink Data Control dialog, select the **TopLink Sessions Configuration** option.

Figure 19–3 Creating a TopLink Data Control (from a Sessions Configuration)

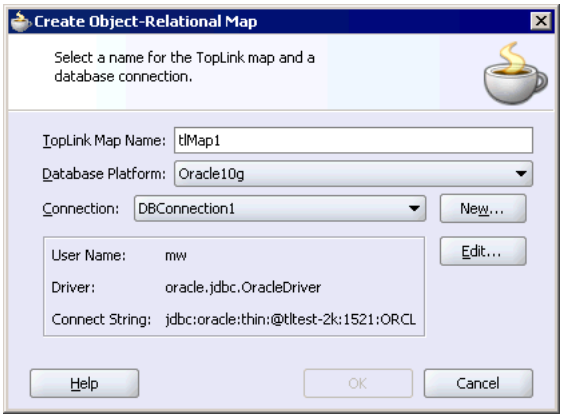


3. Select the specific sessions configuration file (or create a new configuration) and session. You can create a data control for any mapped classes.

19.5 Using Multiple Maps with a TopLink Data Control

You can create multiple TopLink maps for use with each project. Each map can be associated with a specific database and connection, as shown inFigure 19–4.

Figure 19–4 Create Object Relational Map Dialog



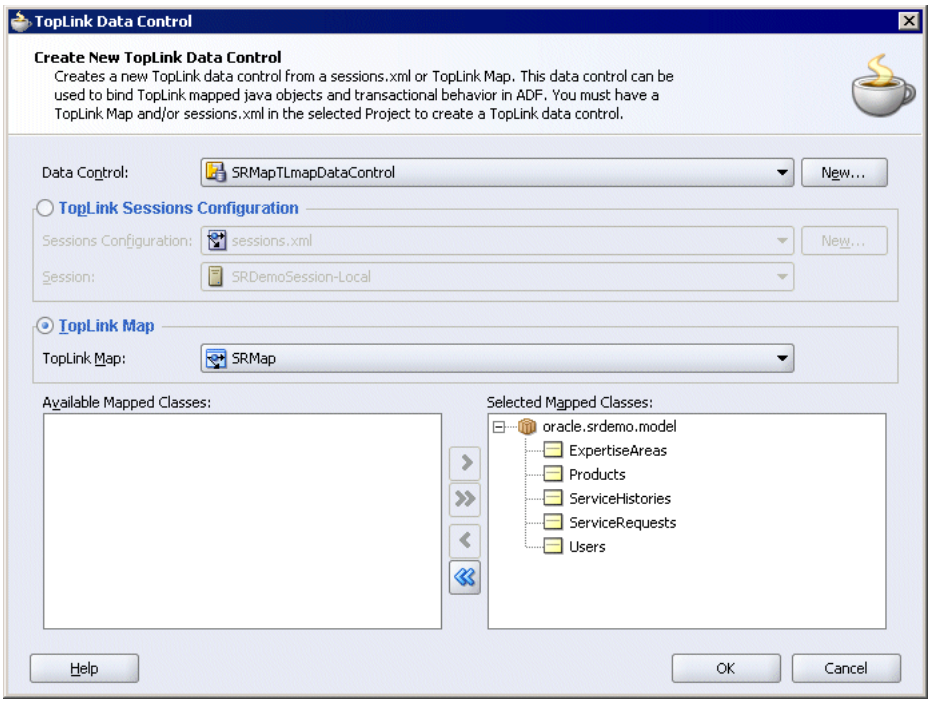
You can create a data control from a TopLink map, similarly to creating a data control from a `sessions.xml` file (see Section 19.4, "Using sessions.xml with a TopLink Data Control").

Use the TopLink Data Control dialog (as shown in Figure 19–5), select **TopLink Map**, and then select the specific map.

To create a TopLink data control from a TopLink map:

1. Right-click the `sessions.xml` file in the Navigator window and select **Create Data Control**.
2. On the TopLink Data Control dialog, select the **TopLink Map** option.

Figure 19–5 Creating a TopLink Data Control (from a TopLink Map)



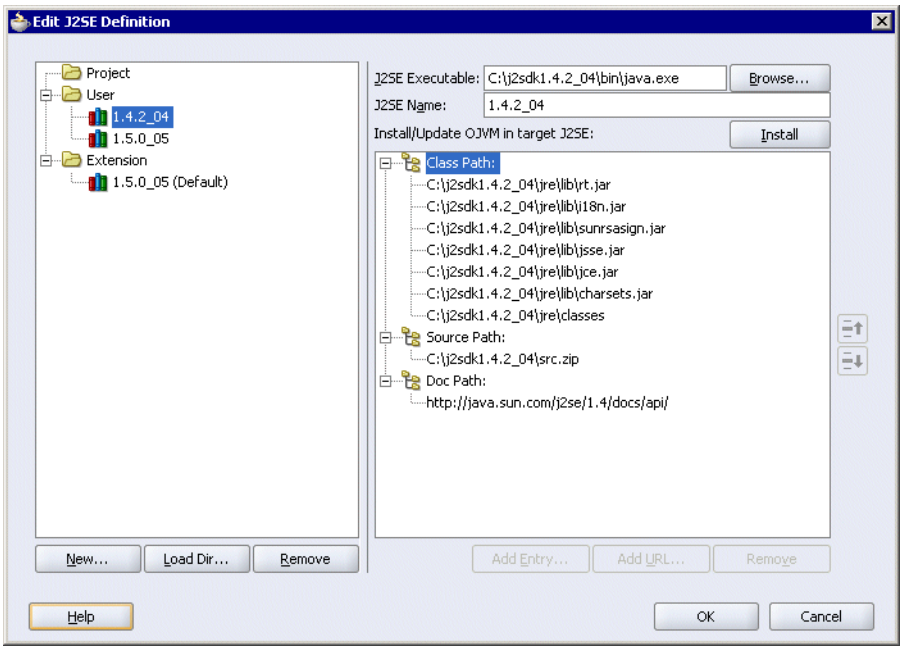
3. Select the specific TopLink map.
4. Select the specific sessions configuration file (or create a new configuration) and session. You can create a data control for any mapped classes.

19.6 Compiling TopLink Classes with Specific JDK Versions

By default, when compiling TopLink classes, JDeveloper uses JDK 1.5 generic collection types for relationships. This will causes errors if you compile your project using a different JDK version (such as 1.4).

Before generating TopLink mappings you must change the default JSEE library for your project. In the Default Project Properties Dialog – Libraries page, click **Change** to select a new J2SE definition for the project. On the Manage Libraries Dialog – Edit J2SE Definitions Page (see Figure 19–6), select a (or create a new) J2SE definition to use.

Figure 19–6 Edit J2SE Definition Page



20

Creating Data Control Adapters

If you need data controls beyond those that are provided by JDeveloper, you can create your own. ADF supports two main ways to create data controls:

- Create a JavaBean to represent the data source.
- Create a data control adapter for the data source type.

This chapter describes the second option: creating a data control adapter. For information about data controls, see Chapter 1, "Introduction to Oracle ADF Applications".

This chapter contains the following topics:

- Section 20.1, "Introduction to the Simple CSV Data Control Adapter"
- Section 20.2, "Overview of Steps to Create a Data Control Adapter"
- Section 20.3, "Implement the Abstract Adapter Class"
- Section 20.4, "Implement the Data Control Definition Class"
- Section 20.5, "Implement the Data Control Class"
- Section 20.6, "Create any Necessary Supporting Classes"
- Section 20.7, "Create an XML File to Define Your Adapter"
- Section 20.8, "Build Your Adapter"
- Section 20.9, "Package and Deploy Your Adapter to JDeveloper"
- Section 20.10, "Location of Javadoc Information"

20.1 Introduction to the Simple CSV Data Control Adapter

This chapter shows a simple CSV data control adapter as an example of a custom data control adapter. This adapter is a simplified version of the CSV data control adapter that ships with JDeveloper.

The chapter describes what the simple CSV data control adapter does and the classes that make up the adapter.

The simple CSV data control adapter retrieves comma-separated values from a file and displays them on a page. To use the adapter in JDeveloper, you can do one of the following:

- right-click a node that represents a CSV file and choose "Create Data Control" from the context menu
- drag and drop a node on the Data Control Palette

In either case, the node must map to a CSV text file, and the name of the file must have a `.csv` extension. You do not have to enter any metadata because the simple CSV data control adapter extracts the metadata from the node.

After you create a data control using the simple CSV adapter, the data control appears in the Data Control Palette. You can then drag and drop it onto a view page.

To simplify some details, the simple CSV adapter hardcodes the following items:

- The fields in the CSV file are comma-separated.
- The delimiter character is the double-quote character.
- The CSV file uses UTF-8 encoding.
- The first line in the file specifies column names.
- The name of the CSV file must have a `.csv` extension.

(The CSV adapter that ships with JDeveloper enables you to set these values.)

When you create a data control adapter, you create it so that it represents a source type, not a source instance. In the case of the CSV adapter, the source type is CSV files. To specify a specific data instance, for example, a particular CSV file, the user creates a data control with the help of the data control adapter and associates the instance with *metadata*. The metadata specifies the data for the instance. In the case of the simple CSV adapter, the metadata includes the path to a specific CSV file.

The responsibilities of a data control adapter include:

- Providing metadata for the data control instance
- Creating a data control instance using the stored metadata during runtime

Data control adapters run within the *adapter framework*. The adapter framework takes care of storing the metadata, integrating the data control adapter with the ADF lifecycle, and integrating with JDeveloper during design time.

20.2 Overview of Steps to Create a Data Control Adapter

To create data control adapters:

1. Create classes to extend abstract classes and implement interfaces in the adapter framework. These classes are used during design time and runtime. You have to create three classes as described in these sections:
   - Section 20.3, "Implement the Abstract Adapter Class"
   - Section 20.4, "Implement the Data Control Definition Class"
   - Section 20.5, "Implement the Data Control Class"

   You can also create additional classes as required by your adapter. For the simple CSV adapter, it includes two additional classes: `CSVHandler` and `CSVParser`. These classes are shown in Section 20.6, "Create any Necessary Supporting Classes".
2. Create a definition file, `adapter-definition.xml`, to register your adapter with ADF. This file contains the class name of your adapter implementation and references the libraries that your adapter needs to run. See Section 20.7, "Create an XML File to Define Your Adapter".
3. Install your data control adapter in JDeveloper by packaging your class files and the definition file in a JAR file and placing the JAR file in JDeveloper's classpath. See Section 20.9, "Package and Deploy Your Adapter to JDeveloper".

Invoking Your Adapter

After installing your data control adapter in JDeveloper, you can invoke it by right-clicking a node in JDeveloper that your data control adapter supports and selecting "Create Data Control" from the context menu. The data control adapter declares the node types that it supports in its `adapter-definition.xml` configuration file (described in Section 20.7, "Create an XML File to Define Your Adapter").

For example, if your adapter supports database connection nodes, when you right-click on a database connection, then you can select Create Data Control from the context menu to invoke your adapter.

Note that this chapter does not cover how to create a wizard, or how to pass values from a wizard to your adapter.

20.3 Implement the Abstract Adapter Class

Implementing the `AbstractAdapter` class is optional. It is required only if you want to enable the user to create a data control by dragging and dropping a node onto the Data Control Palette. In this case, the dropped node represents the data source associated with the data control that you are creating. If you do not want this feature, you do not have to implement this class. For example, the CSV data control adapter that ships with JDeveloper does not implement this class because it does not support the drag-and-drop operation. Instead, this adapter displays a wizard to collect information from the user.

The simple CSV adapter implements the `AbstractAdapter`. When the user drags and drops a node onto the Data Control Palette, JDeveloper checks to see which adapter can handle the type of node that was dropped. You specify the node types that your adapter can handle in the `adapter-definition.xml` file. This file is used to register your adapter with JDeveloper. See Section 20.7, "Create an XML File to Define Your Adapter" for details about this file.

In your class, you have to implement some methods in the `AbstractAdapter` class, as described in these sections:

- Section 20.3.4, "Implementing the initialize Method"
- Section 20.3.5, "Implementing the invokeUI Method"
- Section 20.3.6, "Implementing the getDefinition Method"

20.3.1 Location of JAR Files

The abstract class `oracle.adf.model.adapter.AbstractAdapter` is located in the `JDEV_HOME/bc4j/lib/adfm.jar` file.

20.3.2 Abstract Adapter Class Outline

Example 20–1 shows an outline of a class that implements the `AbstractAdapter` class.

Example 20–1 Outline for Class That Implements AbstractAdapter

```
import oracle.adf.model.adapter.AbstractAdapter;
import oracle.adf.model.adapter.DTContext;
import oracle.adf.model.adapter.AbstractDefinition;

public class MyAdapter extends AbstractAdapter
{
   public void initialize(Object sourceObj, DTContext ctx)
   {
      // you need to implement this method.
      // see Section 20.3.4, "Implementing the initialize Method".
   }

   public boolean invokeUI()
   {
      // you need to implement this method.
      // see Section 20.3.5, "Implementing the invokeUI Method".
   }

   public AbstractDefinition getDefinition()
   {
      // you need to implement this method.
      // see Section 20.3.6, "Implementing the getDefinition Method".
   }
}
```

20.3.3 Complete Source for the SampleDCAdapter Class

Example 20–2 shows the complete source for the `SampleDCAdapter` class. This is the class that implements `AbstractAdapter` for the simple CSV adapter. Subsequent sections describe the methods in this class.

Example 20–2 Complete Source for SampleDCAdapter

```
package oracle.adfinternal.model.adapter.sample;

import java.net.URL;

import oracle.adf.model.adapter.AbstractAdapter;
import oracle.adf.model.adapter.AbstractDefinition;
import oracle.adf.model.adapter.DTContext;

import oracle.ide.Context;

public class SampleDCAdapter extends AbstractAdapter
{
   // JDev Context
   private Context              mJdevCtx    = null;

   // Source object of data
   private Object               mSrc        = null;
   // Source Location
   private String               mSrcLoc     = null;
```

```
// data control name
private String               mDCName     = null;
// data control definition
private AbstractDefinition   mDefinition = null;

public SampleDCAdapter()
{
}

/**
 * Initializes the adapter from a source object.
 * <p>
 * The source object can be different thing depending on the context of the
 * design time that the adapter is used in. For JDeveloper, the object will
 * be a JDeveloper node.
 * </p>
 * <p>
 * Adapter implementations will check the <code>"ctx"</code> parameter to
 * get the current design time context. The source object will be used to
 * extract the information for the data source.
 * </p>
 * @param sourceObj Object that contains information about the data source
 *            that will be used to define the data control.
 * @param ctx Current design time context.
 */
public void initialize(Object sourceObj, DTContext ctx)
{
   mSrc = sourceObj;
   mJdevCtx = (Context) ctx.get(DTContext.JDEV_CONTEXT);
}

/**
 * Invlokes the UI at the design time.
 * <p>
 * This method is a call back from the JDeveloper design time environment to
 * the adapters to bring up any UI if required to gather information about
 * the data source they represent.
 * </p>
 *
 * @return false if the user cancels the operation. The default retrun value
 * is true.
 */
public boolean invokeUI()
{
   // First check if this is a JDev environment.
   if (mJdevCtx != null && mSrc != null)
   {
      if (extractDataSourceInfo(mSrc))
      {
         SampleDCDef def = new SampleDCDef(mSrcLoc,mDCName);
         mDefinition = def;
         return true;
      }
      return false;
   }
   return false;
}

 /**
 * <p>
```

```
     * The Definition instance obtained can be used by the ADF design time to
     * capture the data control metadata.
     *</p>
     *
     * @return The definition instance describing the data control design time.
     */
    public AbstractDefinition getDefinition()
    {
       return mDefinition;
    }

    /**
     * @param source the data source object.
     * @return false if data type is unknown.
     */
    public boolean canCreateDataControl(Object source)
    {
       return extractDataSourceInfo(source);
    }

    /**
     * Extracts information from a data source. This method extracts name
     * from the object.
     * @param obj the data source object.
     */
    private boolean extractDataSourceInfo(Object obj)
    {
       mDCName = "SampleDC";

       // See if the node dropped is a text node of CSV type.
       // We will assume that the CSV data file must end with .csv
       if (obj instanceof oracle.ide.model.TextNode)
       {
          oracle.ide.model.TextNode tn = (oracle.ide.model.TextNode) obj;
          URL url = tn.getURL();
          String loc = url.getFile();
          // Check if the file has a matching extension
          if (loc.endsWith(".csv"))
          {
             mSrcLoc = loc;
             String path = url.getPath();
             int index = path.lastIndexOf('/');

             if (index != -1)
             {
                String fileName = path.substring(index+1);
                int dotIndex = fileName.lastIndexOf('.');
                mDCName = fileName.substring(0,dotIndex);
             }
             return true;
          }
       }
       return false;
    }

}
```

20.3.4 Implementing the initialize Method

The framework calls the `initialize` method when the user drags and drops a node onto the Data Control Palette. The method has the following signature:

Example 20–3 initialize Signature

```
public abstract void initialize(Object sourceObj, DTContext ctx);
```

The *`sourceObj`* parameter specifies the node that was dropped. You can check this to ensure that the node type is something your adapter can handle.

The *`ctx`* parameter specifies the design time context. The package path for DTContext is `oracle.adf.model.adapter.DTContext`.

In the `initialize` method, you should perform these tasks:

- check if the source node is something that you support
- if you support the node, then extract all the information that you need to create a data control instance from the source node. If the information is not sufficient to create a data control instance, you can display some UI in the `invokeUI` method to get the user to enter the required information.

For the simple CSV adapter, the `initialize` method simply sets some class variables. These class variables are checked later in the `invokeUI` method.

Example 20–4 initialize Method

```
public void initialize(Object sourceObj, DTContext ctx)
{
   mSrc = sourceObj;
   mJdevCtx = (Context) ctx.get(DTContext.JDEV_CONTEXT);
}
```

20.3.5 Implementing the invokeUI Method

This method enables you to display any UI to collect information from the user about the dropped data source. The method has the following signature in the `AbstractAdapter`:

Example 20–5 invokeUI Signature

```
public boolean invokeUI()
{
   return true;
}
```

The method should return `false` if the user cancels the operation in the UI. This means that the data control is not created.

The method should return `true` (which is the default implementation) if the UI was run to collect the information.

The simple CSV adapter uses the `initialize` method to call `extractDataSourceInfo`, which performs the following:

- checks that the node right-clicked by the user represents a text file and that the filename has a `.csv` extension
- gets the filename of the CSV file

- sets the `mSrcLoc` and `mDCName` class variables. `mSrcLoc` points to the location of the CSV file, and `mDCName` is the name used for the data control. In this case, it is just the name of the CSV file without the `.csv` extension.

  These variables are used by `invokeUI` to instantiate a `SampleDCDef` object. The `SampleDCDef` object, which is another class you have to implement, is described in Section 20.4, "Implement the Data Control Definition Class".

Example 20–6 shows the `invokeUI` method:

Example 20–6 invokeUI

```
public boolean invokeUI()
{
   // First check if this is a JDev environment.
   if (mJdevCtx != null && mSrc != null)
   {
      if (extractDataSourceInfo(mSrc))
      {
         SampleDCDef def = new SampleDCDef(mSrcLoc,mDCName);
         mDefinition = def;
         return true;
      }
      return false;
   }
   return false;
}
```

20.3.6 Implementing the getDefinition Method

This method returns the definition of the data control that was created from information gathered from the dropped source node. The method has the following signature:

Example 20–7 getDefinition Signature

```
public abstract AbstractDefinition getDefinition();
```

The `AbstractDefinition` class is the data control definition class that you created. See Section 20.4, "Implement the Data Control Definition Class".

In the simple CSV adapter, the `getDefinition` method returns the value of the `mDefinition` class variable, which was set in the `invokeUI` method. `mDefinition` refers to the data control definition class that you created (`SampleDCDef` in the case of the simple CSV adapter).

Example 20–8 getDefinition

```
public AbstractDefinition getDefinition()
{
   return mDefinition;
}
```

20.4 Implement the Data Control Definition Class

This class needs to provide all the information that the framework needs to instantiate a data control during design time and runtime. This class is responsible for performing these operations:

- creating a default constructor. See Section 20.4.4, "Creating a Default Constructor".
- collecting metadata from the user about the data source. See Section 20.4.5, "Collecting Metadata from the User".
- defining the structure of the output. The structure defines what the user sees when the user expands the data control in the Data Control Palette. The user can then drag elements from the data control entry in the Data Control Palette to a page to create a view component. See Section 20.4.6, "Defining the Structure of the Data Control".
- creating an instance of the data control class using that metadata. The data control class is a class that you implement. See Section 20.4.7, "Creating an Instance of the Data Control".
- enabling the framework to load the metadata from the DCX file. See Section 20.4.8, "Setting the Metadata for Runtime".
- setting a name for your data control. See Section 20.4.9, "Setting the Name for the Data Control".

20.4.1 Location of JAR Files

The data control definition class needs to extend the abstract class `oracle.adf.model.adapter.AbstractDefinition`. This class is located in the `JDEV_HOME/bc4j/lib/adfm.jar` file.

20.4.2 Data Control Definition Class Outline

Example 20–9 is an outline showing the methods you have to implement when you create a data control definition class. The sample is taken from `SampleDCDef`, which is the data control definition class for the simple CSV data control adapter.

Example 20–9 Outline for the Data Control Definition Class

```
import oracle.adf.model.adapter.AbstractDefinition;
import org.w3c.dom.Node;
import oracle.binding.meta.StructureDefinition;
import oracle.binding.DataControl;
import java.util.Map;

public class SampleDCDef extends AbstractDefinition
{
   // default constructor
   public SampleDCDef ()
   {
      // you need a default constructor.
      // see Section 20.4.4, "Creating a Default Constructor".
   }

   public Node getMetadata()
   {
      // you need to implement this method.
      // see Section 20.4.5, "Collecting Metadata from the User".
   }
```

```
   public StructureDefinition getStructure()
   {
      // you need to implement this method.
      // see Section 20.4.6, "Defining the Structure of the Data Control".
   }

   public DataControl createDataControl()
   {
      // you need to implement this method.
      // see Section 20.4.7, "Creating an Instance of the Data Control".
   }

   public void loadFromMetadata(Node node, Map params)
   {
      // you need to implement this method.
      // see Section 20.4.8, "Setting the Metadata for Runtime".
   }

   public String getDCName()
   {
      // you need to implement this method.
      // see Section 20.4.9, "Setting the Name for the Data Control".
   }
}
```

20.4.3 Complete Source for the SampleDCDef Class

Example 20–10 shows the complete source for the `SampleDCDef` class:

Example 20–10 Complete Source for the SampleDCDef Class

```
package oracle.adfinternal.model.adapter.sample;

import java.io.InputStream;

import java.util.Map;
import oracle.binding.DataControl;
import oracle.binding.meta.StructureDefinition;

import oracle.adf.model.adapter.AbstractDefinition;

import oracle.adf.model.adapter.AdapterDCService;
import oracle.adf.model.adapter.AdapterException;
import oracle.adf.model.adapter.dataformat.AccessorDef;
import oracle.adf.model.adapter.dataformat.StructureDef;
import oracle.adf.model.adapter.utils.NodeAttributeHelper;

import oracle.adf.model.utils.SimpleStringBuffer;

import oracle.adfinternal.model.adapter.sample.CSVHandler;
import oracle.adfinternal.model.adapter.sample.SampleDataControl;
import oracle.adfinternal.model.adapter.url.SmartURL;

import oracle.xml.parser.v2.XMLDocument;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
```

```
public class SampleDCDef extends AbstractDefinition
{
   // Name of the root accessor for a definition
   public static final String RESULT_ACC_NAME = "Result";

   // Namespace for the metadata definition.
   public static final String SAMPLEDC_NS =
                           "http://xmlns.oracle.com/adfm/adapter/sampledc";

   // Definition tag as the root
   public static final String DEFINITION = "Definition";

   // Attribute to contain the source URL
   public static final String SOURCE_LOC = "SourceLocation";

   // Name of the data control
   private String mName = "SampleDC";

   // the structure definition
   private StructureDef mStructDef = null;

   // URL for this definition.
   private String mCSVUrl = null;

   public SampleDCDef()
   {
   }

   public SampleDCDef(String csvURL,String dcName)
   {
      mCSVUrl = csvURL;
      mName = dcName;
   }

   public Node getMetadata()
   {
      XMLDocument xDoc = new XMLDocument();
      Element metadata = xDoc.createElementNS(SAMPLEDC_NS, DEFINITION);
      metadata.setAttribute(SOURCE_LOC, mCSVUrl.toString());
      return metadata;
   }

   public StructureDefinition getStructure()
   {
      if (mStructDef == null)
      {
         // create an empty StructureDefinition
         mStructDef = new StructureDef(getName());
         SmartURL su = new SmartURL(mCSVUrl.toString());
         InputStream isData = su.openStream();
         CSVHandler csvHandler = new CSVHandler(isData, true, "UTF-8", ",", "\"");

         // Name of the accessor or the method structure to hold the attributes
         String opName = new SimpleStringBuffer(50).append(getDCName())
                                                   .append("_")
                                                   .append(RESULT_ACC_NAME)
                                                   .toString();

         StructureDef def = (StructureDef)csvHandler.getStructure(opName, null);
```

```
      // Create the accessor definition
      AccessorDef accDef =
                    new AccessorDef(RESULT_ACC_NAME, mStructDef, def, true);
      def.setParentType(StructureDef.TYPE_ACCESSOR);
      accDef.setBindPath(new SimpleStringBuffer(50)
                                    .append(mStructDef.getFullName())
                                    .append(".")
                                    .append(AdapterDCService.DC_ROOT_ACC_NAME)
                                    .toString());
      mStructDef.addAccessor(accDef);
   }
   return mStructDef;
}

public void loadFromMetadata(Node node, Map params)
{
   try
   {
      // Get the information from the definition
      NodeList listChld = node.getChildNodes();
      int cnt = listChld.getLength();
      Node chld;

      for (int i = 0; i < cnt; i++)
      {
         chld = listChld.item(i);
         // System.out.println("Tag: " + chld.getNodeName());
         if (DEFINITION.equalsIgnoreCase(chld.getNodeName()))
         {
            // Load the required attributes
            NodeAttributeHelper attribs =
            new NodeAttributeHelper(chld.getAttributes());
            mCSVUrl = attribs.getValue(SOURCE_LOC);
         }
      }
   }
   catch (AdapterException ae)
   {
      throw ae;
   }
   catch (Exception e)
   {
      throw new AdapterException(e);
   }
}

public DataControl createDataControl()
{
   SampleDataControl dcDataControl = new SampleDataControl(mCSVUrl);
   return dcDataControl;
}

public String getDCName()
{
   return mName;
}
```

```
   public String getAdapterType()
   {
   return "oracle.adfm.adapter.SampleDataControl";
   }
}
```

20.4.4 Creating a Default Constructor

You need to create a default constructor for the data control definition class. The simple CSV adapter has an empty default constructor:

Example 20–11 SampleDCDef Default Constructor

```
public SampleDCDef()
{
}
```

The default constructor is used only during runtime. It is not used during design time.

20.4.5 Collecting Metadata from the User

Metadata in a data control adapter provides information on the data source. The data control definition class uses the metadata to create a data control. Examples of metadata for the full-featured CSV data control adapter include the URL to the CSV file, the field separator character, and the quote character. For the simple CSV adapter, the metadata consists of only the location of the CSV file.

A data control adapter can collect metadata in different ways. Examples:

- The CSV data control adapter that comes with JDeveloper uses a wizard to collect metadata from the user.
- The web service data control adapter also uses a wizard to collect metadata. Alternatively, users can drag a web service connection node and drop it on the Data Control Palette. The web service adapter extracts metadata from the node instead of launching the wizard.

  When the user drags and drops a node onto the Data Control Palette, the adapter framework looks for an adapter that can handle the type of node that was dropped by searching the registered data control adapters. Data control adapters declare which node types they support. The nodes are JDeveloper nodes that represent specific source types. When the framework finds an adapter that supports the type of node that was dropped, it invokes the data control adapter, which then extracts the required information from the node.
- The simple CSV adapter extracts metadata from a node when the user right-clicks a node and selects "Create Data Control" from the context menu.

Regardless of how a data control adapter retrieves the metadata, you must implement the `getMetadata` method in your data control definition class. The framework calls the method to get the metadata.

This method returns the metadata in the form of a `Node` object. The `getMetadata` method has the following signature:

Example 20–12 getMetadata Signature

```
public org.w3c.dom.Node getMetadata();
```

In the simple CSV adapter, the `getMetadata` method retrieves the metadata from the `mCSVUrl` class variable and inserts the value in an `Element` object.

Example 20–13 getMetadata Method

```
public Node getMetadata()
{
   XMLDocument xDoc = new XMLDocument();
   Element metadata = xDoc.createElementNS(SAMPLEDC_NS, DEFINITION);
   metadata.setAttribute(SOURCE_LOC, mCSVUrl.toString());
   return metadata;
}
```

The framework extracts the information from `getMetadata`'s return value (the `Node` object) and writes the information to the `DataControls.dcx` file. For example, after the user has created a CSV data control, the file looks like the following:

Example 20–14 DataControls.dcx File

```
<?xml version="1.0" encoding="UTF-8" ?>
<DataControlConfigs xmlns="http://xmlns.oracle.com/adfm/configuration"
                  version="10.1.3.36.45" Package="view" id="DataControls">

   <AdapterDataControl id="testdata"
                  FactoryClass="oracle.adf.model.adapter.DataControlFactoryImpl"
                  ImplDef="oracle.adfinternal.model.adapter.sample.SampleDCDef"
                  SupportsTransactions="false"
                  SupportsSortCollection="false" SupportsResetState="false"
                  SupportsRangesize="false" SupportsFindMode="false"
                  SupportsUpdates="false" Definition="testdata"
                  BeanClass="testdata"
                  xmlns="http://xmlns.oracle.com/adfm/datacontrol">

      <Source>
         <Definition
               SourceLocation="/C:/Application1/ViewController/public_
html/testdata.csv"/>
      </Source>
   </AdapterDataControl>
</DataControlConfigs>
```

The value of the `id` attribute of the `AdapterDataControl` tag ("`testdata`") is extracted from the name of the CSV file. The other attributes in the `AdapterDataControl` tag contain information about the simple CSV adapter itself. In the `Definition` element, the framework writes the metadata provided by the node; the `SourceLocation` attribute specifies the location of the CSV file.

20.4.6 Defining the Structure of the Data Control

Structure in a data control definition describes the items that appear when the user expands the data control in the Data Control Palette. Items that can appear include methods, accessors, and attributes of the underlying service that are available to the user to invoke or display. The user can drag these items onto a view page.

In your data control definition class, you need to implement the `getStructure` method. The framework calls this method when the user expands the data control in the Data Control Palette.

The `getStructure` method has the following signature:

Example 20–15 getStructure Signature

```
public oracle.binding.meta.StructureDefinition getStructure();
```

`StructureDefinition` is an interface. You can find more information about this interface in the online help in JDeveloper, under Reference > Oracle ADF Model API Reference.

Example 20–16 getStructure Method

```
public StructureDefinition getStructure()
{
   if (mStructDef == null)
   {
      // create an empty StructureDefinition
      mStructDef = new StructureDef(getName());
      SmartURL su = new SmartURL(mCSVUrl.toString());
      InputStream isData = su.openStream();
      CSVHandler csvHandler = new CSVHandler(isData, true, "UTF-8", ",", "\"");

      // Name of the accessor or the method structure to hold the attributes
      String opName = new SimpleStringBuffer(50).append(getDCName())
                                                 .append("_")
                                                 .append(RESULT_ACC_NAME)
                                                 .toString();

      StructureDef def = (StructureDef)csvHandler.getStructure(opName, null);
      // Create the accessor definition
      AccessorDef accDef =
                       new AccessorDef(RESULT_ACC_NAME, mStructDef, def, true);
      def.setParentType(StructureDef.TYPE_ACCESSOR);
      accDef.setBindPath(new SimpleStringBuffer(50)
                                        .append(mStructDef.getFullName())
                                        .append(".")
                                        .append(AdapterDCService.DC_ROOT_ACC_NAME)
                                        .toString());
      mStructDef.addAccessor(accDef);
   }
   return mStructDef;
}
```

20.4.7 Creating an Instance of the Data Control

The framework calls the `createDataControl` method in the data control definition class to create a data control instance. The `createDataControl` method has the following signature:

Example 20–17 createDataControl Signature

```
public oracle.binding.DataControl createDataControl();
```

The `DataControl` object returned by the method is an instance of the data control class that you create. Section 20.5, "Implement the Data Control Class" describes this class.

In the data control definition for the simple CSV adapter, the `createDataControl` method looks like the following:

Example 20–18 createDataControl Method

```
public DataControl createDataControl()
{
   SampleDataControl dcDataControl = new SampleDataControl(mCSVUrl);
   return dcDataControl;
}
```

The `SampleDataControl` class is described in more detail in Section 20.5, "Implement the Data Control Class".

20.4.8 Setting the Metadata for Runtime

When the user runs the view page that references your data control, the framework reads the metadata from the DCX file and invokes the `loadFromMetadata` method in the data control definition class to load the data control with the metadata saved during design time.

Recall that the framework wrote the metadata to the DCX file in the `getMetadata` method. See Section 20.4.5, "Collecting Metadata from the User".

The `loadFromMetadata` method has the following signature:

Example 20–19 loadFromMetadata Signature

```
public void loadFromMetadata(org.w3c.dom.Node node, java.util.Map params);
```

The *`node`* parameter contains the metadata. In the simple CSV adapter, the method looks like the following:

Example 20–20 loadFromMetadata Method

```
public void loadFromMetadata(Node node, Map params)
{
   try
   {
      // Get the information from the definition
      NodeList listChld = node.getChildNodes();
      int cnt = listChld.getLength();
      Node chld;

      for (int i = 0; i < cnt; i++)
      {
         chld = listChld.item(i);
```

```
            // System.out.println("Tag: " + chld.getNodeName());
            if (DEFINITION.equalsIgnoreCase(chld.getNodeName()))
            {
               // Load the required attributes
               NodeAttributeHelper attribs =
               new NodeAttributeHelper(chld.getAttributes());
               mCSVUrl = attribs.getValue(SOURCE_LOC);
            }
         }
      }
      catch (AdapterException ae)
      {
         throw ae;
      }
      catch (Exception e)
      {
         throw new AdapterException(e);
      }
   }
```

20.4.9 Setting the Name for the Data Control

You need to implement the `getDCName` method to return a string that is used to identify the data control instance in the Data Control Palette. `getDCName` has the following signature:

Example 20–21 getDCName Signature

```
public String getDCName();
```

In the simple CSV adapter, the method just returns the value of the `mName` class variable, which was set by the `SampleDCDef(String csvURL, String dcName)` constructor. This constructor was called in the `SampleDCAdapter` class. `mName` is the name of the CSV file without the `.csv` extension.

Example 20–22 getDCName Method

```
public String getDCName()
{
   return mName;
}
```

Note that each data control instance must have a unique name within an application. For example, if you have two CSV data controls in an application, you can name them "CSV1" and "CSV2". For the CSV data control adapter that is shipped with JDeveloper, the user can enter the name in the wizard. For the simple CSV adapter, the name is the name of the CSV file without the `.csv` extension.

20.5 Implement the Data Control Class

The data control class must be able to access the data source based on the metadata that was saved during design time. This class is instantiated by the `createDataControl` method in the data control definition class (see Section 20.4.7, "Creating an Instance of the Data Control").

This class needs to:

- Extend the abstract class `oracle.adf.model.AbstractImpl`.
- Implement one of the following data control interfaces:

Table 20–1 Data Control Interfaces

Interface	When to Use
`oracle.binding.DataControl`	Implement this interface if you do not need to demarcate the start and end of a request and if you do not need transactional support.
`oracle.binding.ManagedDataControl`	Implement this interface if you need to demarcate the start and end of a request. This interface extends `DataControl`, which means that you have to implement the methods in `DataControl` as well.
`oracle.binding.TransactionalDataControl`	Implement this interface if you need transactional support. The interface requires you to implement the `rollbackTransaction` and `commitTransaction` methods, in addition to the methods in the `DataControl` interface. (`TransactionalDataControl` extends the `DataControl` interface.)

20.5.1 Location of JAR Files

The abstract class `oracle.adf.model.AbstractImpl` is located in the `JDEV_HOME/bc4j/lib/adfm.jar` file.

The data control interfaces are located in the `JDEV_HOME/bc4j/lib/adfbinding.jar` file.

20.5.2 Data Control Class Outline

The following class outline for a data control class shows the methods you have to implement:

Example 20–23 Outline for a Data Control Class

```
import oracle.adf.model.adapter.AbstractImpl;
import oracle.binding.DataControl;
import java.util.HashMap;

public class SampleDataControl extends AbstractImpl implements ManagedDataControl
{
   public boolean invokeOperation(java.util.Map map,
                                  oracle.binding.OperationBinding action)
   {
      // you need to implement this method.
      // see Section 20.5.4, "Implementing the invokeOperation Method".
   }

   public String getName()
   {
      // you need to implement this method.
      // see Section 20.5.5, "Implementing the getName Method".
   }
```

```
   public void release(int flags)
   {
      // you need to implement this method.
      // see Section 20.5.6, "Implementing the release Method".
   }

   public Object getDataProvider()
   {
      // you need to implement this method.
      // see Section 20.5.7, "Implementing the getDataProvider Method".
   }
}
```

20.5.3 Complete Source for the SampleDataControl Class

Example 20–24 shows the complete source for the `SampleDataControl` class.

Example 20–24 Complete Source for the SampleDataControl Class

```
package oracle.adfinternal.model.adapter.sample;

import java.io.InputStream;

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

import javax.naming.Context;

import oracle.binding.ManagedDataControl;
import oracle.binding.OperationInfo;

import oracle.adf.model.adapter.AdapterException;
import oracle.adf.model.adapter.AbstractImpl;
import oracle.adf.model.adapter.dataformat.CSVHandler;

import oracle.adfinternal.model.adapter.url.SmartURL;

// Data control that represents a URL data source with CSV data format.
public class SampleDataControl extends AbstractImpl
   implements ManagedDataControl
{
   //URL to access the data source
   private String mCSVUrl = null;

   public SampleDataControl()
   {
   }

   public SampleDataControl(String csvUrl)
   {
      mCSVUrl = csvUrl;
   }
```

```
public boolean invokeOperation(java.util.Map map,
                               oracle.binding.OperationBinding action)
{
   Context ctx = null;
   try
   {
      // We are interested of method action binding only.
      if (action == null)
      {
         return false;
      }

      OperationInfo method = action.getOperationInfo();
      // No method defined, we are not interested.
      if (method == null)
      {
         return false;
      }

      // Execute only when the adapter execute is invoked
      if (METHOD_EXECUTE.equals(method.getOperationName()))
      {
         Object retVal = null;
         if (mCSVUrl != null)
         {
            SmartURL su = new SmartURL(mCSVUrl);
            InputStream isData = su.openStream();
            CSVHandler csvHandler =
                              new CSVHandler(isData,true,"UTF-8",",","\"");
            Map properties = new HashMap();
            retVal = csvHandler.getResult(properties);
         }

         Map rootDataRow = new java.util.HashMap(2);
         rootDataRow.put(SampleDCDef.RESULT_ACC_NAME, retVal);
         ArrayList aRes = new ArrayList(2);
         aRes.add(rootDataRow);

         processResult(aRes.iterator(), map, action);
         return true;
      }
   }
   catch (AdapterException ae)
   {
      throw ae;
   }
   catch (Exception e)
   {
      throw new AdapterException(e);
   }
   return false;
}

/**
* Perform request level initialization of the DataControl.
* @param requestCtx a HashMap representing request context.
*/
```

```
    public void beginRequest(HashMap requestCtx)
    {
    }

    /**
    * perform request level cleanup of the DataControl.
    * @param requestCtx a HashMap representing request context.
    */
    public void endRequest(HashMap requestCtx)
    {
    }

    /**
    * return false as resetState was deferred to endRequest processing
    */
    public boolean resetState()
    {
       return false;
    }

    /**
    * returns the name of the data control.
    */
    public String getName()
    {
       return mName;
    }

    /**
    * releases all references to the objects in the data provider layer
    */
    public void release(int flags)
    {
    }

    /**
    * Return the Business Service Object that this datacontrol is associated with.
    */
    public Object getDataProvider()
    {
       return null;
    }

}
```

20.5.4 Implementing the invokeOperation Method

You must implement the `invokeOperation` method in your data control class. The framework invokes this method when the user runs the view page.

This method is declared in the `DataControl` interface. The method has the following signature:

Example 20–25 invokeOperation Signature

```
public boolean invokeOperation(java.util.Map bindingContext,
                               oracle.binding.OperationBinding action);
```

The *bindingContext* parameter contains the return values fetched from the data source. The keys for retrieving the values are generated by the framework. Typically you do not need to process the values unless you need to filter or transform them.

The *action* parameter specifies the method that generated the values. The method could be a method supported by the underlying service, as in the case of a web service. The framework calls the data control even for some built-in actions if the data control wants to override the default behavior. You can check this parameter to determine if you need to process the action or not. For data controls that represent data sources that do not expose methods, the framework creates an action `AbstractImpl.METHOD_EXECUTE` to the execute the query for a data control.

The method should return `false` if it does not handle an action.

In the simple CSV adapter, the `invokeOperation` method checks that the method is METHOD_EXECUTE before fetching the data. It invokes the `CSVHandler` class, which invokes the `CSVParser` class, to get the data from the CSV file.

Example 20–26 invokeOperation Method

```
public boolean invokeOperation(java.util.Map map,
                               oracle.binding.OperationBinding action)
{
   Context ctx = null;
   try
   {
      // We are interested in method action binding only.
      if (action == null)
      {
         return false;
      }

      OperationInfo method = action.getOperationInfo();
      // No method defined, we are not interested.
      if (method == null)
      {
         return false;
      }

      // Execute only when the adapter execute is invoked
      if (METHOD_EXECUTE.equals(method.getOperationName()))
      {
         Object retVal = null;
         if (mCSVUrl != null)
         {
            SmartURL su = new SmartURL(mCSVUrl);
            InputStream isData = su.openStream();
            CSVHandler csvHandler =
                              new CSVHandler(isData, true, "UTF-8", ",", "\"");
            Map properties = new HashMap();
            retVal = csvHandler.getResult(properties);
         }
```

```
            Map rootDataRow = new java.util.HashMap(2);
            rootDataRow.put(SampleDCDef.RESULT_ACC_NAME, retVal);
            ArrayList aRes = new ArrayList(2);
            aRes.add(rootDataRow);

            processResult(aRes.iterator(), map, action);
            return true;
         }
      }
      catch (AdapterException ae)
      {
         throw ae;
      }
      catch (Exception e)
      {
         throw new AdapterException(e);
      }
      return false;
   }
```

Note that `invokeOperation` calls the `processResult` method after fetching the data. See the next section for details.

20.5.4.1 About Calling processResult

`invokeOperation` should call `processResult` to provide updated values to the framework. The method puts the result into the binding context for the framework to pick up. The method has the following syntax:

Example 20–27 processResult Syntax

```
public void processResult(Object result,
                          Map bindingContext,
                          oracle.binding.OperationBinding action)
```

In the *`result`* parameter, specify the updated values.

In the *`bindingContext`* parameter, specify the binding context. This is typically the same binding context passed into the `invokeOperation` method.

In the *`action`* parameter, specify the operation. This is typically the same action value passed into the `invokeOperation` method.

20.5.4.2 Return Value for invokeOperation

Return `true` from `invokeOperation` if you handled the action in the method. Return `false` if the action should be handled by the framework.

20.5.5 Implementing the getName Method

Implement the `getName` method to return the name of the data control as used in a binding context.

This method is declared in the `DataControl` interface. It has the following signature:

Example 20–28 getName Signature

```
public String getName();
```

In the simple CSV adapter, the method simply returns `mName`, which is a variable defined in the `AbstractImpl` class.

Example 20–29 getName Method

```
public String getName()
{
   return mName;
}
```

20.5.6 Implementing the release Method

The framework calls the release method to release all references to the objects in the data provide layer.

This method is declared in the `DataControl` interface. It has the following signature:

Example 20–30 release Signature

```
public void release(int flags);
```

The *`flags`* parameter indicate which references should be released:

- `REL_ALL_REFS`: The data control should release all references to the view and model objects.
- `REL_DATA_REFS`: The data control should release references to data provider objects.
- `REL_VIEW_REFS`: The data control should release all references to view or UI layer objects.

In the simple CSV data control adapter, the `release` method is empty. However, if your data control uses a connection, it should close and release the connection in this method.

20.5.7 Implementing the getDataProvider Method

This method returns the business service object associated with this data control.

This method is declared in the `DataControl` interface. It has the following signature:

Example 20–31 getDataProvider Signature

```
public Object getDataProvider();
```

In the simple CSV data control adapter, this method just returns null.

20.6 Create any Necessary Supporting Classes

In addition to the required classes, which implement ADF interfaces, you can create any supporting classes for your adapter, if necessary. The simple CSV adapter includes two supporting classes: `CSVHandler` and `CSVParser`. These classes read and parse the CSV files into rows and fields. See Section 20.11, "Contents of Supporting Files" for complete source listing for these classes.

20.7 Create an XML File to Define Your Adapter

To define your adapter for JDeveloper, create a file called `adapter-definition.xml` and place it in a directory called `meta-inf`. Note that the file and directory names are case-sensitive.

A typical `adapter-definition.xml` file contains the following entries:

Example 20–32 Description of adapter-definition.xml File

```
<AdapterDefinition>
   <Adapter Name="unique name for the adapter"
            ClassName="full name of class that implements AbstractAdapter">

      <Schema Namespace="name of schema that defines the data control metadata for
this adapter"
              Location="location of schema definition file"/>

      <Source>
         <Type Name="name of source type that the adapter can handle to create a
data control"
               JDevNode="full class name of supported node"/>
      </Source>

      <JDevContextHook Class="full name of class that provides the JDeveloper
context hook, if any"/>

      <Dependencies>
         <Library Path="full path name of the JAR file that the adapter needs in
order to run"/>
      </Dependencies>

   </Adapter>
</AdapterDefinition>
```

The `AdapterDefinition` tag is the container tag for all adapters.

Each `Adapter` tag describes an adapter. It has the following attributes:

- `Name` specifies a unique name for the adapter. The framework uses this name to identify the adapter.
- `ClassName` specifies the full Java class that implements the `AbstractAdapter`.

The `Schema` tag defines the namespace and the schema definition for the adapter metadata. JDeveloper registers the schema so that the metadata can be validated at design time. You can define all the namespaces and schemas supported by the adapters. This is optional.

The `Source` tag specifies the node (or data source) types that the adapter supports. It has the following attributes:

- `JDevNode` specifies the Java class for the supported node type. This node type can appear in JDeveloper's Connection Navigator.
- Name: any string

The `JDevContextHook` tag specifies additions to the context menu (the menu that appears when the user right clicks on the metadata node for the data control instance in the Structure Pane).

The `Dependencies` tag lists the library files that your adapter requires during runtime. The framework adds the library files to the project when the user uses a data control based on your adapter.

The `adapter-definition.xml` file for the simple CSV data control adapter looks like the following:

Example 20–33 adapter-definition.xml File for the Simple CSV Adapter

```
<AdapterDefinition>
   <Adapter Name="oracle.adfm.adapter.SampleDataControl"
            ClassName="oracle.adfinternal.model.adapter.sample.SampleDCAdapter">
      <Schema Namespace="http://xmlns.oracle.com/adfm/adapter/sample"
            Location="/oracle/adfinternal/model/adapter/sample/sampleDC.xsd"/>
      <Source>
         <Type Name="csvNode" JDevNode="oracle.ide.model.TextNode"/>
      </Source>
      <Dependencies>
         <Library Path="${oracle.home}/jlib/sampledc.jar"/>
      </Dependencies>
   </Adapter>
</AdapterDefinition>
```

The `sampleDC.xsd` file is shown in Section 20.11.1, "sampleDC.xsd".

20.8 Build Your Adapter

You need to add the following libraries to your project in order to build your adapter:

1. In the Project Properties dialog in JDeveloper, select **Libraries** on the left side.
2. Click **Add Library** on the right side and add the following libraries:
   - JSR-227 API
   - ADF Model Generic Runtime
   - Oracle XML Parser v2
3. Click **Add Jar/Directory** on the right side and add the following libraries:
   - `JDEV_HOME/ide/lib/ide.jar`
   - `JDEV_HOME/ide/lib/javatools.jar`
   - `JDEV_HOME/bc4j/jlib/dc-adapter.jar`

20.9 Package and Deploy Your Adapter to JDeveloper

Perform these steps to deploy your adapter to JDeveloper:

1. Create an `extension.xml` file in the `meta-inf` directory (the same directory that contains the `adapter-definition.xml` file).

   You need to do this because you are deploying the adapter as a JDeveloper extension. You use the `extension.xml` to add your JAR files to JDeveloper's classpath.

   The `extension.xml` file contains the following lines:

Example 20–34 extension.xml

```
<?xml version = '1.0' encoding = 'UTF-8'?>
<extension xmlns="http://jcp.org/jsr/198/extension-manifest"
           id="oracle.adfm.sample-adapters"
           version="10.1.3.36.45"
           esdk-version="1.0">
   <name>ADFm Sample Adapter</name>
   <owner>Oracle Corporation</owner>
   <dependencies>
      <import>oracle.BC4J</import>
      <import>oracle.j2ee</import>
   </dependencies>
   <classpaths>
      <classpath>../../BC4J/jlib/dc-adapters.jar</classpath>
      <classpath>../../jlib/sampledc.jar</classpath>
   </classpaths>

   <hooks>
      <!-- Adapter-specific data control library definitions -->
      <libraries xmlns="http://xmlns.oracle.com/jdeveloper/1013/jdev-libraries">
         <library name="Sample Data Control" deployed="true">
            <classpath>../../jlib/sampledc.jar</classpath>
         </library>
      </libraries>
   </hooks>
</extension>
```

For details on the tags in the `extension.xml` file, see the file `JDEV_HOME/jdev/doc/extension/ide-extension-packaging.html`.

2. Create a JAR file that contains the class files for your adapter, the `adapter-definition.xml` file, and the `extension.xml` file. The XML files must be in a `meta-inf` directory.

   For the simple CSV adapter, the JAR file is called sampledc.jar, and it contains the following files:

Example 20–35 sampledc.jar

```
connections.xml
extension/meta-inf/extension.xml
meta-inf/adapter-definition.xml
meta-inf/Manifest.mf
oracle/adfinternal/model/adapter/sample/CSVHandler$1.class
oracle/adfinternal/model/adapter/sample/CSVHandler.class
oracle/adfinternal/model/adapter/sample/CSVParser.class
oracle/adfinternal/model/adapter/sample/SampleDataControl.class
oracle/adfinternal/model/adapter/sample/SampleDCAdapter.class
oracle/adfinternal/model/adapter/sample/SampleDCDef.class
```

3. Copy the JAR file to the `JDEV_HOME/jlib` directory.
4. Create another JAR file to contain only the extension.xml file and the manifest file in the meta-inf directory. For the simple CSV adapter, the JAR file is called oracle.adfm.sampledc.10.1.3.jar, and it contains the following files:

Example 20–36 oracle.adfm.sampledc.10.1.3.jar

```
meta-inf/extension.xml
meta-inf/Manifest.mf
```

5. Copy the second JAR file (for example, `oracle.adfm.sampledc.10.1.3.jar`) to the `JDEV_HOME/jdev/extensions` directory.
6. Stop JDeveloper, if it is running.
7. Start JDeveloper. When you right-click on a node type that your adapter supports, you should see the "Create Data Control" menu item.

If you want more information on JDeveloper extensions, you can download the Extension SDK:

1. In JDeveloper, choose Help | Check for Updates. This starts the Check for Updates wizard.
2. On the Welcome page of the wizard, click **Next**.
3. On the Source page, select **Search Update Centers**, and select all the locations listed in that section. Click **Next**.
4. On the Updates page, select **Extension SDK**. Click **Next** to download and install the extension SDK.
5. On the Summary page, click **Finish**. You will need to restart JDeveloper so that it can access the Extension SDK files.

   For help on the Extension SDK, open the JDeveloper's online help, and navigate to Extending JDeveloper > Extending JDeveloper with the Extension SDK.

20.10 Location of Javadoc Information

The JDeveloper online help provides reference information for the classes described in this chapter in Javadoc format.

Table 20–2 Location of Javadoc

Class / Interface	Location of Javadoc in the Online Help

`AbstractDefinition`
`AbstractImpl`
`AbstractAdapter`
	Reference > Oracle ADF Model API Reference > Packages > oracle.adf.model.adapter > Class Summary

`StructureDefinition`	Reference > Oracle ADF Model API Reference > Packages > oracle.binding.meta > Interface Summary

`DataControl`
`ManagedDataControl`
`TransactionalDataControl`
	Reference > Oracle ADF Model API Reference > Packages > oracle.binding > Interface Summary

20.11 Contents of Supporting Files

This section shows the contents of the following files:

- Section 20.11.1, "sampleDC.xsd"
- Section 20.11.2, "CSVHandler Class"
- Section 20.11.3, "CSVParser"

20.11.1 sampleDC.xsd

Example 20–37 shows the contents of the `sampleDC.xsd` file.

Example 20–37 sampleDC.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://xmlns.oracle.com/adfm/adapter/test"
            xmlns="http://xmlns.oracle.com/adfm/adapter/test"
            elementFormDefault="qualified">
   <xsd:element name="Definition">
      <xsd:complexType>
         <xsd:attribute name="SourceLocation" type="xsd:string"/>
      </xsd:complexType>
   </xsd:element>
</xsd:schema>
```

20.11.2 CSVHandler Class

Example 20–38 shows the contents of the `CSVHandler` class.

Example 20–38 CSVHandler

```
package oracle.adfinternal.model.adapter.sample;

import java.io.InputStream;

import java.util.Iterator;
import java.util.List;
import java.util.Map;

import oracle.binding.meta.DefinitionContext;
import oracle.binding.meta.StructureDefinition;

import oracle.adf.model.utils.SimpleStringBuffer;

import oracle.adf.model.adapter.AdapterException;
import oracle.adf.model.adapter.dataformat.AttributeDef;
import oracle.adf.model.adapter.dataformat.StructureDef;
import oracle.adfinternal.model.adapter.sample.CSVParser;
import oracle.adf.model.adapter.utils.Utility;
```

```
/**
 * Format handler for character separated values.
 * <p>
 * This class generates structures according to the JSR 227 specification from
 * a CSV data stream by parsing the data. The data types are guessed from the
 * value of the first data line. It can extract values from a CSV data stream
 * as well.
 * <p>
 * Data controls that deals with CSV data can use this class to generate data
 * and structure.
 *
 * @version 1.0
 * @since 10.1.3
 */
public class CSVHandler
{
  // stream containing the data.
  private InputStream mDataStream;

  // if the first row contains the names
  private boolean mIsFirstRowNames = false;

  // Encoding styles
  private String mEncStyle;

  // Character value separator
  private String mDelimiter;

  // Character used to quote a multi-word string
  private String mQuoteChar;

  // Column names
  private List mColNames = null;

  ////////////////////////////// Constructors ////////////////////////////////

  /**
   * Creats a CSV format handler object.
   *
   * @param is input stream that contains the CSV data.
   * @param isFirstRowNames flag to indicate if the first row of the CSV data
   *        can be treated as column names.
   * @param encodingStyle encoding style of the data.
   * @param delim character value separators.
   * @param quoteChar value that can be treated as quote.
   */
  public CSVHandler(
    InputStream is,
    boolean isFirstRowNames,
    String encodingStyle,
    String delim,
    String quoteChar)
  {
    mDataStream = is;
    mIsFirstRowNames = isFirstRowNames;
    mEncStyle = encodingStyle;
    mDelimiter = delim;
    mQuoteChar = quoteChar;
}
```

```
//////////////////// Impl of FormatHandler ////////////////////////////////

/**
 * Returns the structure definition extracted for the data format.
 * <p>
 *
 * @param name name of the root structure.
 * @param ctx definition context information.
 * @return the structure information extracted.
 */
public StructureDefinition getStructure(String name, DefinitionContext ctx)
{
  StructureDef attrParent = null;
  try
  {
    CSVParser parser;

    if (mEncStyle == null)
    {
      parser = new CSVParser(mDataStream);
    }
    else
    {
      parser = new CSVParser(mDataStream, mEncStyle);
    }

    parser.setSeparators(mDelimiter.toCharArray());
    if (mQuoteChar != null && mQuoteChar.length() != 0)
    {
      parser.setQuoteChar(mQuoteChar.charAt(0));
    }

    // Get the column names
    Iterator colNames = getColNames(parser).iterator();

    // Create the structure definition
    attrParent = new StructureDef(name);

    // Parse the data to get the attributes
    if (mIsFirstRowNames)
    {
      parser.nextLine();
    }

    String[] vals = parser.getLineValues();
    if (vals != null)
    {
      int i = 0;
      while (colNames.hasNext())
      {
        String type = "java.lang.String";
        if (i < vals.length)
        {
          type = checkType(vals[i]);
          ++i;
        }
```

```
        AttributeDef attr =
          new AttributeDef((String) colNames.next(), attrParent, type);
        attrParent.addAttribute(attr);
      }
    }
    else
    {
      while (colNames.hasNext())
      {
        AttributeDef attr =
          new AttributeDef((String) colNames.next(),
                           attrParent, "java.lang.String");
        attrParent.addAttribute(attr);
      }
    }
  }
  catch (Exception e)
  {
    throw new AdapterException(e);
  }
  return attrParent;

}


/**
 * Returns the resulting data extracted from the input.
 * @param params parameters passed containig the context information.
 * @return <code>Iterator</code> of <code>Map</code> objects for the result.
 *         If no data found it can return null. The <code>Map</code>
 *         contains the value of attributes as defined in the data structure.
 *         For complex data, <code>Map</code>s can contain other iterator of
 *         <code>Map</code>s as well.
 */
public Iterator getResult(Map params)
{
  try
  {
    final CSVParser parser;
    if (mEncStyle == null)
    {
      parser = new CSVParser(mDataStream);
    }
    else
    {
      parser = new CSVParser(mDataStream, mEncStyle);
    }

    parser.setSeparators(mDelimiter.toCharArray());
    if (mQuoteChar != null && mQuoteChar.length() != 0)
    {
      parser.setQuoteChar(mQuoteChar.charAt(0));
    }

    final List cols = getColNames(parser);
    final boolean bEndOfData = (mIsFirstRowNames) ? !parser.nextLine() : false;
```

```
  //return the data iterator
  return new Iterator()
  {
    CSVParser _parser = parser;
    Iterator _colNames = cols.iterator();
    boolean _eof = bEndOfData;

    public void remove()
    {
    }

    public boolean hasNext()
    {
      return !_eof;
    }

    public Object next()
    {
        try
            {
              if (_eof)
              {
                return null;
              }

              java.util.HashMap map = new java.util.HashMap(5);

              // Create the current row as Map
              String[] data = _parser.getLineValues();
              int i = 0;
              while (_colNames.hasNext())
              {
                String val = null;
                if (i < data.length)
                {
                  val = data[i];
                }

                map.put(_colNames.next(), val);
                i++;
              }

              // get the next data line.
              _eof = !_parser.nextLine();

              return map;
        }
          catch (Exception e)
          {
            throw new AdapterException(e);
          }
        }

  };

}
```

```
  catch (AdapterException ae)
  {
    throw ae;
  }
  catch (Exception e)
  {
    throw new AdapterException(e);
  }
}


//=============================================================================
// Class Helper Methods
//=============================================================================

 /**
  * Attempts to obtain the Java type from the string value.
  * @param data String value whose datatype has to be guessed.
  * @return Java type name.
  */
 private String checkType(String data)
 {
   try
   {
     // We first try to convert the value into a long number.
     // If successful, we will use long; if it throws NumberFormatException,
     // we will attempt to convert it to float. If this too fails, we return
     // string.
     if (data != null)
     {
       try
       {
         // Try to conver the value into an integer number.
         long numTest = Long.parseLong(data);
         return "java.lang.Long"; //NOTRANS
       }
       catch (NumberFormatException nfe)
       {
         // Try to convert the value into float number.
         float numTest = Float.parseFloat(data);
         return "java.lang.Float"; //NOTRANS
       }
     }
     else
     {
       return "java.lang.String"; //NOTRANS
     }
   }
   catch (NumberFormatException nfe)
   {
     // If conversion failed, we assume this is a string.
     return "java.lang.String";
   }
 }
```

```
/**
 * Gets the column names.
 */
/**
 * Gets the column names.
 */
private List getColNames(CSVParser parser)
{
  try
  {
    if (mColNames == null)
    {
      // Get the first row. If the first row is NOT the column names, we need
      // to generate column names for them.

      if (!parser.nextLine())
      {
        // No data found.
        // ToDo: resource
        new Exception("No data");
      }

      mColNames = new java.util.ArrayList(10);

      String[] cols = parser.getLineValues();
      if (mIsFirstRowNames)
      {
        makeValidColumnNames(cols);
        for (int i = 0; i < cols.length; i++)
        {
          mColNames.add(cols[i]);
        }
      }
      else
      {
        for (int i = 0; i < cols.length; i++)
        {
          String colName =
            new SimpleStringBuffer(20).append("Column").append(i).toString();
          mColNames.add(colName);
        }
      }
    }

    return mColNames;
  }
  catch (Exception e)
  {
    throw new AdapterException(e);
  }
}
```

```
  /**
    * Make valid column names for all columns in CSV data source.
    *
    * This method applies the following rules to translate the given string
    * to a valid column name which can be accepted by EL:
    *
    * 1. If the first character of the string is digit,
    *    prefix the string with '_'.
    * 2. Translate any characters other than letter, digit, or '_'  to '_'.
    *
    *
    */
  private String[] makeValidColumnNames(String[] cols)
  {
    for (int i = 0; i <cols.length; i++)
    {
       // Trim out leading or ending white spaces
      if (cols[i] != null && cols[i].length() > 0)
      {
        cols[i] = cols[i].trim();
      }

      if (cols[i] == null || cols[i].length() == 0)
      {
         // Default as "column1", "column2", ... if column name null
        cols[i] = new SimpleStringBuffer("column").append(i+1).toString();
      }
      else
      {
        // Check special characters
        try
        {
          cols[i] = Utility.normalizeString(cols[i]);
        }
        catch (Exception e)
        {
           // On error, simply default to "columnX".
          cols[i] = new SimpleStringBuffer("column").append(i+1).toString();
        }
      }
    }
    return cols;
  }

}
```

20.11.3 CSVParser

Example 20–39 shows the contents of the `CSVParser` class.

Example 20–39 CSVParser

```
package oracle.adfinternal.model.adapter.sample;

import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.LineNumberReader;
import java.util.ArrayList;

import oracle.adf.model.utils.SimpleStringBuffer;

public final class CSVParser
{
  ////////////////////////////// Constants ////////////////////////////////////

  /** UTF8 encoding, used for hadling data in different languages. */
  public static final String UTF8_ENCODING = "UTF8";

  /** Quote character */
  private static char CHAR_QUOTE = '"';

  /** Comma (seperator) character */
  private static char CHAR_COMMA = ',';


  ////////////////////////////// Class Variables //////////////////////////////

  /**
   * CSV stream reader
   */
  private LineNumberReader mReader;

  /** Buffer to store one line of values.  */
  private ArrayList mValueArrayList = new ArrayList();

  /** Buffer to store one string value.  */
  private SimpleStringBuffer mValueBuffer = new SimpleStringBuffer(256);

  /** Current processed line.  */
  private String mLine = null;

  /** Current character position in the current line.  */
  private int mLinePosition = -1;

  /** Length of current line. */
  private int mLineLength = 0;

  /** If last character is comma. */
  private boolean mLastCharIsComma = false;

  /** Value separator character set. The separator can be one of these values.*/
  private char[] mSepCharSet = {CHAR_COMMA};

  /** Quote character. */
  private char mQuoteChar = CHAR_QUOTE;
```

```
////////////////////////////// Constructors ///////////////////////////////

/**
 * Constructor
 *
 * @param pInputStream CSV input stream
 * @throws Exception any error occurred
 */
public CSVParser(InputStream pInputStream) throws Exception
{
  // If no encoding is passed in, use "UTF-8" encoding
  this(pInputStream, UTF8_ENCODING);
}

/**
 * Constructor
 *
 * @param pInputStream CSV input stream
 * @param pEnc character encoding
 * @throws Exception any error occurred
 */
public CSVParser(InputStream pInputStream, String pEnc) throws Exception
{
  if (pInputStream == null)
  {
    throw new Exception("Null Input Stream."); //TODO: Resource
  }

  mReader = new LineNumberReader(new InputStreamReader(pInputStream, pEnc));
}

////////////////////////////// Public Methods ///////////////////////////////

/**
 * Sets the separator characters as a list of possible separators for the
 * data. CSV data may have more than one separators. By default this parser
 * considers comma (,) as the data separator.
 * @param seps  Array of separator charactors.
 */
public void setSeparators(char[] seps)
{
  if ((seps != null) && (seps.length > 0))
  {
    mSepCharSet = seps;
  }
}

/**
 * Sets the quote character.
 * @param ch Quote character.
 */
public void setQuoteChar(char ch)
{
  mQuoteChar = ch;
}
```

```
/**
 * Moves to the next line of the data.
 * @return returns false if the end of data reached.
 * @throws Exception any error occurred
 */
public boolean nextLine() throws Exception
{
  setLine(mReader.readLine());
  if (mLine == null)
  {
    // End of file
    mValueArrayList.clear();
    return false;
  }

  parseLine();

  return true;
}

/**
 * Gets values of next line.
 * @return next line elements from input stream. If end of data reached,
 *         it returns null.
 * @throws Exception any error occurred
 */
public String[] getLineValues() throws Exception
{
  if (mValueArrayList.size() > 0)
  {
    String[] ret = new String[mValueArrayList.size()];
    return (String[]) mValueArrayList.toArray(ret);
  }

  return null;
}



////////////////////////// Class Helpers ///////////////////////////////////

/**
 * Checks if the character is a valid separator.
 */
private boolean isSeparator(char ch)
{
  for (int i = 0; i < mSepCharSet.length; i++)
  {
    if (ch == mSepCharSet[i])
    {
      return true;
    }
  }

  return false;
}

/**
 * Tests if end of line has reached.
 * @return true if end of line.
```

```
 */
public boolean isEndOfLine()
{
  // If last char is comma, must return at least one more value
  return (mLinePosition >= mLineLength) && (!mLastCharIsComma);
}

/**
 * Sets current line to be processed
 *
 * @param line the line to be processed
 */
private void setLine(String line)
{
  mLine = line;

  if (line != null)
  {
    mLineLength = line.length();
    mLinePosition = 0;
  }

}

/**
 * If next character is quote character
 *
 * @return true if next character is quote
 */
private boolean isNextCharQuote()
{
  if ((mLinePosition + 1) >= mLineLength)
  {
    // no more char in the line
    return false;
  }
  else
  {
    char ch = mLine.charAt(mLinePosition + 1);
    if (ch == mQuoteChar)
    {
      return true;
    }
    else
    {
      return false;
    }
  }
}
```

```
/**
 * Parse one line.
 *
 * @return values of the line
 * @throws Exception any error occurred
 */
private void parseLine() throws Exception
{
  mValueArrayList.clear();

  String[] values = null;
  String value = null;

  while (!isEndOfLine())
  {
    value = getNextValue();
    mValueArrayList.add(value);
  }
}

/**
 * Gets next value from current line.
 * @return next data value.
 */
private String getNextValue() throws Exception
{
  mLastCharIsComma = false;

  // Clean up value buffer first
  if (mValueBuffer.length() > 0)
  {
    mValueBuffer.setLength(0);
  }

  boolean insideQuote = false;
  boolean firstChar = true;
  boolean endValue = false;

  // Scan char by char
  while ((mLinePosition < mLineLength) && !endValue)
  {
    boolean copyChar = true;
    char ch = mLine.charAt(mLinePosition);

    // If first char
    if (firstChar)
    {
      // Only check quote at first char
      if (ch == mQuoteChar)
      {
        insideQuote = true;
        copyChar = false;
      }
```

```
        // Also need to check comma at first char
        else if (isSeparator(ch))
        {
          copyChar = false;
          endValue = true;
          mLastCharIsComma = true;
        }

        firstChar = false;
      }
      // Not first char but inside quote
      else if (insideQuote)
      {
        // Check end quote
        if (ch == mQuoteChar)
        {
          copyChar = false;
          // Two sucesstive quote chars inside quote means quote char itself
          if (isNextCharQuote())
          {
            mLinePosition++;
          }
          // Otherwise it is ending quote
          else
          {
            insideQuote= false;
          }
        }
      }
      // Not first char and outside quote
      else
      {
        // Check comma
        if (isSeparator(ch))
        {
          copyChar = false;
          endValue = true;
          mLastCharIsComma = true;
        }
      }

      if (copyChar)
      {
        mValueBuffer.append(ch);
      }

      mLinePosition++;
    }

    if (mValueBuffer.length() > 0)
    {
      return mValueBuffer.toString();
    }
    else
    {
      return null;
    }
  }

}
```

21

Working with Web Services

This chapter contains advice for using web services with ADF projects, and general advice for creating and using web services in JDeveloper

This chapter includes the following sections:

- Section 21.1, "What are Web Services"
- Section 21.2, "Creating Web Service Data Controls"
- Section 21.3, "Securing Web Service Data Controls"

21.1 What are Web Services

Web services is the term for a technology that consists of a set of messaging protocols and programming standards that expose business functions over the Internet using open XML-based standards, and an individual web service is a discrete, reusable software component that is accessed programmatically over the Internet, using HTTP or sometimes SMTP, to return a response.

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

Oracle ADF has built in support to use web services as business service providers in applications. For example, an application could:

- Use some functionality in an application run by another company and exposed as a web service to provide business-to-business e-commerce.
- Use web service made available through a site such as Xmethods.com to provide some standard functionality.
- Find a web service that provides the specified functionality in a UDDI registry and use it at runtime.

You can use Oracle ADF to build applications that target one or all of the tiers in the J2EE platform using your choice of implementation technologies. Using ADF to implement your business services, you gain the additional flexibility to be able to expose parts of your application as web services at any time without code changes.

Factors influencing the decision to deploy a component as a web service are:

- Web services separate the application from the underlying architecture.
- Web services are lightweight, which can result in improved performance across the Internet or an intranet.
- Web services technology is designed to use the Web infrastructure, including HTTP.

It is useful to describe the XML standards on which web services are based.

21.1.1 SOAP

The Simple Object Access Protocol (SOAP) is a lightweight XML-based protocol that is used for the sending and receiving over messages of a transport protocol, usually HTTP or SMTP. The SOAP specification, which you can see at web site of the World Wide Web Consortium, provides a standard way to encode requests and responses. It describes the structure and data types of message payloads using XML Schema.

A SOAP message is constructed of the following components:

- A SOAP envelope that contains the SOAP body, the important part of the SOAP message, and optionally a SOAP header.
- A protocol binding that specifies how the SOAP envelope is sent, that in the case of web services generated in JDeveloper, is via HTTP.

Web services use SOAP, the XML protocol for expressing data as XML and transporting it across the Internet using HTTP, and SOAP allows for more than one way of converting data to XML and back again. JDeveloper supports SOAP RPC encoding, SOAP RPC-literal style, and document-literal style (also known as message style).

The web services you create in JDeveloper can be either for deployment on Oracle SOAP, which is based on Apache SOAP 2.2 and is part of the Oracle Application Server, or to the SOAP server, which is one of the OC4J containers in Oracle Application Server.

21.1.2 WSDL

The Web Services Description Language (WSDL) is an XML language used to describe the syntax of web service interfaces and their locations. You can see the WSDL v1.1 specification at the web site of the World Wide Web Consortium. Each web service has a WSDL document that contains all the information needed to use the service, the location of the service, its name, and information about the methods that the web service exposes. When you use one of JDeveloper's web service publishing wizards to produce your web service, the WSDL document for your service is automatically generated.

21.1.3 UDDI

Universal Description, Discovery and Integration (UDDI) provide a standards-based way of locating web services either by name, or by industry category. UDDI registries can be public, for example the public UDDI registries that are automatically available from JDeveloper, or private, such as a UDDI registry used within an organization. This version of JDeveloper only supports web service discovery using UDDI, however future versions will provide full support for UDDI registration. You can see the UDDI v2 specification at `http://www.uddi.org/`.

JDeveloper's UDDI browser, in the Connections Navigator, stores information about a UDDI registry and allows you to search a UDDI registry using search criteria that you specify to find web services that are described by WSDLs.

You can create your own registry connections to another public UDDI registry, or to a private UDDI registry within your organization. This creates a connection descriptor properties file that contains the enquiry endpoint and the business keys of the registry. You can find this file at `<JDEV_INSTALL>/system<release_and_build_number>/uddiconnections.xml`, where `<JDEV_INSTALL>` is the root directory in which JDeveloper is installed.

JDeveloper's Find Web Service wizard browses UDDI registries to find web services by either name or category. You must have an appropriate connection from your machine so that JDeveloper can make a connection to the UDDI registry you select, for example, a connection to the internet if you want to search a public UDDI registry, and you can only generate a stub to a web service that has a tick in the Is WSDL? column that identifies the registry entry as being defined by a WSDL document.

When you use UDDI registries a term you will come across, and that you may be unfamiliar with, is tModel, short for Technical Model. This represents the technical specification of a web service, and when you search for a web service using the Find Web Service wizard, the wizard also displays other web services that are compatible with the same tModel.

The data structure types used in UDDI are:

- **Service Details** This section gives information about the service, including the name.
- **Business Entity** This is the top-level data structure called businessEntity that contains information about the business providing the web service.
- **Service Bindings** contains the bindingTemplate, that contains information about the service access point, and the tModel that gives the technical specification of the web service.

When the Find Web Services wizard finds a web service, it lists all web services that are compatible with the same tModel.

21.1.4 Web Services Interoperability

A key issue facing web services is how interoperable web services actually are. The key feature of web services is that they use common standards to avoid the problems that earlier solutions to getting different applications to be able to use each other's components, for example CORBA, had. However the standards themselves have been being written at the same time as the organizations have been starting to write, deploy and use web services. This has led to interoperability issues such as web services being written using different standards, for example, not using WSDL to provide web service information.

The Web Services-Interoperability Organization (WS-I) was formed by Oracle and other industry leaders to address these issues of interoperability, and to provide tools so that web services can be tested to see how well they interoperate. JDeveloper helps you to test the interoperability of web services by analyzing a web service for conformity to the WS-I Basic Profile 1.0. First you have to download a WS-I compliant analyzer. There are a number of these available from independent vendors, and one from the WS-I web site. A set of test assertions is used to find out how well a web service conforms to the basic profile, and information is recorded for the following artifacts:

- **Discovery** when a web service has been found using a UDDI registry. If the service has not been found using the Find Web Services wizard, this section of the report returns errors in the Missing Input section.
- **Description** of a web service's WSDL document, where the different elements of the document are examined and non-conformities are reported. An example of a failure in this section is a failure of assertion WSI2703, that gives the message "`WSDL definition does not conform to the schema located at http://schemas.xmlsoap.org/wsdl/soap/2003-02-11.xsd for some element using the WSDL-SOAP binding namespace, or does not conform to the schema located at http://schemas.xmlsoap.org/wsdl/2003-02-11.xsd for some element using the WSDL namespace`."
- Message that tests the request and response messages when the connection is made to the web service and it sends its reply.

For more information about WS-I including the specification, see the web site of The Web Services-Interoperability Organization (WS-I) at ws-i.org.

21.2 Creating Web Service Data Controls

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

Also, you can re-use components created by Oracle ADF to make them available as web services for other applications to access.

21.2.1 How to Create a Web Service Data Control

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

Note:

If you are working behind a firewall and you want to use a web service that is outside the firewall, you must configure the Web/Browser Proxy settings in JDeveloper. Refer to the JDeveloper online help for more information.

To create a web service data control:

1. In the Application Navigator, right-click an application and choose **New**.
2. In the New Gallery, expand Business Tier in the Categories tree, and select **Web Services**.
3. In the Items list, double-click **Web Service Data Control**.
4. Follow the wizard instructions to complete creating the data control.

Alternatively, you can right-click on the WSDL node in the navigator and choose **Create Data Control** from the context menu.

21.3 Securing Web Service Data Controls

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

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

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

21.3.1 WS-Security Specification

The WS-Security specification unifies multiple security technologies to make secure web services interoperable between systems and platforms. The specification can be viewed at http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf.

WS-Security addresses the following aspects of web services security issues:

- Authentication and Authorization

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

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

- Data Authenticity, Integrity and Non-Repudation

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

- Data Privacy

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

- Denial of Service Attacks

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

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

- a web service deployed on OC4J for testing purposes.
- web service running on Oracle Application Server.
- A web service running anywhere in the world that is accessible through the Internet

21.3.2 Creating and Using Keystores

An ADF 10.1.3 Web Services data control can be configured for message level security using either Java Key Store (JKS), or the Oracle Wallet. For information on setting up and using Oracle Wallet, see the Oracle Technology Network at www.oracle.com/technology.

This section describes:

- Creating a keystore using the J2SE 1.4 Keytool utility
- Building a keystore private/public key pairs, which are used for encryption and signing.
- How to obtain a Certificate to issue digital signatures from a root certificating authority.
- How to import the Certificate into the keystore.
- How to export the Certificate with the public key for encryption.

This is illustrated by creating two keystores, one to be configured on the server side, and the other on the client side (the data control side).

Note: The steps outlined in this section for requesting digital certificates is for test purposes only. Deployments intending to use Web Services data control with digital signatures enabled must ensure that trusted certificates are generated compliant to the security policies of the deployment environment.

21.3.2.1 How to Create a Keystore

To create a public private key pair that can be used by the client for encryption and signing, at the command prompt run the following:

Example 21–1 Command to Create a Keystore

```
keytool -genkey -alias clientenckey clientsignkey -keyalg RSA -sigalg SHA1withRSA
-keystore client.jks
```

The keystore utility will prompt you for the keystore password, and then asks questions to determine the distinguished name (DN), which is a unique identifier and consists of the following components:

- CN= common name. This must be a single name without spaces or special characters.
- OU=organizational unit
- O=organization name

- L=locality name
- S=state name
- C=country, a two letter country code

After you answer the questions, the Keytool utility will prompt you for the key password. If the key password is the same as the keystore password, press Enter without entering a value. Otherwise, enter the key password. After you enter the key password, the keystore file client.jks is created in the current directory. It contains a single key pair with the alias clientenckey which can be used to encrypt the SOAP requests from the data control.

Next, create a key pair for digitally signing the SOAP requests made by the data control. At the command prompt run the command again, but use clientsignkey for the alias of the signing key pair.

To list the key entries in the keystore, run the following:

Example 21–2 Command to List Key Pairs in the Keystore

```
keytool -list -keystore client.jks
```

The Keytool utility will prompt you for the store password. Enter the password that was used to create the keystore. Repeat the commands to create a keystore for the server side, and use serverenckey for the encryption key pair, and serversignkey for the signing key pair.

21.3.2.2 How to Request a Certificate

The keytool, by default, generates a self-signed certificate, that is a certificate whose issuer is the same as the generator of the key.

If your public key is to be distributed to the outside world, to allow verification of the digital signatures you have issued, then a trusted Certificate Authority (CA) must issue a certificate vouching your identity on your public key. To do this, create a Certificate request file for the signature key pair you have created and submit the request file to a CA.

At the command prompt, run the following:

Example 21–3 Command to Create a Certificate Request File

```
keytool -certreq -file clientsign.csr -alias clientsignkey -keystore client.jks
```

The Keytool utility will prompt you for the store and key passwords. After you enter the passwords, a certificate request is generated in a file called clientsign.csr for the public key aliased by clientsignkey.

When you are developing your application, you can use a CA such as Verisign to request trial certificates. Go to www.verisign.com, navigate to Free SSL Trial Certificate and create a request. You must enter the same DN information you used when you created the keystore. Verisign's certificate generation tool will ask you to paste the contents of the certificate request file generated by the keytool (in this case, clientsign.csr). Once all the information is correctly provided, the certificate will be sent to the email ID you have provided, and you have to import it into the keystore.

Open the contents of the certificate in a text editor, and save the file as `clientsign.cer`.

You also have to import the root certificate issued by Verisign into the keystore. The root certificate is needed to complete the certificate chain up to the issuer.

The root certificate vouches the identity of the issuer. Follow the instructions in the email you received from Verisign to access the root certificate, and paste the contents of the root certificate into a text file called `root.cer`.

Once you have the root.cer and clientsign.cer files created, run the following command to import the certificates into your keystore:

Example 21–4 Importing the Root Certificate

```
keytool -import -file root.cer -keystore client.jks
```

The Keytool utility will prompt you for the store password. Next you must import your public key certificate.

Import your public key certificate next.

Example 21–5 Importing the Public Key Certificate

```
keytool -import -file clientsign.cer -alias clientsignkey -keystore client.jks
```

The Keytool utility will prompt you for the store and key password. After entering the passwords, execute the same commands to set up the trusted certificate chain in the server keystore.

Perform the same commands steps to set up the trusted certificate chain in the server keystore.

Once the certificate chains are set up, the client and sever are ready to issue digitally signed SOAP requests.

Note:

Trusted certificates are mandatory when issuing digital signatures on the SOAP message. You cannot issue digital signatures with self-signed/untrusted certificates in your keystore.

21.3.2.3 How to Export a Public Key Certificate

The server must export its public key to the client so the client can encrypt the data it sends to the server. The server can then use its corresponding private key to decrypt the data. The server's public key certificate is imported into the client keystore.

At the command prompt, run the following:

Example 21–6 Command to Export the Server's Public Key Certificate

```
keytool -export -file serverencpublic.cer -alias serverenckey -keystore server.jks
```

The Keytool utility will prompt you for the store password.

In this example, `clientencpublic.cer` contains the public key certificate of the client's encryption key. To import this certificate in the server's keystore, run the following:

Example 21–7 Command to Import Client's Encription Key

```
keytool -import -file serverencpublic.cer -alias serverencpublic -keystore
client.jks
```

The Keystore utility will prompt you for the store password.

Similarly, the client must export its public key so that it can be imported into the server's keystore.

Example 21–8 Command to Export the Client's Public Key Certificate

```
keytool -export -file clientencpublic.cer -alias clientenckey -keystore client.jks
```

The Keytool utility will prompt you for the store password.

Example 21–9 Command to Import the Public Key Certificate

```
keytool -import -file clientencpublic.cer -alias clientencpublic -keystore
server.jks
```

The server and client keystores are now ready to be used to configure security for the web service data control.

The Keytool utility will prompt you for the keystore password.

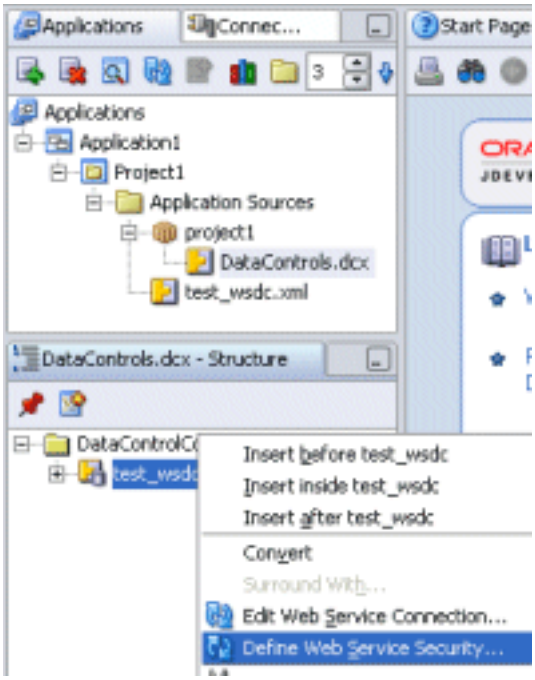
21.3.3 Defining Web Service Data Control Security

Once you have a web services data control in a JDeveloper project, you can define security using the Data Control Security wizard.

To invoke the data control security wizard:

1. Select the web service data control in the Application Navigator.
2. In the Structure window, right-click the web service data control, and choose **Define Web Service Security**.
3. Consult the following sections for more information, or click F1 or Help in the wizard for detailed information about a page of the wizard.

Figure 21–1 Invoking the Data Control Security Wizard



21.3.3.1 How to Set Authentication

WS-Security allows for service level authentication by using either username tokens or binary tokens. In addition to these, the web service client can issue SAML assertion tokens that can be used for server side authentication, or for participation in a federated SSO environment.

Figure 21–2 Select the Type of Authentication

21.3.3.1.1 Testing Authenticated Web Service Data Controls on OC4J

Oracle's WS-Security implementation is integrated with JAZN (JAAS) to achieve the authentication. How authentication using a certificate is done depends on the implementation and integration with the platform security system. This section discusses configuring OC4J as the server where the application is deployed.

> **Note:** When the application is deployed to Oracle Application Server, the administrator should use the security editing tool to add users to the security system, grouping them in the appropriate role and granting appropriate privileges. This example of manually editing `system-jazn-data.xml` is just for testing, and not recommended for working applications.

For Username Token authentication, username/password pair must be a trusted user entry in the JAZN repository.

For X509 Token authentication, the CN (Common Name) on whom the Certificate is issued must be a trusted user in the JAZN repository.

For SAML authentication, the user must be a valid user in the JAZN repository.

To edit the JAZN repository:

- Open <JDEV_INSTALL>/J2EE/home/system-jazn-data.xml and enter the authentication details. For example, for X509 authentication, make an entry under the <users> section similar to:

```
<user>
  <name>King</name>
  <display-name>OC4J Administrator</display-name>
  <description>OC4J Administrator</description>
  <credentials>{903}/LptVQLDeA5sgZFLL5TKlr/qjVFPxB42</credentials>
</user>
```

21.3.3.1.2 Username Tokens

Username tokens provide basic authentication of a username/password pair. The passwords can be transmitted as plain text or digest.

> **Note:** This is not the same as HTTP basic or digest authentication. The concept is similar, but it differs in that the recipient of HTTP authentication is the HTTP server, whereas for the web service data control, the username tokens are passed with the message, and the recipient is the target web service.

Oracle's WS-Security implementation is integrated with JAZN (JAAS) to achieve the authentication. The username/password pair must be a trusted user entry in the JAZN repository.

To use username tokens for authentication:

1. In the Authentication page of the wizard, under Available Operations, select one or more ports or operations to apply the authentication to.
2. Select the authentication type as the Username Token.
3. Enter the remaining information required for username authentication.

21.3.3.1.3 X509 Certificate Authentication

An X509 certificate issued by a trusted CA is a binary security token which can be used to authenticate the client. The client sends its X509 certificate with a digital signature, which is used by the server for authentication. The X509 certificate chain associated with signature key is used for authentication.

You must have the keystore file, with the root certificate of the CA, installed on the server.

> **Note:** An X509 certificate can only be configured at port level, unlike the other authentication types that can be configured at port or operation level.

To use X509 certificate authentication:

1. In the Authentication page of the wizard, select the authentication type as the X509 Token.
2. In the Keystore page of the wizard, and specify the location of the keystore file, and enter the signature key alias and password.

21.3.3.1.4 SAML Assertion Tokens

SAML assertion tokens can be used to allow client to authenticate against the web service, or allow the client to participate in a federated SSO environment, where

authenticated details can be shared between domains in a vendor independent manner.

> **Note:** SAML Assertions will not be issued if the user identity cannot be established by JAZN.

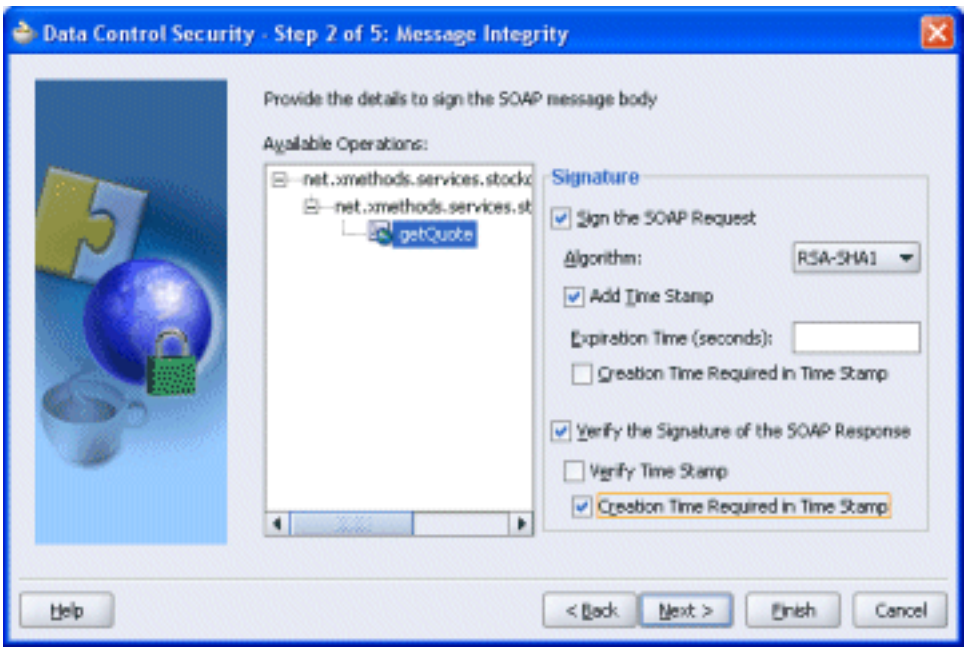
To use SAML authentication:

1. In the Authentication page of the wizard, select the authentication type as the SAML Token.
2. The Subject Name is the username name against which the SAML Assertions will be issued.
3. You can choose Confirmation method as SENDER-VOUCHES or SENDER-VOUCHES-UNSIGNED:
   - ISENDER-VOUCHES (default). The SAML tokens must be digitally signed. This is the preferred method to issue SAML tokens. If you choose this confirmation technique, then you must configure a keystore and enter keystore and signature key information on the Keystore page of the wizard.
   - SENDER-VOUCHES-UNSIGNED. The SAML tokens are transmitted without any digital signatures. If you choose this confirmation technique, then you need not configure a keystore and signature key.

21.3.3.2 How to Set Digital Signatures

You can configure digital signatures on the outgoing SOAP messages, and verify digital signatures on the incoming message from the web service your application is contacting. You can also enforce an expiration window for the digital signatures.

Figure 21–3 Set a Digital Signature



You can set a digital signature on the outgoing SOAP message at port or operation level in the Message Integrity page of the wizard, and verify the digital signatures from the incoming message of the web service.

To sign the SOAP request, and verify the signature of the SOAP response:

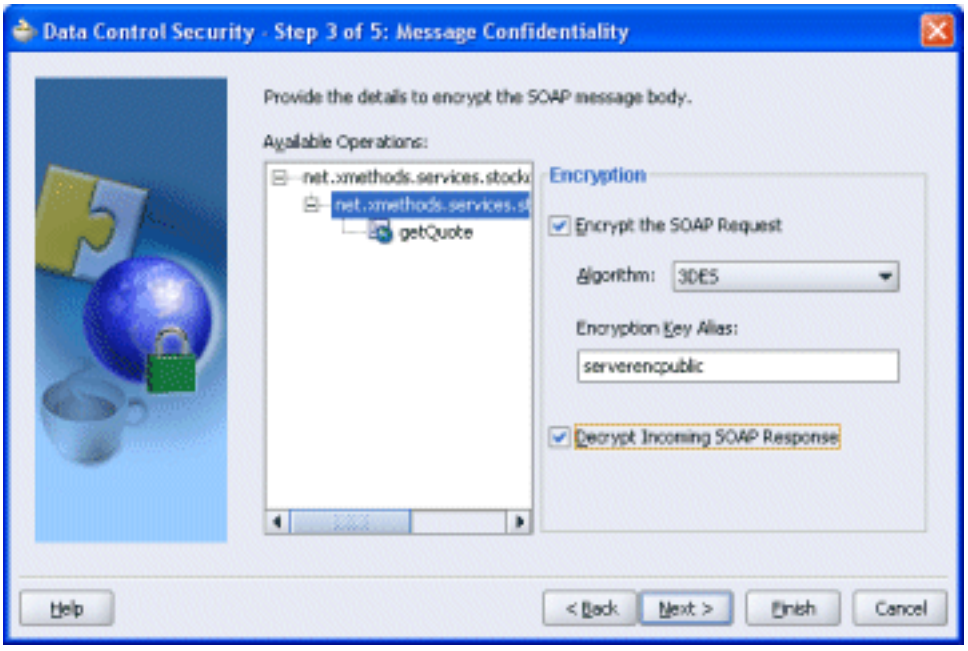
1. In the Message Integrity page of the wizard, select the appropriate options.
2. In the Keystore page of the wizard, and specify the location of the keystore file, and enter the signature key alias and password.

21.3.3.3 How to Set Encryption and Decryption

When you create a web service in JDeveloper, you can set security options in the Web Services Editor. These are then applied at the server side once the web service is deployed. Refer to the JDeveloper online help for complete information.

Before deploying the web service, run the editor and configure encryption and decryption details on the web service. Ensure that you have specified the client's (that is, the data control's) public key to be used for encryption.

Figure 21–4 Set Encryption and Decryption



You can encrypt and outgoing SOAP message at port or operation level in the Message Confidentiality page of the wizard, and decrypt the incoming message from the web service.

To encrypt the SOAP request, and decrypt the SOAP response:

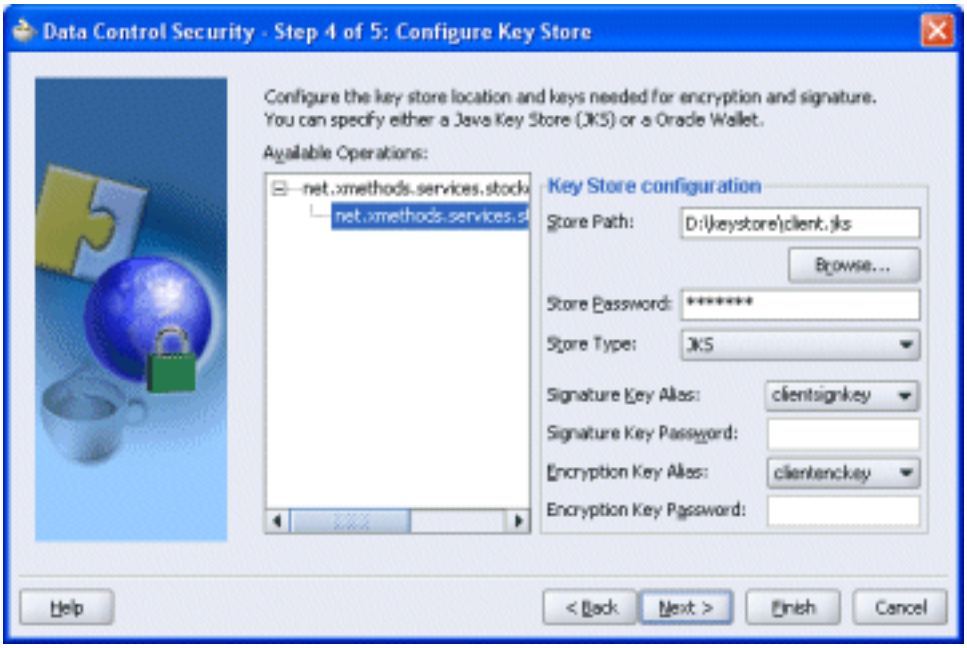
1. In the Message Confidentiality page of the wizard, select the appropriate options. The encryption algorithm you select must be the same as that configured on the server side when the web service was deployed.
2. Enter the server's public key alias to allow the data control to encrypt the key details using the server's public key. In this example, serverencpublic is the server's public key certificate that imported in the key store configuration.

3. If the web service uses incoming message encryption, select Decrypt Incoming SOAP Response.
4. In the Keystore page of the wizard, and specify the location of the keystore file, and enter the encryption key alias and password.

21.3.3.4 How to Use a Key Store

Section 21.3.2.1, "How to Create a Keystore" described setting up keystores for the client (the web service data control) and for the server (a deployed web service). In the Configure Key Store page of the Data Control Security wizard you enter the information needed for the keystore to be used for data control security.

Figure 21–5 Set Key Store Information



The final stage of configuring WS-Security for a data control based on a web service is to specify the keystore details. Enter the information to access the client keystore here, and when the wizard is finished the keys configured in the store will be available for signatures and encryption for all requests generated by the data control and all responses processed by the data control.

To set key store access information:

- In the Configure Key Store page of the wizard, enter the appropriate values.

22

Deploying ADF Applications

This chapter describes how to deploy applications that use ADF to Oracle Application Server as well as to third-party application servers such as JBoss, WebLogic, and WebSphere.

This chapter includes the following sections:

- Section 22.1, "Introduction to Deploying ADF Applications"
- Section 22.2, "Deployment Steps"
- Section 22.3, "Deployment Techniques"
- Section 22.4, "Deploying Applications Using Ant"
- Section 22.5, "Deploying the SRDemo Application"
- Section 22.6, "Deploying to Oracle Application Server"
- Section 22.7, "Deploying to JBoss"
- Section 22.8, "Deploying to WebLogic"
- Section 22.9, "Deploying to WebSphere"
- Section 22.10, "Deploying to Tomcat"
- Section 22.11, "Deploying to Application Servers That Support JDK 1.4"
- Section 22.12, "Installing ADF Runtime Library on Third-Party Application Servers"
- Section 22.13, "Verifying Deployment and Troubleshooting"

22.1 Introduction to Deploying ADF Applications

Deployment is the process through which application files are packaged as an archive file and transferred to the target application server. Deploying ADF applications is only slightly different from deploying standard J2EE applications.

JDeveloper supports the following deployment options:

- Deploying to an application server.
- Deploying to an archive file: Applications can be deployed indirectly by choosing an archive file as the deployment target. You can then use tools provided by the application server vendor to deploy the archive file. Information on deploying to selected other application servers is available on the Oracle Technology Network (`http://www.oracle.com/technology`).
- Deploying for testing: JDeveloper supports two options for testing applications:

Embedded OC4J Server: You can test applications, without deploying them, by running them on JDeveloper's embedded Oracle Containers for J2EE (OC4J) server. OC4J is the J2EE component of Oracle Application Server.

Standalone OC4J: In a development environment, you can deploy and run applications on a standalone version of OC4J prior to deploying them to Oracle Application Server. Standalone OC4J is included with JDeveloper.

Connection to Data Source

You need to configure in JDeveloper a data source that refers to the data source (such as a database) used in your application.

ADF Runtime Library

If you are deploying to third-party application servers (such as JBoss, WebLogic, and WebSphere), you have to install the ADF runtime library on the servers. See Section 22.12, "Installing ADF Runtime Library on Third-Party Application Servers" for details.

For Oracle Application Server, the ADF runtime libraries are already installed.

Standard Packaging

After you have all the necessary files, you package the files for the application for deployment in the standard manner. This gives you an EAR file, a WAR file, or a JAR file.

When you are ready to deploy your application, you can deploy using a variety of tools. You can deploy to most application servers from JDeveloper. You can also use tools provided by the application server vendor. Tools are described in the specific application server sections later in the chapter.

Incompatibilities

When deploying applications to application servers, make sure that features used in the applications are supported by the target application servers. For example, when deploying applications that use EJB 3.0, which is in "early draft review" status at the time this book is written, you need to check that the target application server supports the EJB 3.0 features used in the applications.

22.2 Deployment Steps

To deploy an application, you perform these steps:

Step 1: Install the ADF Runtime Library on the Target Application Server

Step 2: Create a Connection to the Target Application Server

Step 3: Create a Deployment Profile for the JDeveloper Project

Step 4: Create Deployment Descriptors

Step 5: Perform Additional Configuration Tasks Needed for ADF

Step 6: Perform Application Server-Specific Configuration

Step 7: Deploy the Application

Step 1 Install the ADF Runtime Library on the Target Application Server

This step is required if you are deploying ADF applications to third-party application servers, and optional if you are deploying on Oracle Application Server or standalone

OC4J. See Section 22.12, "Installing ADF Runtime Library on Third-Party Application Servers" for installation steps.

JSF applications that contain ADF Faces components have a few additional deployment requirements:

- ADF Faces require Sun's JSF Reference Implementation 1.1_01 (or later) and MyFaces 1.0.8 (or later).
- ADF Faces applications cannot run on an application server that only supports JSF 1.0.

Step 2 Create a Connection to the Target Application Server

In JDeveloper, create a connection to the application server where you want to deploy your application. Note that if your target application server is WebSphere, you can skip this step because JDeveloper cannot create a connection to WebSphere. For WebSphere, you deploy applications using the WebSphere console. See Section 22.9, "Deploying to WebSphere" for details.

To create a connection to an application server:

1. In the Connections Navigator, right click **Application Server** and choose **New Application Server Connection**. The Create Application Server Connection wizard opens.
2. Click **Next** to proceed to the Type page.
3. On the Type page:
   - Provide a name for the connection.
   - In the **Connection Type** list box, select the application server type. You can deploy ADF applications on these application servers:
     - Standalone OC4J 10.1.3
     - Oracle Application Server (10.1.2 or 10.1.3)
     - WebLogic Server (8.x or 9.x)
     - JBoss 4.0.x
     - Tomcat 5.x
   - Click **Next**.
4. If you selected Tomcat as the application server, the Tomcat Directory page appears. Enter the Tomcat's "webapps" directory as requested and click **Next**. This is the last screen for configuring a Tomcat server.
5. If you selected JBoss as the application server, the JBoss Directory page appears. Enter the JBoss's "deploy" directory as requested and click **Next**. This is the last screen for configuring a JBoss server.
6. On the Authentication page enter a user name and password that corresponds to the administrative user for the application server. Click **Next**.
7. On the Connection page, identify the server instance and configure the connection. Click **Next**.
8. On the Test page, test the connection. If not successful, return to the previous pages of the wizard to fix the configuration.

   If you are using WebLogic, you may see this error when testing the connection:

```
Class Not Found Exception -
weblogic.jndi.WLInitialContextFactory
```

This exception occurs when `weblogic.jar` is not in JDeveloper's classpath. You may ignore this exception and continue with the deployment.

9. Click **Finish**.

Step 3 Create a Deployment Profile for the JDeveloper Project

Deployment profiles are project components that govern the deployment of a project or application. A deployment profile specifies the format and contents of the archive file that will be created.

To create a deployment profile:

1. In the Applications Navigator, select the project for which you want to create a profile.
2. Choose **File > New** to open the New Gallery.
3. In the Categories tree, expand **General** and select **Deployment Profiles**.
4. In the **Items** list, select a profile type. For ADF applications, you should select one of the following from the **Items** list:
   - WAR File
   - EAR File
   - EJB JAR File

   If the desired item is not found or enabled, make sure you selected the correct project, and select **All Technologies** in the **Filter By** dropdown list.

   Click **OK**.
5. In the Create Deployment Profile dialog provide a name and location for the deployment profile, and click **OK**.

   The profile, <name>.deploy, will be added to the project, and its Deployment Profile Properties dialog will open.
6. Select 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.

   Typically you can accept the default settings. One of the settings that you might want to change is the J2EE context root (select **General** on the left pane). By default, this is set to the project name. You need to change this if you want users to use a different name to access the application. Note that if you are using custom JAAS LoginModules for authentication with JAZN, the context root name also defines the application name that is used to look up the JAAS LoginModule.
7. Click **OK** to close the dialog.
8. Save the file to keep all changes.

To view or edit a deployment profile, right-click it in the Navigator, and choose **Properties**, or double-click the profile in the Navigator. This opens the Deployment Profile Properties dialog.

Step 4 Create Deployment Descriptors

Deployment descriptors are server configuration files used to define the configuration of an application for deployment and are deployed with the J2EE application as needed. The deployment descriptors 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 that you can use to view and set properties.

In addition to the standard J2EE deployment descriptors (for example: `application.xml`, ejb-jar.xml, and `web.xml`), you can also have deployment descriptors that are specific to your target application server. For example, if you are deploying on Oracle Application Server, you can also have `orion-application.xml`, `orion-web.xml`, and `orion-ejb-jar.xml`.

To create a deployment descriptor:

1. In the Applications Navigator, select the project for which you want to create a descriptor.
2. Choose **File > New** to open the New Gallery.
3. In the Categories tree, expand **General** and select **Deployment Descriptors**.
4. In the **Items** list, select a descriptor type, and click **OK**.

   If the desired item is not found, make sure you selected the correct project, and select **All Technologies** in the **Filter By** dropdown list. If the desired item is not enabled, check to make sure the project does not already have a descriptor of that type. A project may have only one instance of a descriptor.

   JDeveloper starts the Create Deployment Descriptor wizard or opens the file in the editor pane, depending on the type of deployment descriptor you selected.

> **Note:** For EAR files, do not create more than one deployment descriptor per application or workspace. These files are assigned to projects, but have workspace scope. If multiple projects in an application or workspace have the same deployment descriptor, the one belonging to the launched project will supersede the others. This restriction applies to `application.xml`, `data-sources.xml`, `jazn-data.xml`, and `orion-application.xml`.

To view or change deployment descriptor properties:

1. In the Applications Navigator, right-click the deployment descriptor and choose **Properties**. If the context menu does not have a **Properties** item, then the descriptor must be edited as a source file. Choose **Open** from the context menu to open the profile in an XML editor window.
2. Select items in the left pane to open dialog pages in the right pane. Configure the descriptor by setting property values in the pages of the dialog.
3. Click **OK** when you are done.

To edit a deployment descriptor as an XML file:

- In the Applications Navigator, right-click the deployment descriptor and choose **Open**. The file opens in an XML editor.

Step 5 Perform Additional Configuration Tasks Needed for ADF

If your application uses ADF Faces components, ensure that the standard J2EE deployment descriptors contain entries for ADF Faces, and that you include the ADF and JSF configuration files in your archive file (typically a WAR file). When you create ADF Faces components in your application, JDeveloper automatically creates and configures the files for you.

Check that the WAR file includes the following configuration and library files:

- `web.xml`—See Section 4.4.2.1, "More About the web.xml File" for ADF and JSF entries in this file.
- `faces-config.xml` and `adf-faces-config.xml` files. See Section 4.4.2.2, "More About the faces-config.xml File" and Section 4.4.2.3, "Starter adf-faces-config.xml File" for details.
- JAR files used by JSF and ADF Faces:
  - `commons-beanutils.jar`
  - `commons-collections.jar`
  - `commons-digester.jar`
  - `commons-logging.jar`
  - `jsf-api.jar` and `jsf-impl.jar`—These JAR files are the JSF reference implementation that JDeveloper includes by default.

    Note: If you are using another JSF implementation (such as MyFaces), you must include the JAR files for those libraries when you create the deployment profile and remove the JSF JAR files (`jsf-api.jar` and `jsf-impl.jar`) from the WAR file; otherwise, your application will not run correctly.

  - `jstl.jar` and `standard.jar`—These are the libraries for the JavaServer Pages Standard Tag Library (JSTL).
  - `adf-faces-api.jar`—Located in the ADF Faces runtime library, this JAR contains all public ADF Faces APIs and is included in the WAR by default.
  - `adf-faces-impl.jar`—Located in the ADF Faces runtime library, this JAR contains all private ADF Faces APIs and is included in the WAR by default.
  - `adfshare.jar`—Located in the ADF Common runtime library, this JAR contains ADF Faces logging utilities.

    If you have installed the ADF runtime libraries, `adfshare.jar` is included in the WAR by default. Otherwise, you must manually include `adfshare.jar` in `WEB-INF/lib` when creating the WAR deployment profile.

If you are using ADF databound UI components as described in Section 5.2, "Using the Data Control Palette", check that you have the `DataBindings.cpx` file. For information about the file, see Section 5.3, "Working with the DataBindings.cpx File".

A typical WAR directory structure for a JSF application has the following layout:

```
MyApplication/
  JSF pages
  WEB-INF/
    configuration files (web.xml, faces-config.xml etc)
    tag library descriptors (optional)
    classes/
      application class files
      Properties files
    lib/
      commons-beanutils.jar
      commons-collections.jar
      commons-digester.jar
      commons-logging.jar
```

```
jsf-api.jar
jsf-impl.jar
jstl.jar
standard.jar
```

Step 6 Perform Application Server-Specific Configuration

Before you can deploy the application to your target application server, you may need to perform some vendor-specific configuration. See the specific application server sections later in this chapter.

Step 7 Deploy the Application

> **Note:** If you are running WebLogic 8.1, see Section 22.8.3, "WebLogic 8.1 Deployment Notes".

To deploy to the target application server from JDeveloper:

- Right-click the deployment profile, choose **Deploy to** from the context menu, then select the application server connection that you created earlier (in step 2 on page 22-3).

You can also use the deployment profile to create the archive file (EAR, WAR, or JAR file) only. You can then deploy the archive file using tools provided by the target application server. To create an archive file:

- Right-click the deployment profile and choose **Deploy to WAR file** (or **Deploy to EAR file**) from the context menu.

Step 8 Test the Application

Once you've deployed the application, you can test it from the application server. To test run your application, open a browser window and enter a URL of the following type:

- For Oracle Application Server: http://<host>:port/<context root>/<page>
- For Faces pages: http://<host>:port/<context root>/faces/<page>

> **Note:** The reason why `/faces` has to be in the URL for Faces pages is because JDeveloper configures your web.xml file to use the URL pattern of /faces to be associated with the Faces Servlet. The Faces Servlet does its per-request processing, strips out the /faces part in the URL, then forwards 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) and so your JSP is run without the necessary JSF per-request processing.

22.3 Deployment Techniques

Table 22–1 describes some common deployment techniques that you can use during the application development and deployment cycle. The table lists the deployment techniques 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 using scripting tools.

Table 22–1 Deployment Techniques

Deployment Technique	When to Use
Deploy directly from JDeveloper	This technique is typically used when you are developing your application. When you are developing the application, you may want to deploy it quickly for testing. You want deployment to be quick because you will be repeating the editing and deploying process many times. JDeveloper comes with an embedded OC4J server, on which you can run and test your application. You should also deploy your application to an external application server to test it.
Deploy to EAR file, then use the target application server's tools for deployment	This technique is typically used 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 OracleAS Single Sign-On) that are not available on the development server. You can also use the test environment to develop your deployment scripts. The scripts may involve Ant.
Use a script to deploy applications	This technique is typically used on test and production environments. On production environments, system administrators usually run scripts to deploy applications.

22.4 Deploying Applications Using Ant

You can also use Ant to package and deploy applications. The `build.xml` file, which contains the deployment commands for Ant, may vary depending on the target application server.

For deployment to Oracle Application Server using Ant, see the chapter "Deploying with the OC4J Ant Tasks" in the *Oracle Containers for J2EE Deployment Guide*. This chapter provides complete details on how to use Ant to deploy to Oracle Application Server. Oracle provides Ant tasks that are specific to Oracle Application Server.

For deployment to other application servers, see the application server's documentation. If your application server does not provide specific Ant tasks, you may be able to use generic Ant tasks. For example, the generic `ear` task creates an EAR file for you.

For information about Ant, see `http://ant.apache.org`.

22.5 Deploying the SRDemo Application

The SRDemo application includes a project called BuildAndDeploy, which contains EAR and WAR deployment profiles as well as Ant scripts that you can use to build the application. The deployment profiles pull in the appropriate files from the projects in the application workspace to build the EAR and WAR files. You can deploy the EAR or WAR file on your target application server. (You can also deploy directly to your application server from JDeveloper if you have created a connection to your application server.)

To view the properties of a deployment profile, right-click the deployment profile and choose **Properties** from the context menu.

The SRDemo application also includes the `UserInterface/src/META-INF/SRDemo-jazn-data.xml` file. The file contains some usernames and passwords so that the application can work out of the box running on the embedded OC4J server. Note that this file is not distributed in the EAR

file. If you deploy the application to an external application server, you have to set up the relevant credential store on the target application server.

If you want to deploy the application to different application servers, you can create a separate deployment profile for each target application server. This enables you to configure the properties for each target separately.

> **Note:** The SRDemo sample application uses EJB 3.0 features. As a result, it may not run on all application servers. Currently, it has been tested against Oracle Application Server 10.1.3 and OC4J standalone 10.1.3.

22.6 Deploying to Oracle Application Server

This section describes deployment details specific to Oracle Application Server:

- Section 22.6.1, "Oracle Application Server Versions Supported"
- Section 22.6.2, "Oracle Application Server Release 2 (10.1.2) Deployment Notes"
- Section 22.6.3, "Oracle Application Server Deployment Methods"
- Section 22.6.4, "Oracle Application Server Deployment to Test Environments ("Automatic Deployment")"
- Section 22.6.5, "Oracle Application Server Deployment to Clustered Topologies"

22.6.1 Oracle Application Server Versions Supported

Table 22–2 shows the supported versions of Oracle Application Server:

Table 22–2 Support Matrix for Oracle Application Server

Oracle Application Server Version	JDK Version	J2EE Version
Release 3 (10.1.3)	1.5_05	1.4
Release 2 (10.1.2)	1.4	1.3

22.6.2 Oracle Application Server Release 2 (10.1.2) Deployment Notes

If you are deploying to Oracle Application Server Release 2 (10.1.2), you have to perform some additional steps before you can run your ADF applications:

- This version of Oracle Application Server supports JDK 1.4. This means that you need to configure JDeveloper to build your applications with JDK 1.4 instead of JDK 1.5. See Section 22.11, "Deploying to Application Servers That Support JDK 1.4" for details.
- You need to install the ADF runtime libraries on the application server. This is because the ADF runtime libraries that were shipped with Release 2 (10.1.2) need to be updated. To install the ADF runtime libraries, see Section 22.12.1, "Installing the ADF Runtime Libraries from JDeveloper".
- Note that Oracle Application Server Release 2 (10.1.2) supports J2EE 1.3, while JDeveloper 10.1.3 supports J2EE 1.4. This means that if you are using J2EE 1.3 components (such as EJB 2.0), you have to ensure that JDeveloper creates the appropriate configuration files for that version. Configuration files for J2EE 1.3 and 1.4 are different.

Table 22–3 lists the configuration files that need to be J2EE 1.3-compliant, and how to configure JDeveloper to generate the appropriate version of the files.

Table 22–3 Configuring JDeveloper to Generate Configuration Files That Are J2EE 1.3-Compliant

Configuration File	How to Configure JDeveloper to Generate Appropriate Version of the File
`application.xml` `web.xml`	1. Select the project in the Applications Navigator. 2. Select **File > New** to display the New Gallery. 3. In **Categories**, expand **General** and select **Deployment Descriptors**. 4. In **Items**, select **J2EE Deployment Descriptor Wizard** and click **OK**. 5. Click **Next** in the wizard to display the Select Descriptor page. 6. On the Select Descriptor page, select **application.xml** (or **web.xml**) and click **Next**. 7. On the Select Version page, select **1.3** (**2.3** if you are configuring `web.xml`) and click **Next**. 8. On the Summary page, click **Finish**.

`orion-application.xml`
`data-sources.xml`
`oc4j-connectors.xml`
	1. Select the project in the Applications Navigator.
	2. Select **File > New** to display the New Gallery.
	3. In **Categories**, expand **General** and select **Deployment Descriptors**.
	4. In **Items**, select **OC4J Deployment Descriptor Wizard** and click **OK**.
	5. Click **Next** in the wizard to display the Select Descriptor page.
	6. On the Select Descriptor page, select the file you want to configure and click **Next**.
	7. On the Select Version page, select the appropriate version and click **Next**.
	For `orion-application.xml`, select **1.2**.
	For `data-sources.xml`, select **1.0**.
	For `oc4j-connectors.xml`, select **10.0**.
	8. On the Summary page, click **Finish**.

22.6.3 Oracle Application Server Deployment Methods

Instead of deploying applications directly from JDeveloper, you can use JDeveloper to create the archive file, and then deploy the archive file using these methods:

- Using Application Server Control Console. For details, see the "Deploying with Application Server Control Console" chapter in the *Oracle Containers for J2EE Deployment Guide*.
- Using `admin_client.jar`. For details, see the "Deploying with the admin_client.jar Utility" chapter in the *Oracle Containers for J2EE Deployment Guide*.

You can access the *Oracle Containers for J2EE Deployment Guide* from the Oracle Application Server documentation library.

22.6.4 Oracle Application Server Deployment to Test Environments ("Automatic Deployment")

If you are deploying to a standalone OC4J environment that is not a production environment, you can configure OC4J to automatically deploy your application. This method is not recommended for production environments.

For details, see the "Automatic Deployment in OC4J" chapter in the *Oracle Containers for J2EE Deployment Guide*.

22.6.5 Oracle Application Server Deployment to Clustered Topologies

To deploy to clustered topologies, you can use any of the following methods:

- In JDeveloper, you can deploy to a "group" of Oracle Application Server instances. To do this, ensure that the connection to the Oracle Application Server is set to "group" instead of "single instance".
- You can use the `admin_client.jar` command-line utility. This utility enables you to deploy the application to all nodes in a cluster using a single command. `admin_client.jar` is shipped with Oracle Application Server 10.1.3.

  For details, see the "Deploying with the admin_client.jar Utility" chapter in the *Oracle Containers for J2EE Deployment Guide*.

22.7 Deploying to JBoss

This section describes deployment details that are specific to JBoss.

- Section 22.7.1, "JBoss Versions Supported"
- Section 22.7.2, "JBoss Deployment Notes"
- Section 22.7.3, "JBoss Deployment Methods"

22.7.1 JBoss Versions Supported

Table 22–4 shows the supported versions of JBoss:

Table 22–4 Support Matrix for JBoss

JBoss version	JDK version	J2EE version
4.0.2	1.5_04	1.4
4.0.3	1.5_04	1.4

22.7.2 JBoss Deployment Notes

- Before deploying applications that use ADF to JBoss, you need to install the ADF runtime libraries on JBoss. See Section 22.12, "Installing ADF Runtime Library on Third-Party Application Servers" for details.
- If you are running JBoss version 4.0.3, you need to delete the following directories from the JBoss home. This is to facilitate running JSP and ADF Faces components.
  - `deploy/jbossweb-tomcat55.sar/jsf-lib/`
  - `tmp`, `log`, and `data` directories (located at the same level as the `deploy` directory)

  After removing the directories, restart JBoss.

  If you do not remove these directories, you may get the following exception during runtime:

```
org.apache.jasper.JasperException
org.apache.jasper.servlet.JspServletWrapper.service(JspServletWrapper.java:370)
org.apache.jasper.servlet.JspServlet.serviceJspFile(JspServlet.java:314)
org.apache.jasper.servlet.JspServlet.service(JspServlet.java:264)
javax.servlet.http.HttpServlet.service(HttpServlet.java:810)
com.sun.faces.context.ExternalContextImpl.dispatch(ExternalContextImpl.java:322
)
com.sun.faces.application.ViewHandlerImpl.renderView(ViewHandlerImpl.java:130)
```

```
com.sun.faces.lifecycle.RenderResponsePhase.execute(RenderResponsePhase.java:87
)
com.sun.faces.lifecycle.LifecycleImpl.phase(LifecycleImpl.java:200)
com.sun.faces.lifecycle.LifecycleImpl.render(LifecycleImpl.java:117)
javax.faces.webapp.FacesServlet.service(FacesServlet.java:198)
org.jboss.web.tomcat.filters.ReplyHeaderFilter.doFilter(ReplyHeaderFilter.java:
81)

root cause

java.lang.NullPointerException
javax.faces.webapp.UIComponentTag.setupResponseWriter(UIComponentTag.java:615)
javax.faces.webapp.UIComponentTag.doStartTag(UIComponentTag.java:217)
org.apache.myfaces.taglib.core.ViewTag.doStartTag(ViewTag.java:71)
org.apache.jsp.untitled1_jsp._jspx_meth_f_view_0(org.apache.jsp.untitled1_
jsp:84)
org.apache.jsp.untitled1_jsp._jspService(org.apache.jsp.untitled1_jsp:60)
org.apache.jasper.runtime.HttpJspBase.service(HttpJspBase.java:97)
javax.servlet.http.HttpServlet.service(HttpServlet.java:810)
org.apache.jasper.servlet.JspServletWrapper.service(JspServletWrapper.java:322)
org.apache.jasper.servlet.JspServlet.serviceJspFile(JspServlet.java:314)
org.apache.jasper.servlet.JspServlet.service(JspServlet.java:264)
javax.servlet.http.HttpServlet.service(HttpServlet.java:810)
com.sun.faces.context.ExternalContextImpl.dispatch(ExternalContextImpl.java:322
)
com.sun.faces.application.ViewHandlerImpl.renderView(ViewHandlerImpl.java:130)
com.sun.faces.lifecycle.RenderResponsePhase.execute(RenderResponsePhase.java:87
)
com.sun.faces.lifecycle.LifecycleImpl.phase(LifecycleImpl.java:200)
com.sun.faces.lifecycle.LifecycleImpl.render(LifecycleImpl.java:117)
javax.faces.webapp.FacesServlet.service(FacesServlet.java:198)
org.jboss.web.tomcat.filters.ReplyHeaderFilter.doFilter(ReplyHeaderFilter.java:
81)
```

- To deploy applications directly from JDeveloper to JBoss, the directory where the target JBoss application server is installed must be accessible from JDeveloper. This means you need to run JDeveloper and JBoss on the same machine, or you need to map a network drive on the JDeveloper machine to the JBoss machine.

  This is required because JDeveloper needs to copy the EAR file to the `JBOSS_HOME\server\default\deploy` directory in the JBoss installation directory.

- For EJB applications, add a `jboss.xml` deployment descriptor file if you want to support JBoss-specific configuration options for the EJBs. For more information on this file, see `http://www.jboss.org`.

- In the Business Components Project Wizard, set the SQL Flavor to SQL92, and the Type Map to Java. This is necessary because ADF uses the emulated XA datasource implementation when the Business Components application is deployed as an EJB session bean.

- To test an EJB deployed to JBoss, create a library in JDeveloper for the JBoss client-side libraries. The JBoss client-side libraries are located in the `JBOSS_HOME/client` directory.

- For business components JSP applications, choose **Deploy to EAR file** from the context menu to deploy it as an EAR file. You must deploy this application to an EAR file and not a WAR file because JBoss does not add EJB references under the `java:comp/env/` JNDI namespace for a WAR file. If you have set up a connection in JDeveloper to your JBoss server, you can deploy the EAR file directly to the server.

22.7.3 JBoss Deployment Methods

You can deploy to JBoss directly if you have set up a connection in JDeveloper to your JBoss server. When you deploy from JDeveloper, it copies the EAR file to the `JBOSS_HOME\server\default\deploy` directory. JBoss deploys the EAR files that it finds in that directory. You do not have to restart JBoss in order to access the application.

22.8 Deploying to WebLogic

This section describes deployment details that are specific to WebLogic.

- Section 22.8.1, "WebLogic Versions Supported"
- Section 22.8.2, "WebLogic Versions 8.1 and 9.0 Deployment Notes"
- Section 22.8.3, "WebLogic 8.1 Deployment Notes"
- Section 22.8.5, "WebLogic Deployment Methods"

22.8.1 WebLogic Versions Supported

Table 22–5 shows the supported versions of WebLogic:

Table 22–5 Support Matrix for WebLogic

WebLogic version	JDK version	J2EE version
8.1 SP4	1.4 ADF applications have been certified against the Sun JDK, but not the JRockit JDK.	1.3
9.0	1.5	1.4

22.8.2 WebLogic Versions 8.1 and 9.0 Deployment Notes

- Before deploying applications that use ADF to WebLogic, you need to install the ADF runtime libraries on WebLogic. See Section 22.12, "Installing ADF Runtime Library on Third-Party Application Servers" for details.
- When you click Test Connection in the Create Application Server Connection wizard, you may get the following exception:

  ```
  Class Not Found Exception -
  weblogic.jndi.WLInitialContextFactory
  ```

  This exception occurs when `weblogic.jar` is not in JDeveloper's classpath. You may ignore this exception and continue with the deployment.
- You may get an exception in JDeveloper when trying to deploy large EAR files. The workaround is to deploy the application using the server console.

22.8.3 WebLogic 8.1 Deployment Notes

- This version of WebLogic supports JDK 1.4. This means that you need to configure JDeveloper to build your applications with JDK 1.4 (such as the JDK provided by WebLogic) instead of JDK 1.5. See Section 22.11, "Deploying to Application Servers That Support JDK 1.4" for details.
- WebLogic 8.1 is only J2EE 1.3 compliant. This means that you need to create an `application.xml` file that complies with J2EE 1.3. To create this file in JDeveloper, make the following selections:

1. Select the project in the Applications Navigator.
2. Select **File > New** to display the New Gallery.
3. In **Categories**, expand **General** and select **Deployment Descriptors**.
4. In **Items**, select **J2EE Deployment Descriptor Wizard** and click **OK**.
5. Click **Next** in the wizard to display the Select Descriptor page.
6. On the Select Descriptor page, select **application.xml** and click **Next**.
7. On the Select Version page, select **1.3** and click **Next**.
8. On the Summary page, click **Finish**.

- Similarly, your `web.xml` needs to be compliant with J2EE 1.3 (which corresponds to servlet 2.3 and JSP 1.2). To create this file in JDeveloper, follow the steps as shown above, except that you select **web.xml** in the Select Descriptor page, and **2.3** in the Select Version page.
- If you are using Struts in your application, you need to create the `web.xml` file at version 2.3 first, then create any required Struts configuration files. If you reverse the order (create Struts configuration files first), this will not work because creating a Struts configuration file also creates a `web.xml` file if one does not already exist, but this `web.xml` is for J2EE 1.4, which will not work with WebLogic 8.1.

22.8.4 WebLogic 9.0 Deployment Notes

- When you are deploying to WebLogic 9.0 from JDeveloper, ensure that the HTTP Tunneling property is enabled in the WebLogic console. This property is located under Servers > *ServerName* > Protocols. *ServerName* refers to the name of your WebLogic server.

22.8.5 WebLogic Deployment Methods

You can deploy directly to WebLogic if you have set up a connection in JDeveloper to your WebLogic server.

You can also deploy using the WebLogic console (for example: `http://<weblogic_host:port>/console/`).

22.9 Deploying to WebSphere

This section describes deployment details that are specific to WebSphere.

- Section 22.9.1, "WebSphere Versions Supported"
- Section 22.9.2, "WebSphere Deployment Notes"
- Section 22.9.3, "WebSphere Deployment Methods"

22.9.1 WebSphere Versions Supported

Table 22–6 shows the supported versions of WebSphere:

Table 22–6 Support Matrix for WebSphere

WebSphere version	JDK version	J2EE version
6.0.1	1.4.2	1.4

22.9.2 WebSphere Deployment Notes

- This version of WebSphere supports JDK 1.4. This means that you need to configure JDeveloper to build your applications with JDK 1.4 instead of JDK 1.5. See Section 22.11, "Deploying to Application Servers That Support JDK 1.4" for details.
- Before you can deploy applications that use ADF to WebSphere, you need to install the ADF runtime libraries on WebSphere. See Section 22.12.2, "Configuring WebSphere 6.0.1 to Run ADF Applications" for details. Note that JDeveloper cannot connect to WebSphere application servers. This means you have to use the manual method of installing the ADF runtime libraries.
- Check that you have the following lines in the `web.xml` file for the ADF application you want to deploy:

```
<servlet>
  <servlet-name>jsp</servlet-name>
  <servlet-class>com.ibm.ws.webcontainer.jsp.servlet.JspServlet</servlet-class>
</servlet>
```

- You may need to configure data sources and other variables for deployment. Use the correct DataSource name, JNDI name, URLs, etc, that were used when creating the application.
- After deploying the application, you need to add the appropriate shared library reference for the ADF application, depending on your application's SQL flavor and type map. You created the shared library in step 5 on page 22-20.

22.9.3 WebSphere Deployment Methods

You can deploy using the WebSphere console (for example: `http://<websphere_host:port>/ibm/console/`).

22.10 Deploying to Tomcat

This section describes deployment details that are specific to Tomcat.

22.10.1 Tomcat Versions Supported

Table 22–7 shows the supported versions of Tomcat:

Table 22–7 Support Matrix for Tomcat

Tomcat version	JDK version	J2EE version
5.5.9	1.5	1.4

22.10.2 Tomcat Deployment Notes

- Before deploying applications that use ADF to Tomcat, you need to install the ADF runtime libraries on Tomcat. See Section 22.12, "Installing ADF Runtime Library on Third-Party Application Servers" for details.
- After you install the ADF runtime libraries, rename the file `TOMCAT_HOME/common/jlib/bc4jdomgnrc` to `bc4jdomgnrc.jar` (that is, add the `.jar` extension to the filename). This file is required for users who are using the Java type mappings.

- You can deploy applications to Tomcat from JDeveloper (if you have set up a connection to your Tomcat server), or you can also deploy applications using the Tomcat console.

22.11 Deploying to Application Servers That Support JDK 1.4

If you are deploying to an application server that uses JDK 1.4, you need to configure JDeveloper to build your applications using JDK 1.4. By default, JDeveloper 10.1.3 uses JDK 1.5. If you build an application with JDK 1.5 and run it on an application server that supports JDK 1.4, you may get "unsupported class version" errors.

Application servers that support JDK 1.4 include Oracle Application Server Release 2 (10.1.2), WebLogic 8.1, and WebSphere.

To configure JDeveloper to build projects with JDK 1.4:

1. Install J2SE 1.4 on the machine running JDeveloper.
2. Configure JDeveloper with the J2SE 1.4 that you installed:
   a. In JDeveloper, choose Tools > Manage Libraries. This displays the Manage Libraries dialog.
   b. In the Manage Libraries dialog, choose the **J2SE Definitions** tab.
   c. On the right-hand side, click the **Browse** button for the **J2SE Executable** field and navigate to the `J2SE_1.4/bin/java.exe` file, where `J2SE_1.4` refers to the directory where you installed J2SE 1.4.
   d. Click **OK**.
3. Configure your project to use J2SE 1.4:
   a. In the Project Properties dialog for your project, select **Libraries** on the left-hand side.
   b. On the right-hand side, click the **Change** button for the **J2SE Version** field. This displays the Edit J2SE Definition dialog.
   c. In the Edit J2SE Definition dialog, on the left-hand side, select **1.4** under **User**.
   d. Click **OK** in the Edit J2SE Definition dialog.
   e. Click **OK** in the Project Properties dialog.

22.11.1 Switching Embedded OC4J to JDK 1.4

When you run an Oracle JDeveloper 10.1.3 application using the Embedded OC4J server, the application is configured for JDK 1.5. If you then try to switch to JDK 1.4, you will see JSP compile failures. To remedy this you need to force the application files to be re-compiled when OC4J is restarted with JDK 1.4. To configure Embedded OC4J to JDK 1.4:

1. Configure JDeveloper 10.1.3.4 according to the steps above.
2. Stop the embedded OC4J server instance.
3. Delete the following directory:

   ORACLE_HOME/j2ee/instance/application-deployments
4. Start the embedded server again.

22.12 Installing ADF Runtime Library on Third-Party Application Servers

Before you can deploy applications that use ADF on third-party application servers, you need to install the ADF runtime libraries on those application servers. You can perform the installation using a wizard or you can do it manually:

- For WebLogic, JBoss, and Tomcat, you can install the ADF runtime libraries from JDeveloper using the ADF Runtime Installer wizard. See Section 22.12.1, "Installing the ADF Runtime Libraries from JDeveloper".
- For WebSphere, you have to install the ADF runtime libraries manually. See Section 22.12.2, "Configuring WebSphere 6.0.1 to Run ADF Applications".
- For all application servers, you can install the ADF runtime libraries manually. See Section 22.12.3, "Installing the ADF Runtime Libraries Manually".

22.12.1 Installing the ADF Runtime Libraries from JDeveloper

You can install the ADF runtime libraries from JDeveloper on selected application servers. The supported application servers are listed in the Tools > ADF Runtime Installer submenu.

Note that for WebSphere, you need to install the libraries manually. See Section 22.12.2, "Configuring WebSphere 6.0.1 to Run ADF Applications".

To install the ADF Runtime Libraries from JDeveloper:

1. Stop all instances of the target application server.
2. (WebLogic only) Create a new WebLogic domain, if you do not already have one. You will install the ADF runtime libraries in the domain.

   Steps for creating a domain in WebLogic are provided here for your convenience.

   Note: The domain must be configured to use Sun's JDK.

   Steps for Creating Domains in WebLogic 8.1:

   a. From the Start menu, choose Programs > BEA WebLogic Platform 8.1 > Configuration Wizard. This starts up the Configuration wizard.
   b. On the Create or Extend a Configuration page, select **Create a new WebLogic Configuration**. Click **Next**.
   c. On the Select a Configuration Template page, select **Basic WebLogic Server Domain**. Click **Next**.
   d. On the Choose Express or Custom Configuration page, select **Express**. Click **Next**.
   e. On the Configure Administrative Username and Password page, enter a username and password. Click **Next**.
   f. On the Configure Server Start Mode and Java SDK page, make sure you select Sun's JDK. Click **Next**.
   g. On the Create WebLogic Configuration page, you can change the domain name. For example, you might want to change it to `jdevdomain`.

   Steps for Creating Domains in WebLogic 9.0:

a. From the Start menu, choose Programs > BEA Products > Tools > Configuration Wizard. This starts up the Configuration wizard.

b. On the Welcome page, select **Create a new WebLogic Domain**. Click **Next**.

c. On the Select a Domain Source page, select **Generate a domain configured automatically to support the following BEA products**. Click **Next**.

d. On the Configure Administrator Username and Password page, enter a username and password. Click **Next**.

e. On the Configure Server Start Mode and JDK page, make sure you select Sun's JDK. Click **Next**.

f. On the Customize Environment and Services Settings page, select **No**. Click **Next**.

g. On the Create WebLogic Domain page, set the domain name. For example, you might want to set it to `jdevdomain`. Click **Create**.

3. Start the ADF Runtime Installer wizard by choosing Tools > ADF Runtime Installer > *Application_Server_Type*. *Application_Server_Type* is the type of the target application server (for example, Oracle Application Server, WebLogic, JBoss, or standalone OC4J).

4. Proceed through the pages in the wizard. For detailed instructions for any page in the wizard, click **Help**. You need to enter the following information in the wizard:

   - On the Home Directory page, select the home or root directory of the target application server.
   - (WebLogic only) On the Domain Directory page, select the home directory of the WebLogic domain where you want to install the ADF libraries. You created this domain in step 2 on page 22-17.
   - On the Installation Options page, choose **Install the ADF Runtime Libraries**.
   - On the Summary page, check the details and click **Finish**.

5. (WebLogic only) Edit WebLogic startup files so that WebLogic includes the ADF runtime library when it starts up.

   Steps for WebLogic 8.1:

   a. Make a backup copy of the `WEBLOGIC_HOME\user_projects\domains\jdevdomain\startWebLogic.cmd` (or `startWebLogic.sh`) file because you will be editing it in the next step. "`jdevdomain`" is the name of the domain that you created earlier in step 2 on page 22-17.

   b. In the `startWebLogic.cmd` (or `startWebLogic.sh`) file, add the "`call "setupadf.cmd"`" line (for Windows) before the "`set CLASSPATH`" line:

   ```
   call "setupadf.cmd"
   set CLASSPATH=%WEBLOGIC_CLASSPATH%;%POINTBASE_CLASSPATH%;
           %JAVA_HOME%\jre\lib\rt.jar;%WL_HOME%\server\lib\webservices.jar;
           %CLASSPATH%
   ```

   The `setupadf.cmd` script was installed by the ADF Runtime Installer wizard in the `WEBLOGIC_HOME\user_projects\domains\jdevdomain` directory.

   c. To start WebLogic, change directory to the `jdevdomain` directory and run `startWebLogic.cmd`:

   ```
   > cd WEBLOGIC_HOME\user_projects\domains\jdevdomain
   ```

```
> startWebLogic.cmd
```

Steps for WebLogic 9.0:

a. Make a backup copy of the `%DOMAIN_HOME%\bin\setDomainEnv.cmd` file because you will be editing it in the next step.

`%DOMAIN_HOME%` is specified in the `startWebLogic.cmd` (or `startWebLogic.sh`) file. For example, if you named your domain `jdevdomain`, then `%DOMAIN_HOME%` would be `BEA_HOME\user_projects\domains\jdevdomain`. You created the domain earlier in step 2 on page 22-17.

b. In the `%DOMAIN_HOME%\bin\setDomainEnv.cmd` file, add the "`call "%DOMAIN_HOME%\setupadf.cmd"`" line before the "`set CLASSPATH`" line:

```
call "%DOMAIN_HOME%\setupadf.cmd"
set CLASSPATH=%PRE_CLASSPATH%;%WEBLOGIC_CLASSPATH%;%POST_CLASSPATH%;
    %WLP_POST_CLASSPATH%;%WL_HOME%\integration\lib\util.jar;%CLASSPATH%
```

c. If the "set CLASSPATH" line does not have `%CLASSPATH%`, then add it to the line, as shown above.

d. To start WebLogic, change directory to `%DOMAIN_HOME%` and run `startWebLogic.cmd`:

```
> cd %DOMAIN_HOME%
> startWebLogic.cmd
```

6. (WebLogic only) Before you run JDeveloper, configure JDeveloper to include the WebLogic client in its class path.

a. Make a backup copy of the `JDEVELOPER_HOME\jdev\bin\jdev.conf` file because you will be editing it in the next step.

b. Add the following line to the `jdev.conf` file:

```
AddJavaLibFile <WEBLOGIC_HOME>\server\lib\weblogic.jar
```

Replace `<WEBLOGIC_HOME>` with the fullpath to the directory where you installed WebLogic.

7. Restart the target application server. If you are running WebLogic, you may have already started up the server.

Managing Multiple Versions of the ADF Runtime Library

Application servers may contain different versions of the ADF runtime libraries, but at any time only one version (the active version) is accessible to deployed applications. The other versions are archived.

You can use the ADF Runtime Installer wizard to make a different version the active version. On the Installation Options page in the wizard, choose the Restore option.

22.12.2 Configuring WebSphere 6.0.1 to Run ADF Applications

Before you can run ADF applications on WebSphere 6.0.1, you have to perform these steps:

1. Create the `install_adflibs_1013.sh` (or `.cmd` on Windows) script, as follows:

   If you are running on UNIX:

   a. Copy the source shown in Section 22.12.2.1, "Source for install_adflibs_1013.sh Script" and paste it to a file. Save the file as `install_adflibs_1013.sh`.

   b. Enable execute permission on `install_adflibs_1013.sh`.

   ```
   > chmod a+x install_adflibs_1013.sh
   ```

   If you are running on Windows, copy the source shown in Section 22.12.2.2, "Source for install_adflibs_1013.cmd Script" and paste it to a file. Save the file as `install_adflibs_1013.cmd`.

   You will run the script later, in step 3.

2. Stop the WebSphere processes.

3. Run the `install_adflibs_1013.sh` (`.cmd` on Windows) script to install the ADF libraries, as follows:

   a. Set the ORACLE_HOME environment variable to point to the JDeveloper installation.

   b. Set the WAS_ADF_LIB environment variable to point to the location where you want to install the ADF library files. Typically this is the WebSphere home directory. The library files are installed in the `WAS_ADF_LIB/lib` and `WAS_ADF_LIB/jlib` directories.

   c. Run the script. *<script_dir>* refers to the directory where you created the script.

   ```
   > cd <script_dir>
   > install_adflib_1013.sh          // if on Windows, use the .cmd extension
   ```

4. Start WebSphere processes.

5. Use the WebSphere administration tools to create a new shared library. Depending on your application, you create one of the shared libraries below.

   - For applications that use Oracle SQL flavor and type map, create the ADF10.1.3-Oracle shared library:

     Set the name of the shared library to `ADF10.1.3-Oracle`.

     Set the classpath to include all the JAR files in `WAS_ADF_LIB\lib` and `WAS_ADF_LIB\jlib` except for `WAS_ADF_LIB\jlib\bc4jdomgnrc.jar`. This JAR file is used for generic type mappings.

     `WAS_ADF_LIB` refers to the directory that will be used as a library defined in the WebSphere console. `WAS_ADF_LIB` contains the ADF library files.

- For applications that use non-Oracle SQL flavor and type map, create the ADF10.1.3-Generic shared library:

  Set the name of the shared library to `ADF10.1.3-Generic`.

  Set the classpath to include `WAS_ADF_LIB\jlib\bc4jdomgnrc.jar` and all the JAR files in `WAS_ADF_LIB\lib` except for `bc4jdomorcl.jar`. `WAS_ADF_LIB` refers to the directory that will be used as a library defined in the WebSphere console. `WAS_ADF_LIB` contains the ADF library files.

6. Add the following parameter in the Java command for starting up WebSphere.

   ```
   -Djavax.xml.transform.TransformerFactory=org.apache.xalan.processor.TransformerFactoryImpl
   ```

7. Shut down and restart WebSphere so that it uses the new parameter.

22.12.2.1 Source for install_adflibs_1013.sh Script

Example 22–1 shows the source for the `install_adflibs_1013.sh` script. Instead of copying the ADF runtime library files manually to your WebSphere environment, you can use this script. See Section 22.12.2, "Configuring WebSphere 6.0.1 to Run ADF Applications" for details.

The `install_adflibs_1013.sh` script is for use on UNIX environments. If you are running on Windows, see Section 22.12.2.2, "Source for install_adflibs_1013.cmd Script".

Example 22–1 install_adflibs_1013.sh

```
#!/bin/sh

EXIT=0
if [ "$ORACLE_HOME" = "" ]
  then
    echo "Error: The ORACLE_HOME environment variable must be set before executing
this script."
    echo "This should point to your JDeveloper installation directory"
    EXIT=1
fi
if [ "$WAS_ADF_LIB" = "" ];
  then
    echo "Error: The WAS_ADF_LIB environment variable must be set before executing
this script."
    echo "This should point to the location where you would like the ADF jars to
be copied."
    EXIT=1
fi

if [ "$EXIT" -eq 0 ]
then

if [ ! -d $WAS_ADF_LIB ]; then
  mkdir $WAS_ADF_LIB
fi
if [ ! -d $WAS_ADF_LIB/lib ]; then
  mkdir $WAS_ADF_LIB/lib
fi
if [ ! -d $WAS_ADF_LIB/jlib ]; then
  mkdir $WAS_ADF_LIB/jlib
fi
```

```
# Core BC4J runtime
cp $ORACLE_HOME/BC4J/lib/adfcm.jar  $WAS_ADF_LIB/lib/
cp $ORACLE_HOME/BC4J/lib/adfm.jar  $WAS_ADF_LIB/lib/
cp $ORACLE_HOME/BC4J/lib/adfmweb.jar  $WAS_ADF_LIB/lib/
cp $ORACLE_HOME/BC4J/lib/adfshare.jar  $WAS_ADF_LIB/lib/
cp $ORACLE_HOME/BC4J/lib/bc4jct.jar  $WAS_ADF_LIB/lib/
cp $ORACLE_HOME/BC4J/lib/bc4jctejb.jar  $WAS_ADF_LIB/lib/
cp $ORACLE_HOME/BC4J/lib/bc4jdomorcl.jar  $WAS_ADF_LIB/lib/
cp $ORACLE_HOME/BC4J/lib/bc4jimdomains.jar  $WAS_ADF_LIB/lib/
cp $ORACLE_HOME/BC4J/lib/bc4jmt.jar  $WAS_ADF_LIB/lib/
cp $ORACLE_HOME/BC4J/lib/bc4jmtejb.jar  $WAS_ADF_LIB/lib/
cp $ORACLE_HOME/BC4J/jlib/dc-adapters.jar  $WAS_ADF_LIB/lib/
cp $ORACLE_HOME/BC4J/jlib/adf-connections.jar  $WAS_ADF_LIB/lib/

#  Core BC4J jlib runtime
cp $ORACLE_HOME/BC4J/jlib/bc4jdomgnrc.jar $WAS_ADF_LIB/jlib/
cp $ORACLE_HOME/BC4J/jlib/adfui.jar $WAS_ADF_LIB/lib/
cp $ORACLE_HOME/BC4J/jlib/adfmtl.jar $WAS_ADF_LIB/lib/

#  Oracle Home jlib runtime
cp $ORACLE_HOME/jlib/jdev-cm.jar $WAS_ADF_LIB/lib/
cp $ORACLE_HOME/jlib/jsp-el-api.jar $WAS_ADF_LIB/lib/
cp $ORACLE_HOME/jlib/oracle-el.jar $WAS_ADF_LIB/lib/
cp $ORACLE_HOME/jlib/commons-el.jar $WAS_ADF_LIB/lib/

#  Oracle MDS runtime
cp $ORACLE_HOME/jlib/commons-cli-1.0.jar $WAS_ADF_LIB/lib/
cp $ORACLE_HOME/jlib/xmlef.jar $WAS_ADF_LIB/lib/
cp $ORACLE_HOME/mds/lib/mdsrt.jar $WAS_ADF_LIB/lib/
cp $ORACLE_HOME/mds/lib/concurrent.jar $WAS_ADF_LIB/lib/

#  Oracle Diagnostic
cp %ORACLE_HOME%/diagnostics/lib/commons-cli-1.0.jar $WAS_ADF_LIB/lib/

#  SQLJ Runtime
cp $ORACLE_HOME/sqlj/lib/translator.jar  $WAS_ADF_LIB/lib/
cp $ORACLE_HOME/sqlj/lib/runtime12.jar  $WAS_ADF_LIB/lib/

#  Intermedia Runtime
cp $ORACLE_HOME/ord/jlib/ordhttp.jar  $WAS_ADF_LIB/lib/
cp $ORACLE_HOME/ord/jlib/ordim.jar  $WAS_ADF_LIB/lib/

#  Toplink
cp $ORACLE_HOME/toplink/jlib/toplink.jar  $WAS_ADF_LIB/lib/
cp $ORACLE_HOME/toplink/jlib/antlr.jar  $WAS_ADF_LIB/lib/

#  OJMisc
cp $ORACLE_HOME/jlib/ojmisc.jar  $WAS_ADF_LIB/lib/

#  XML Parser
cp $ORACLE_HOME/lib/xmlparserv2.jar  $WAS_ADF_LIB/lib/

#  JDBC
cp $ORACLE_HOME/jdbc/lib/ojdbc14.jar  $WAS_ADF_LIB/lib/
cp $ORACLE_HOME/jdbc/lib/ojdbc14dms.jar  $WAS_ADF_LIB/lib/
cp $ORACLE_HOME/lib/dms.jar  $WAS_ADF_LIB/lib/

#  XSQL Runtime
cp $ORACLE_HOME/lib/xsqlserializers.jar  $WAS_ADF_LIB/lib/
cp $ORACLE_HOME/lib/xsu12.jar  $WAS_ADF_LIB/lib/
```

```
cp $ORACLE_HOME/lib/xml.jar  $WAS_ADF_LIB/lib/

fi
```

22.12.2.2 Source for install_adflibs_1013.cmd Script

Example 22–2 shows the source for the `install_adflibs_1013.cmd` script. Instead of copying the ADF runtime library files manually to your WebSphere environment, you can use this script. See Section 22.12.2, "Configuring WebSphere 6.0.1 to Run ADF Applications" for details.

The `install_adflibs_1013.cmd` script is for use on Windows environments. If you are running on UNIX, see Section 22.12.2.1, "Source for install_adflibs_1013.sh Script".

Example 22–2 install_adflibs_1013.cmd

```
@echo off
if {%ORACLE_HOME%} =={} goto :oracle_home

if {%WAS_ADF_LIB%} =={} goto :was_adf_lib

mkdir %WAS_ADF_LIB%
mkdir %WAS_ADF_LIB%\lib
mkdir %WAS_ADF_LIB%\jlib

@REM Core BC4J runtime
copy %ORACLE_HOME%\BC4J\lib\adfcm.jar  %WAS_ADF_LIB%\lib\
copy %ORACLE_HOME%\BC4J\lib\adfm.jar  %WAS_ADF_LIB%\lib\
copy %ORACLE_HOME%\BC4J\lib\adfmweb.jar  %WAS_ADF_LIB%\lib\
copy %ORACLE_HOME%\BC4J\lib\adfshare.jar  %WAS_ADF_LIB%\lib\
copy %ORACLE_HOME%\BC4J\lib\bc4jct.jar  %WAS_ADF_LIB%\lib\
copy %ORACLE_HOME%\BC4J\lib\bc4jctejb.jar  %WAS_ADF_LIB%\lib\
copy %ORACLE_HOME%\BC4J\lib\bc4jdomorcl.jar  %WAS_ADF_LIB%\lib\
copy %ORACLE_HOME%\BC4J\lib\bc4jimdomains.jar  %WAS_ADF_LIB%\lib\
copy %ORACLE_HOME%\BC4J\lib\bc4jmt.jar  %WAS_ADF_LIB%\lib\
copy %ORACLE_HOME%\BC4J\lib\bc4jmtejb.jar  %WAS_ADF_LIB%\lib\
copy %ORACLE_HOME%\BC4J\lib\collections.jar  %WAS_ADF_LIB%\lib\
copy %ORACLE_HOME%\BC4J\lib\adfbinding.jar  %WAS_ADF_LIB%\lib\
copy %ORACLE_HOME%\BC4J\jlib\dc-adapters.jar  %WAS_ADF_LIB%\lib\
copy %ORACLE_HOME%\BC4J\jlib\adf-connections.jar  %WAS_ADF_LIB%\lib\

@REM Core BC4J jlib runtime
copy %ORACLE_HOME%\BC4J\jlib\bc4jdomgnrc.jar %WAS_ADF_LIB%\jlib\
copy %ORACLE_HOME%\BC4J\jlib\adfui.jar %WAS_ADF_LIB%\lib\
copy %ORACLE_HOME%\BC4J\jlib\adfmtl.jar %WAS_ADF_LIB%\lib\

@REM Oracle Home jlib runtime
copy %ORACLE_HOME%\jlib\jdev-cm.jar %WAS_ADF_LIB%\lib\
copy %ORACLE_HOME%\jlib\jsp-el-api.jar %WAS_ADF_LIB%\lib\
copy %ORACLE_HOME%\jlib\oracle-el.jar %WAS_ADF_LIB%\lib\
copy %ORACLE_HOME%\jlib\commons-el.jar %WAS_ADF_LIB%\lib\

@REM Oracle MDS runtime
copy %ORACLE_HOME%\jlib\commons-cli-1.0.jar %WAS_ADF_LIB%\lib\
copy %ORACLE_HOME%\jlib\xmlef.jar %WAS_ADF_LIB%\lib\
copy %ORACLE_HOME%\mds\lib\mdsrt.jar %WAS_ADF_LIB%\lib\
copy %ORACLE_HOME%\mds\lib\concurrent.jar %WAS_ADF_LIB%\lib\
```

```
@REM Oracle Diagnostic
copy %ORACLE_HOME%\diagnostics\lib\ojdl.jar %WAS_ADF_LIB%\lib\

@REM SQLJ Runtime
copy %ORACLE_HOME%\sqlj\lib\translator.jar  %WAS_ADF_LIB%\lib\
copy %ORACLE_HOME%\sqlj\lib\runtime12.jar  %WAS_ADF_LIB%\lib\

@REM Intermedia Runtime
copy %ORACLE_HOME%\ord\jlib\ordhttp.jar  %WAS_ADF_LIB%\lib\
copy %ORACLE_HOME%\ord\jlib\ordim.jar  %WAS_ADF_LIB%\lib\

@REM Toplink
copy %ORACLE_HOME%\toplink\jlib\toplink.jar  %WAS_ADF_LIB%\lib\
copy %ORACLE_HOME%\toplink\jlib\antlr.jar  %WAS_ADF_LIB%\lib\

@REM OJMisc
copy %ORACLE_HOME%\jlib\ojmisc.jar  %WAS_ADF_LIB%\lib\

@REM XML Parser
copy %ORACLE_HOME%\lib\xmlparserv2.jar  %WAS_ADF_LIB%\lib\

@REM JDBC
copy %ORACLE_HOME%\jdbc\lib\ojdbc14.jar  %WAS_ADF_LIB%\lib\
copy %ORACLE_HOME%\jdbc\lib\ojdbc14dms.jar  %WAS_ADF_LIB%\lib\
copy %ORACLE_HOME%\lib\dms.jar  %WAS_ADF_LIB%\lib\

@REM XSQL Runtime
copy %ORACLE_HOME%\lib\xsqlserializers.jar  %WAS_ADF_LIB%\lib\
copy %ORACLE_HOME%\lib\xsu12.jar  %WAS_ADF_LIB%\lib\
copy %ORACLE_HOME%\lib\xml.jar  %WAS_ADF_LIB%\lib\

goto :end

:oracle_home
@echo Set the ORACLE_HOME pointing to the directory of your 10.1.3 JDeveloper
installation.

:was_adf_lib
if {%WAS_ADF_LIB%} =={} @echo Set the WAS_ADF_LIB environment variable pointing to
the directory where you would like to install ADF libraries.

:end
```

22.12.3 Installing the ADF Runtime Libraries Manually

Instead of using the ADF Runtime Installer wizard in JDeveloper to install the libraries, you can also install the libraries manually on your target application server.

Table 22–8 lists the files that you must copy to your application server before you deploy any ADF applications. In the table, `JDEV_INSTALL` refers to the directory where you installed JDeveloper.

- For JBoss, the destination directory is `JBOSS_HOME/server/default/lib`.
- For WebLogic, the destination directory is `WEBLOGIC_HOME/ADF/lib`. You have to create the `ADF` directory, and under it, the `lib` and `jlib` directories.
- For Tomcat, the destination directory is `TOMCAT_HOME/common/lib`.

Table 22–8 ADF Runtime Library Files to Copy

Copy These Files:	Notes
From `JDEV_INSTALL/BC4J/lib`: ■ `adfcm.jar` ■ `adfm.jar` ■ `adfmweb.jar` ■ `adfshare.jar` ■ `bc4jct.jar` ■ `bc4jctejb.jar` ■ `bc4jdomorcl.jar` or `bc4jdomgnrc.jar` **Note:** Only one of these files is required, depending on which mapping type you used to build your application. If you are using the Oracle type mappings, copy `bc4jdomorcl.jar`. If the application was built using "Java" type mappings, copy `bc4jdomgnrc.jar` instead. `bc4jdomgnrc.jar` is located in `JDEV_INSTALL/BC4J/jlib`. ■ `bc4jimdomains.jar` ■ `bc4jmt.jar` ■ `bc4jmtejb.jar` ■ `collections.jar` ■ `adfbinding.jar`	These are the ADF runtime library files.
From `JDEV_INSTALL/BC4J/jlib`: ■ `adfmtl.jar` ■ `bc4jdomgnrc.jar` (see the note above) ■ `adfui.jar`	These are the ADF runtime library files.
From `JDEV_INSTALL/jlib`: ■ `jdev-cm.jar` ■ `commons-el.jar` ■ `oracle-el.jar` ■ `jsp-el-api.jar`	These are the JDeveloper runtime library files.
From `JDEV_INSTALL/jlib`: ■ `commons-cli-1.0.jar` ■ `xmlef.jar` From `JDEV_INSTALL/mds/lib`: ■ `mdsrt.jar` ■ `concurrent.jar`	These are the Oracle MDS files.
From `JDEV_INSTALL/diagnostics/lib`: ■ `ojdl.jar`	These are the Oracle diagnostics files.
From `JDEV_INSTALL/jlib`: ■ `ojmisc.jar`	These are the OJMisc runtime files.
From `JDEV_INSTALL/lib`: ■ `xmlparserv2.jar`	This file is for XML support.

Table 22–8 (Cont.) ADF Runtime Library Files to Copy

Copy These Files:	Notes
From `JDEV_INSTALL/toplink/jlib`: ■ `toplink.jar` ■ `antlr.jar`	These are the TopLink library files.
From `JDEV_INSTALL/lib`: ■ `xml.jar` ■ `xsqlserializers.jar` ■ `xsu12.jar`	These are the XSQL library files.
From `JDEV_INSTALL/ord/jlib`: ■ `ordhttp.jar` ■ `ordim.jar`	These files are for *inter*Media Text support. *inter*Media Text is a feature for storing, retrieving, and manipulating audio, document, image, and video data in an Oracle database.
From `JDEV_INSTALL/sqlj/lib`: ■ `runtime12.jar` ■ `translator.jar`	These are the SQLJ runtime library files.
From `JDEV_INSTALL/jdbc/lib`: ■ `ojdbc14.jar` ■ `ojdbc14dms.jar` From `JDEV_INSTALL/lib`: ■ `dms.jar`	These are the JDBC runtime library files.
From `JDEV_INSTALL/javacache/lib`: ■ `cache.jar`	These are the Java Cache runtime library files.
From `JDEV_INSTALL/BC4J/redist`: ■ `webapp.war` or `bc4j.ear`	This file is for Business Components web application image and cascading style sheet support. If you are running Tomcat, copy the `webapp.war` file to the `TOMCAT_HOME/webapps` directory. If you are running JBoss, copy the `bc4j.ear` file to the `JBOSS_HOME/server/default/deploy` directory.

The destination directory (the directory to which you copy these files) depends on your application server:

22.12.3.1 Installing the ADF Runtime Libraries from a Zip File

You can also install the ADF runtime libraries by downloading `adfinstaller.zip` from OTN and following the directions below.

To install the ADF Runtime Libraries:

1. To initiate the download, go to the JDeveloper Download page on OTN, here:

   `http://www.oracle.com/technology/software/products/jdev/index.html`

   Unzip `adfinstaller.zip` to the target directory.

2. Set the `DesHome` variable in the `adfinstaller.properties` file to specify the home directory of the destination application server:

   For example:

   Oracle AS: `DesHome=c:\\oas1013`

OC4J: `DesHome=c:\\oc4j`

JBoss: `DesHome=c:\\jboss-4.0.3`

Tomcat: `DesHome=c:\\jakarta-tomcat-5.5.9`

WebLogic: `DesHome=c:\\bea\weblogic90` (note server home directory is in weblogic subdirectory)

3. Set the `type` variable in the `adfinstaller.properties` file to specify the platform for the application server where the ADF libraries are to be installed. The choices are `OC4J/AS/TOMCAT/JBOSS/WEBLOGIC`.

   For example:

   ```
   type=AS
   ```

4. Set the `UserHome` variable in the `adfinstaller.properties` file to specify the WebLogic domain for which ADF is being configured. This setting is only used for WebLogic, and ignored for all other platforms. For example:

   ```
   UserHome= c:\\bea\weblogic90\\user_
   projects\\domains\\adfdomain
   ```

5. Shut down all instances of the application server running on the target platform.

6. Run the following command if you only wish to see the version of the ADF Installer:

   ```
   java -jar runinstaller.jar –version
   ```

7. Run the following command on the command line prompt:

   ```
   java -jar runinstaller.jar adfinstaller.properties
   ```

22.12.4 Deleting the ADF Runtime Library

If you used the wizard to install the ADF runtime library, you should use the wizard to delete the library. On the Installation Options page in the wizard, choose the **Delete** option.

If you installed the ADF runtime library manually, you can just manually delete the files from your application server.

22.13 Verifying Deployment and Troubleshooting

After you deploy your application, test it to ensure that it runs correctly on the target application server. This section provides some common troubleshooting tips.

- Section 22.13.1, "How to Test Run Your Application"
- Section 22.13.2, ""Class Not Found" or "Method Not Found" Errors"
- Section 22.13.3, "Application Is Not Using data-sources.xml File on Target Application Server"
- Section 22.13.4, "Using jazn-data.xml with the Embedded OC4J Server"

22.13.1 How to Test Run Your Application

Once you've deployed the application, you can run it from the application server. To test run your application, open a browser window and enter a URL of the following type:

- For Oracle Application Server: http://<host>:port/<context root>/<page>
- For Faces pages: http://<host>:port/<context root>/faces/<page>

22.13.2 "Class Not Found" or "Method Not Found" Errors

Problem

You get "Class Not Found" or "Method Not Found" errors during runtime.

Solution

Check that ADF runtime libraries are installed on the target application server, and that the libraries are at the correct version.

You can use the ADF Runtime Installer wizard in JDeveloper to check the version of the ADF runtime libraries. To launch the wizard, choose **Tools > ADF Runtime Installer >** ***Application_Server_Type. Application_Server_Type*** is the type of the target application server (for example, WebLogic, JBoss, or standalone OC4J).

22.13.3 Application Is Not Using data-sources.xml File on Target Application Server

Problem

After deploying and running your application, you find that your application is using the `data-sources.xml` file that is packaged in the application's EAR file, instead of using the `data-sources.xml` file on the target application server. You want the application to use the `data-sources.xml` file on the target application server.

Solution

When you create your EAR file in JDeveloper, choose not to include the `data-sources.xml` file. To do this:

1. Choose **Tools > Preferences** to display the Preferences dialog.
2. Select **Deployment** on the left side.
3. Deselect **Bundle Default data-sources.xml During Deployment**.
4. Click **OK**.
5. Re-create the EAR file.

Before redeploying your application, undeploy your old application and ensure that the `data-sources.xml` file on the target application server contains the appropriate entries needed by your application.

22.13.4 Using jazn-data.xml with the Embedded OC4J Server

If your application uses `jazn-data.xml`, you should be aware of how the embedded OC4J server uses this file: If the embedded OC4J server finds a `jazn-data.xml` file in the application's `META-INF` directory, then the embedded OC4J server will use it. The embedded OC4J server will also set the `<workspace>-oc4j-app.xml` file to point

to this `jazn-data.xml` file. This enables you to edit the `jazn-data.xml` file using the Embedded OC4J Server Preferences dialog.

If there is no `jazn-data.xml` file in `META-INF`, the embedded OC4J server will create a *<workspace>*`-jazn-data.xml` file in the workspace root. You would then have to go and edit that file (or use the Embedded OC4J Server Preferences dialog to do so).

Part IV

Appendices

Part IV contains the following appendices:

- Appendix A, "Reference ADF XML Files"
- Appendix B, "Reference ADF Binding Properties"

A

Reference ADF XML Files

This appendix provides reference for the Oracle ADF metadata files that you create in your data model and user interface projects. You may use this information when you want to edit the contents of the metadata these files define.

This appendix includes the following sections:

- Appendix A.1, "About the ADF Metadata Files"
- Appendix A.2, "ADF File Overview Diagram"
- Appendix A.3, "ADF File Syntax Diagram"
- Appendix A.4, "DataControls.dcx"
- Appendix A.5, "Structure Definition Files"
- Appendix A.6, "DataBindings.cpx"
- Appendix A.7, "<pageName>PageDef.xml"
- Appendix A.8, "web.xml"
- Appendix A.9, "j2ee-logging.xml"
- Appendix A.10, "faces-config.xml"
- Appendix A.11, "adf-faces-config.xml"
- Appendix A.12, "adf-faces-skins.xml"

A.1 About the ADF Metadata Files

Metadata files in the Oracle ADF application are structured XML files used by the application to:

- Specify the parameters, methods, and return values available to your application's Oracle ADF data control usages.
- Create objects in the Oracle ADF binding context and to define the runtime behavior of those objects.
- Define configuration information about the UI components in JSF and Oracle ADF Faces.
- Define application configuration information for the J2EE application server.

In the case of ADF bindings, you can use the binding-specific editors to customize the runtime properties of the binding objects. You can open a binding's editor when you display the Structure window for a page definition file and choose **Properties** from the context menu.

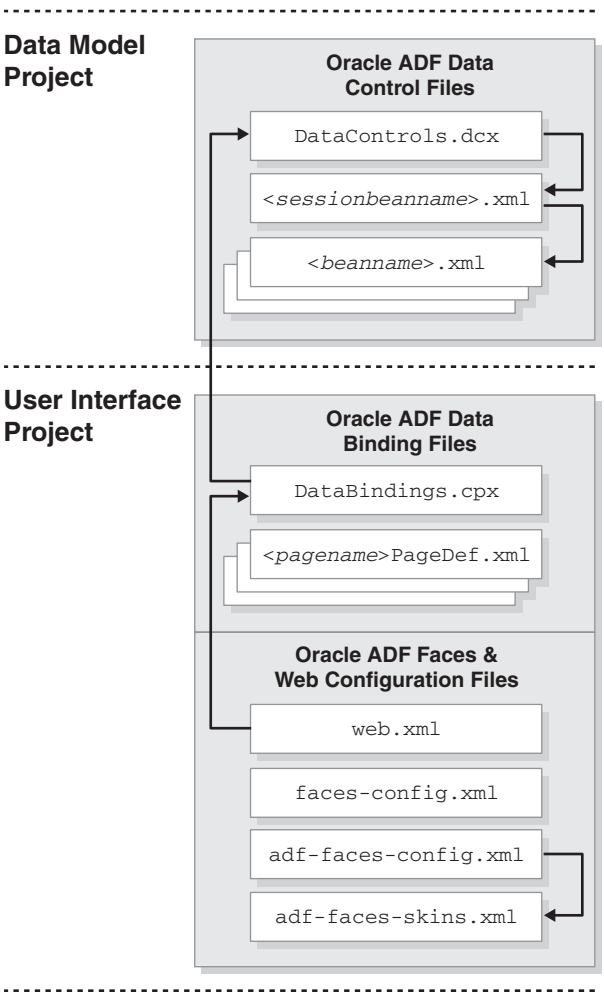
Additionally, you can view and edit the contents of any metadata file in JDeveloper's XML editor. The easiest way to work with these file is through the Structure window and Property Inspector. In the Structure window, you can select an element and in the Property Inspector, you can define attribute values for the element, often by choosing among dropdown menu choices. Use this reference to learn the choices you can select in the case of the Oracle ADF-specific elements.

A.2 ADF File Overview Diagram

The relationship between the Oracle ADF metadata files defines dependencies between the model data and the user interface projects. The dependencies are defined as file references within XML elements of the files.

Figure A–1 illustrates the hierarchical relationship of the XML metadata files that you may work with in the Oracle ADF application that uses an EJB session bean as a service interface to JavaBeans and JSF web pages.

Figure A–1 Oracle ADF File Hierarchy Overview for an EJB-based Web Application



A.2.1 Oracle ADF Data Control Files

These XML configuration files required in an Oracle ADF application appear in the data model project:

- `DataControls.dcx` is the registry for the data controls in the application. It contains information about the type of data control needed to work with a particular service (e.g. EJB, JavaBean, XML, web service, etc.) and how to construct the data control at runtime.

  For details about what you can configure in the `DataControls.dcx` file, see Section A.4.

- Structure definition files are the structure definition XML files for each business service type in the application. It contains information about the type of data control needed to work with a particular service (e.g. EJB, JavaBean, XML, web service, etc.) and how to construct the data control at runtime. For example, in the SRDemo application, which uses an EJB session bean as a service interface to JavaBeans, these files appear in the data model project:

  - *<sessionbeanname>*`.xml`—This is the structure definition XML file for each data type involved in the service interface. The name matches the name of that data type. For an EJB service interface, there is one structure definition file for the service class itself.

  - *<beanname>*`.xml`—This is the structure definition XML file for each JavaBean that appears as method return values or method arguments in the service interface.

    For details about what you can configure in the structure definition files, see Section A.5.

- `adfm.xml` is the registry for the data controls in the JDeveloper design time. The Data Control Palette uses the file to locate the DataControls.dcx file that appears in the data model project. For a sample of the `adfm.xml` file, see Section A.4.3.

A.2.2 Oracle ADF Data Binding Files

These standard XML configuration files for an Oracle ADF application appear in your user interface project:

- `DataBindings.cpx`— This file contains the pageMap, page definitions references, and data control references. The file is created the first time you create a data binding for a UI component (either from the Structure window or from the Data Control Palette). The `DataBindings.cpx` file defines the Oracle ADF binding context for the entire application. The binding context provides access to the bindings across the entire application. The `DataBindings.cpx` file also contains references to the *<pagename>*`PageDef.xml` files that define the metadata for the Oracle ADF bindings in each web page.

  See Appendix A.6, "DataBindings.cpx" for details about what you can configure in the `DataBindings.cpx` file.

- *<pagename>*`PageDef.xml`—This is the page definition XML file. This file is created each time you design a new web page using the Data Control Palette or Structure window. These XML files contain the metadata used to create the bindings that populate the data in the web page's UI components. For every web page that refers to an ADF binding, there must be a corresponding page definition file with binding definitions.

See Appendix A.7, "<pageName>PageDef.xml" for details about what you can configure in the <*pagename*>PageDef.xml file.

A.2.3 Oracle ADF Faces and Web Configuration Files

These XML configuration files required in a JSF application appear in your user interface project:

- web.xml—Part of the application's configuration is determined by the contents of its J2EE application deployment descriptor, web.xml. The web.xml file defines everything about your application that a server needs to know. The file plays a role in configuring the Oracle ADF data binding by setting up the ADFBindingFilter. Additional runtime settings include servlet runtime and initialization parameters, custom tag library location, and security settings.

  For details about ADF data binding and JSF configuration options, see Appendix A.8, "web.xml".

- faces-config.xml—This JSF configuration file lets you register a JSF application's resources, such as validators, converters, managed beans, and navigation rules. While an application can have more than one configuration resource file, and that file can have any name, typically the filename is faces-config.xml.

  For details about JSF configuration options, see Appendix A.10, "faces-config.xml".

- adf-faces-config.xml—This ADF Faces configuration file lets you configure ADF Faces-specific user interface features such as accessibility levels, custom skins, enhanced debugging, and right-to-left page rendering.

  For details about ADF Faces configuration options, see Appendix A.11, "adf-faces-config.xml".

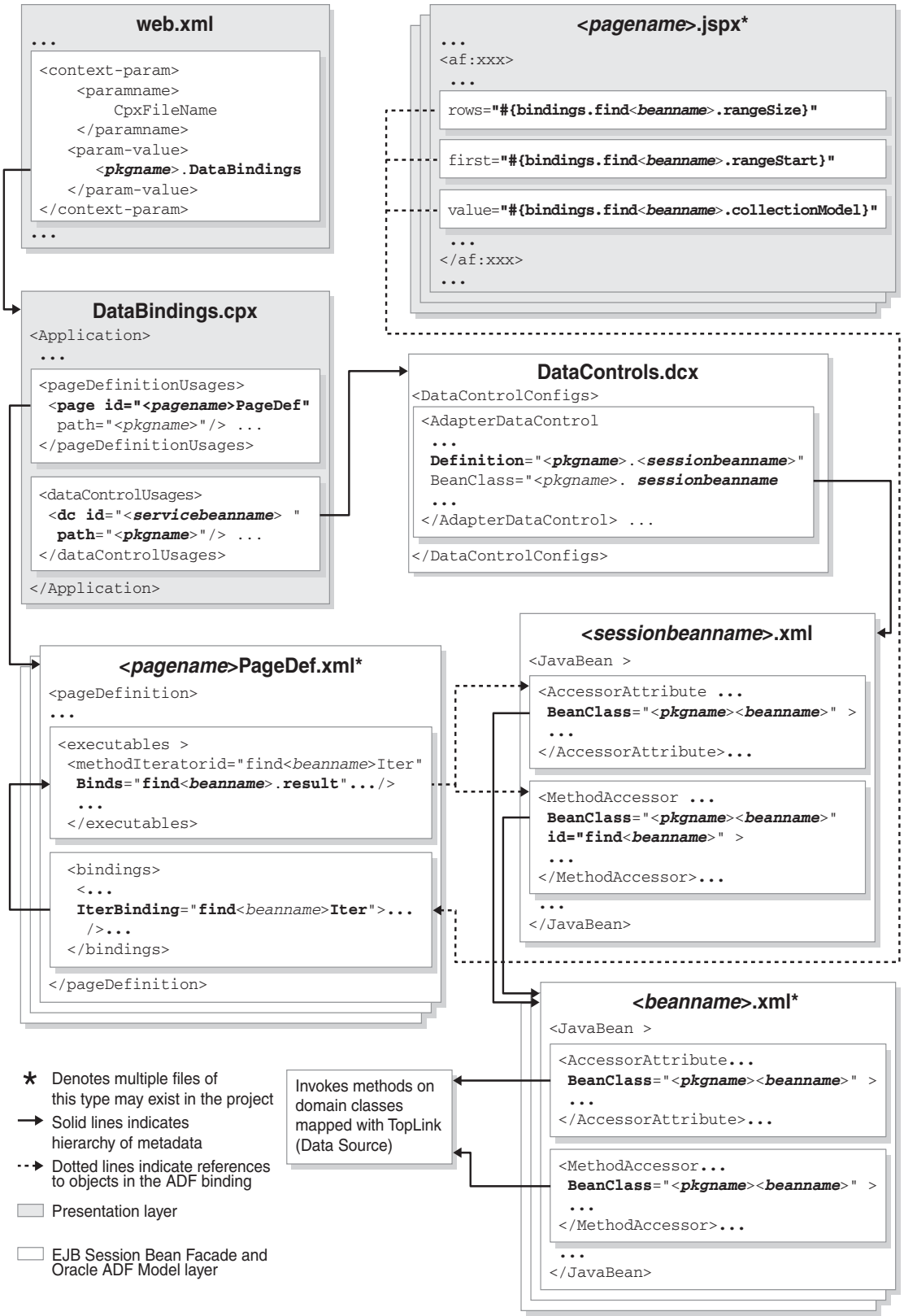
A.3 ADF File Syntax Diagram

Figure A–2 illustrates the hierarchical relationship of the XML metadata files that you may work with in the web application that uses an EJB session bean as a service interface to JavaBeans. At runtime, the objects created from these files interact in this sequence:

1. When the first request for an ADF databound web page occurs, the servlet registers the Oracle ADF servlet filter ADFBindingFilter named in the web.xml file.
2. The binding filter creates a binding context by reading the CpxFileName init param from the web.xml file.
3. The binding context creates the binding container by loading the <pagename>PageDef.xml file as referenced by the <pagemap> element from the DataBindings.cpx file.
4. The binding container's prepareModel phase prepares/refreshes all the executables.
5. An iterator binding gets executed by referencing the named method on the session bean facade specified by the data control factory named in the DataControls.dcx file.
6. The binding container also creates the bindings defined in the <bindings> section of the <pagename>PageDef.xml file for the mapped web page.

7. The web page references to ADF bindings through EL using the expression `#{bindings}` get resolved by accessing the binding container of the page.
8. The page pulls the available data from the bindings on the binding container.

Figure A–2 Oracle ADF File Hierarchy and Syntax Diagram for an EJB-based Web Application



A.4 DataControls.dcx

The DataControls.dcx file is created in the /src/package directory of the data model project folder when you create data controls on the business services. There can be one .dcx file for each model project. The .dcx file identifies the Oracle ADF model layer data control classes that facilitate the interaction between the client and the available business service. There will be one data control definition for each data control type used in the project.

The JDeveloper design time maintains path information for the DataControls.dcx file in the adfm.xml registry file located in the model project's META-INF folder. When you create a data control, JDeveloper will automatically update this file.

In the case of EJB, web services, and bean-based data controls, you can edit this file in the Property Inspector to alter data control settings. For example, you can use the .dcx file to customize the global properties of each data control, such as whether to turn on/off sorting. See Table A–1 for details about the attributes.

The Application Navigator displays the .dcx file in the default package of the Application Sources folder. When you double-click the file node, the data control description appears in the XML Source Editor. To edit the data control attributes, use the Property Inspector and select the desired attribute in the Structure window.

A.4.1 Syntax of the DataControls.dcx File

The toplevel element of the DataControls.dcx file is <DataControlConfigs>:

```
<DataControlConfigs xmlns="http://xmlns.oracle.com/adfm/configuration"
                    version="10.1.3.35.65" Package="oracle.srdemo.model"
                    id="DataControls">
```

where the XML namespace attribute (xmlns) specifies the URI to which the data controls bind at runtime. Only the package name is editable; all other attributes should have the values shown.

Figure A–3 displays the toplevel <DataControlConfigs> element. Note that the BaseDataControl element is a placeholder for the concrete data control types. In the SRDemo application, the data control type is the AdapterDataControl.

Figure A–3 Schema Overview for DataControl.dcx

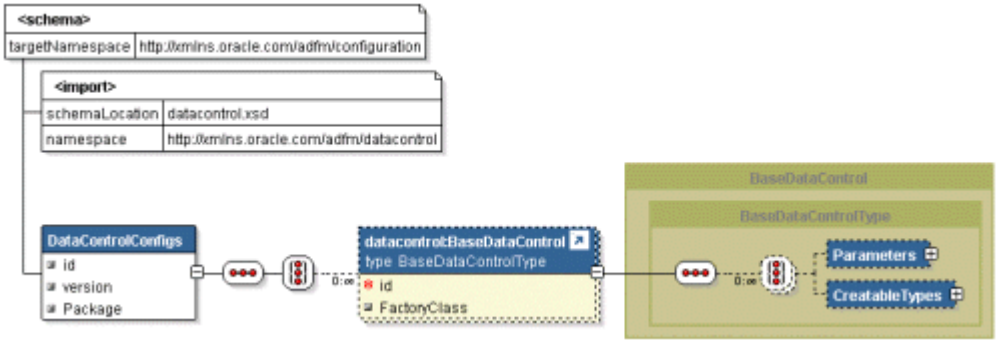
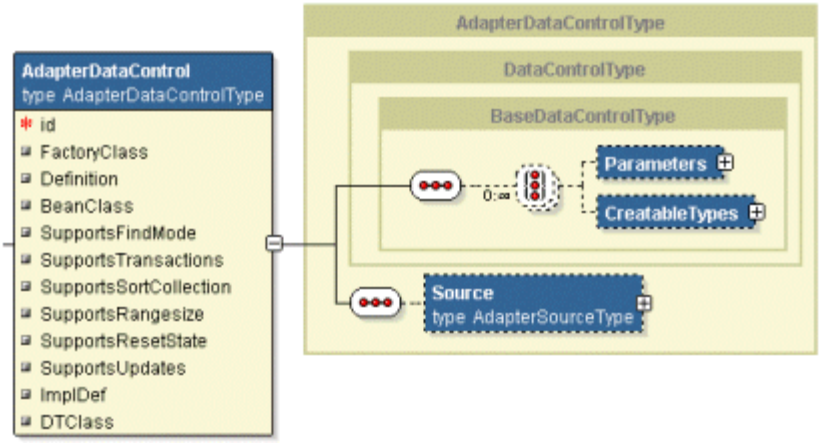


Figure A–4 displays the <AdapterDataControl> element that substitutes for the placeholder <BaseDataControlType> element. Note that each business service for which you have created a data control, will have its own <AdapterDataControl> definition.

Figure A–4 Schema Overview for Adapter Data Control in DataControl.dcx



The child elements have the following usages:

- <AdapterDataControl> created by the Adapter Data Control to define properties of the data control. The properties of the data control definition varies with the business service for which the data control is created.
- <CreatableTypes> defines types from which a constructor method may be called. For example, a type may be an EJB, TopLink object, JavaBean, or web service. Contains one or more child elements <TypeInfo>.
- <Source> defines the service for which the data control is created. Used only in the case of adapter-based data controls, such as EJB session facade data controls. In the case of the EJB session facade, contains the child element <ejb-definition>.

Table A–1 describes the attributes of the DataControls.dcx elements.

Table A–1 DataControls.dcx File Metadata

Element Syntax	Attributes	Attribute Description
<AdapterDataControl>	BeanClass	Fully qualified package name. Identifies the class for which the data control is created. In the case of the EJB session facade, this the session bean
	Definition	Identifies the class for which the data control is created. This is used for backward compatibility.
	FactoryClass	Fully qualified package name. Identifies the ADF runtime factory class that creates an instance of the data control.
	id	Unique identifier. Referenced by the DataBindings.cpx file.
	ImplDef	Internal.
	SupportsFindMode	Determines whether the data control supports preparing queries with user supplied criteria when preparing ADF iterator binding objects. Default is false for EJB session facade beans. Set to false if you want to globally prevent all iterator-bound web pages in the application from operating in find mode.
	SupportsRangeSize	Determines whether the data control supports returning a user-defined number of data objects when preparing ADF iterator binding objects. Default is false for EJB session facade beans.

Table A–1 (Cont.) DataControls.dcx File Metadata

Element Syntax	Attributes	Attribute Description
	SupportsResetState	This attribute is deprecated. Determines whether the data control supports resetting the state. Default is false for EJB session facade beans; not supported.
	SupportsSortCollection	Determines whether the data control supports data object sorting on the service collection. Default is false for EJB session facade beans; not supported for collections exposed by EJB session facade finder methods.
	SupportsTransaction	Determines whether the data control supports create, edit, and delete operations on the business service. Default is false for EJB session facade beans.
	SupportsUpdates	Determines whether the data control supports write operations. Default is true for EJB session facade beans.
	xmlns	URI used to identify the data control configuration namespace. At runtime, the data control object locates the runtime factory responsible for creating the definition objects for elements in its namespace.

`<CreateableTypes>`
`  <TypeInfo  />`
`</CreateableTypes>`
	FullName	Identifies the full type name of the Createable type. This element is defined only for those types that have constructors that appear in the Constructors folder of the Data Control Palette

`<Source>`
`  <ejb-definition  />`
`</Source>`
	ejb-business-interface	The Remote or Local interface that will be used to communicate with this Session bean

	ejb-interface-type	Either local or remote.
	ejb-name	The EJB's name.
	ejb-type	The EJB's type, currently only Session is supported.
	ejb-version	Either 3.0, 2.1, or 2.0.
	xmlns	This is for internal use only and refers to the schema namespace; this cannot be updated.

A.4.2 Sample of the DataControls.dcx File

Example A–1 shows the syntax for a sample data control definition file. Notice that there are two session beans identified by the AdapterDataControl: SRPublicFacade and SRAdminFacade.

Example A–1 DataControls.dcx Sample File

```
<?xml version="1.0" encoding="UTF-8" ?>
<DataControlConfigs xmlns="http://xmlns.oracle.com/adfm/configuration"
                    version="10.1.3.35.65" Package="oracle.srdemo.model"
                    id="DataControls">
```

```
  <AdapterDataControl id="SRPublicFacade"
                      FactoryClass="oracle.adf.model.adapter.DataControlFactoryImpl"
                        ImplDef="oracle.adfinternal.model.adapter.ejb.EjbDefinition"
                        SupportsTransactions="true" SupportsSortCollection="false"
                        SupportsResetState="false" SupportsRangesize="false"
                        SupportsFindMode="true"
                        Definition="oracle.srdemo.model.SRPublicFacade"
                        BeanClass="oracle.srdemo.model.SRPublicFacade"
                        xmlns="http://xmlns.oracle.com/adfm/datacontrol"
                        SupportsUpdates="true">
    <CreatableTypes>
      <TypeInfo FullName="oracle.srdemo.model.entities.Product"/>
      <TypeInfo FullName="oracle.srdemo.model.entities.ExpertiseArea"/>
      <TypeInfo FullName="oracle.srdemo.model.entities.ServiceHistory"/>
      <TypeInfo FullName="oracle.srdemo.model.entities.User"/>
      <TypeInfo FullName="oracle.srdemo.model.entities.ServiceRequest"/>
    </CreatableTypes>
    <Source>
      <ejb-definition ejb-version="3.0" ejb-name="SRPublicFacade"
                      ejb-type="Session" ejb-interface-type="local"
                      ejb-business-interface="oracle.srdemo.model.SRPublicFacade"
                      xmlns="http://xmlns.oracle.com/adfm/adapter/ejb"/>
    </Source>
  </AdapterDataControl>
  <AdapterDataControl id="SRAdminFacade"
                      FactoryClass="oracle.adf.model.adapter.DataControlFactoryImpl"
                        ImplDef="oracle.adfinternal.model.adapter.ejb.EjbDefinition"
                        SupportsTransactions="true" SupportsSortCollection="false"
                        SupportsResetState="false" SupportsRangesize="false"
                        SupportsFindMode="true"
                        Definition="oracle.srdemo.model.SRAdminFacade"
                        BeanClass="oracle.srdemo.model.SRAdminFacade"
                        xmlns="http://xmlns.oracle.com/adfm/datacontrol"
                        SupportsUpdates="true">
    <CreatableTypes>
      <TypeInfo FullName="oracle.srdemo.model.entities.ServiceHistory"/>
      <TypeInfo FullName="oracle.srdemo.model.entities.User"/>
      <TypeInfo FullName="oracle.srdemo.model.entities.ServiceRequest"/>
      <TypeInfo FullName="oracle.srdemo.model.entities.Product"/>
      <TypeInfo FullName="oracle.srdemo.model.entities.ExpertiseArea"/>
    </CreatableTypes>
    <Source>
      <ejb-definition ejb-version="3.0" ejb-name="SRAdminFacade"
                      ejb-type="Session" ejb-interface-type="local"
                      ejb-business-interface="oracle.srdemo.model.SRAdminFacade"
                      xmlns="http://xmlns.oracle.com/adfm/adapter/ejb"/>
    </Source>
  </AdapterDataControl>
</DataControlConfigs>
```

A.4.3 Sample of the adfm.xml File

The `adfm.xml` file is the registry for the data controls in the JDeveloper design time. For instance, the Data Control Palette uses the supplied path to facilitate locating the data controls used in the model project. When you create a data control, JDeveloper will automatically update this file located in the META-INF folder of the data model project.

Example A–2 shows the data control registry syntax.

Example A–2 Data Control Registry Syntax

```
<MetadataDirectory xmlns="http://xmlns.oracle.com/adfm/metainf"
             version="10.1.3.xx.xx">
        <DataControlRegistry path="test/model/DataControls.dcx"/>
</MetadataDirectory>
```

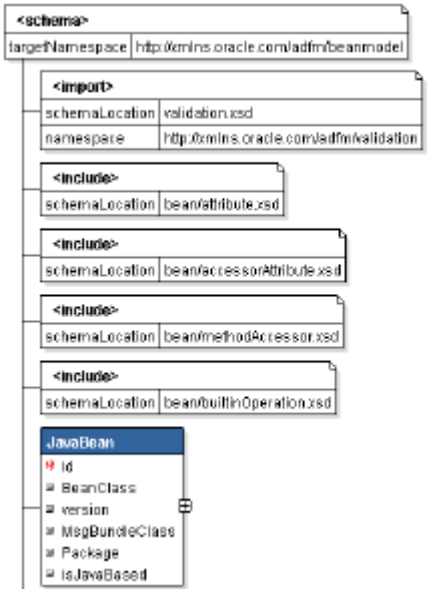
A.5 Structure Definition Files

Structure definition files are created to support a data control's structure, and the structure of read-only and updateable attributes and collections.

When you register a session bean as an Oracle ADF data control, an XML definition file is created in the Model project for every session bean. The structure definition file has the same name as the session bean, but has a .xml extension. A structure definition is also created for each EJB entity and TopLink POJO.

Figure A–5 shows the toplevel definition for the JavaBean structure definition.

Figure A–5 Schema Root for the Structure Definition of a JavaBean



A.5.1 Syntax for the Structure Definition for a JavaBean

The toplevel element of the structure definition is <JavaBean>:

```
<JavaBean xmlns="http://xmlns.oracle.com/adfm/beanmodel" version="10.1.3.35.83"
          id="<beanname>" BeanClass="oracle.srdemo.model.entities.<beanname>"
          Package="oracle.srdemo.model.entities" isJavaBased="true">
```

where the XML namespace attribute (xmlns) specifies the URI to which the structure definition binds at runtime. Only the package name is editable; all other attributes should have the values shown.

Figure A–6 displays the <Attribute> child element of the <JavaBean> element. It defines the structure definition of a bean attribute.

Figure A–6 Schema for Attribute

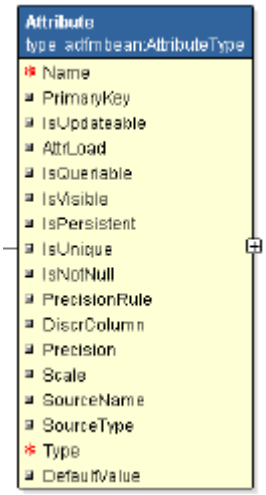


Figure A–7 displays the <AccessorMethod> child element of the <JavaBean> element. It defines the structure information for an attribute that returns an object.

Figure A–7 Schema for AccessorAttribute

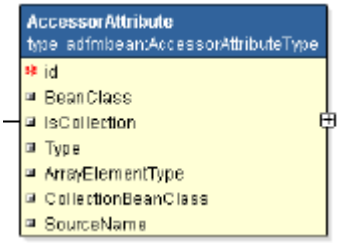


Figure A–8 displays the <MethodAccessor> child element of the <JavaBean> element. It defines the structure information for a method that returns an object or collection. Note that unlike attribute accessors, method accessors can define parameters.

Figure A–8 Schema for MethodAccessor

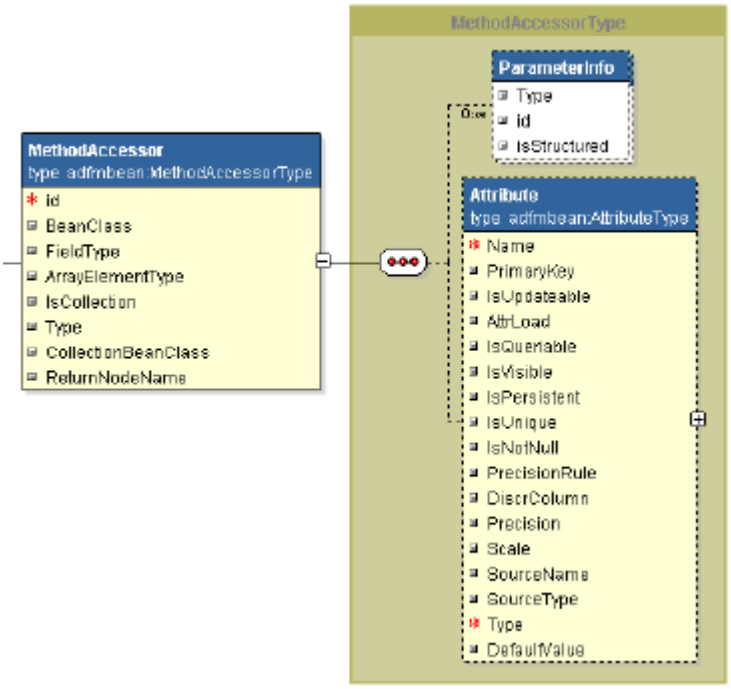


Table A–2 describes the attributes of the structure definition of the bean. Note that each attribute or method specified by the bean, will have its own <Attribute>, <AccessorAttribute>, and <MethodAccessor> definition and each method parameter will have its own <ParameterInfo> definition.

Table A–2 Attributes of the Structure Definition for a Bean

Element Syntax	Attributes	Attribute Description
<Attribute>	AttrLoad	Internal
	DefaultValue	This field is used only in case of variables and ADF Business Components datacontrol. For beans, since the beans themselves create a new bean instance, it is assumed that all properties are appropriately defaulted in the new instance.
	DiscrColumn	Internal.
	IsNotNull	Determines if the attribute is mandatory.
	IsPersistent	Determines if an attribute is persistent or transient
	IsQueriable	Determines if this attribute can participate in a WHERE clause
	IsUnique	Determines if the attribute is unique, and will have a UNIQUE constraint generated in the database.
	IsUpdateable	Determines if the attribute is always updateable, only updateable while new, or never updateable.
	IsVisible	.Determines if the attribute is visible or hidden.
	Name	The attributes name.
	PrecisionRule	Determines whether precision should be applied. If true, the precision rule is applied.

Table A–2 (Cont.) Attributes of the Structure Definition for a Bean

Element Syntax	Attributes	Attribute Description
	PrimaryKey	.Determines the attributes that participate in a primary key.
<AttributeAccessor>	ArrayElementType	This is only used for ADF Business Components domains, to define the array type.
	BeanClass	Fully qualified package name. Identifies the full path to the type's structure definition.
	CollectionBeanClass	Fully qualified package name. Identifies the full path to the method accessor's structure definition. In the case of a session bean that accesses an entity that is a collection, this value will be either ReadOnlyCollection or UpdateableCollection.
	id	Unique identifier. Same as the method name that appears in the bean class.
	IsCollection	Identifies whether the method accessor returns a collection. Set to true when the entity accessed is a collection and specify the CollectionBeanClass attribute value.
	SourceName	The name of the property on the source of this bean.
	BeanClass	Fully qualified package name. Identifies the full path to the type's structure definition.
	CollectionBeanClass	Fully qualified package name. Identifies the full path to the method accessor's structure definition. In the case of a session bean that accesses an entity that is a collection, this value will be either ReadOnlyCollection or UpdateableCollection.
	id	Unique identifier. Same as the method name that appears in the bean class.
	IsCollection	Identifies whether the method accessor returns a collection. Set to true when the entity accessed is a collection and specify the CollectionBeanClass attribute value.
	ReturnNodeName	Unique identifier. Name used in the Data Control Palette to display the return node.
<MethodAccessor> <ParameterInfo /> </MethodAccessor>	id	Unique identifier. Same as the method parameter name that appears in the method signature of the session bean class
	Type	Specifies the data type of the parameter

A.5.2 Sample Structure Definition for the <sessionbeanname>.xml File

Example A–3 shows the sample SRAdminFacade.xml file from the SRDemo application. Notice that the SRAdminFacade.xml file lists attributes, accessors, and operations. Notice that operations may have parameters which reference other structure definitions (which are in turn composed of attributes, accessors, and operations).

Example A–3 Structure Definition of SRAdminFacade.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<JavaBean xmlns="http://xmlns.oracle.com/adfm/beanmodel" version="10.1.3.35.83"
          id="SRAdminFacade" BeanClass="oracle.srdemo.model.SRAdminFacade"
          Package="oracle.srdemo.model" isJavaBased="true">
  <MethodAccessor IsCollection="false"
                  Type="oracle.srdemo.model.entities.ExpertiseArea"
                  BeanClass="oracle.srdemo.model.entities.ExpertiseArea"
                  id="createExpertiseArea"
                  ReturnNodeName="oracle.srdemo.model.entities.ExpertiseArea">
    <ParameterInfo id="product" Type="oracle.srdemo.model.entities.Product"
                   isStructured="true"/>
    <ParameterInfo id="user" Type="oracle.srdemo.model.entities.User"
                   isStructured="true"/>
    <ParameterInfo id="prodId" Type="java.lang.Integer" isStructured="false"/>
    <ParameterInfo id="userId" Type="java.lang.Integer" isStructured="false"/>
    <ParameterInfo id="expertiseLevel" Type="java.lang.String"
                   isStructured="false"/>
    <ParameterInfo id="notes" Type="java.lang.String" isStructured="false"/>
  </MethodAccessor>
  <MethodAccessor IsCollection="false"
                  Type="oracle.srdemo.model.entities.Product"
                  BeanClass="oracle.srdemo.model.entities.Product"
                  id="createProduct"
                  ReturnNodeName="oracle.srdemo.model.entities.Product">
    <ParameterInfo id="prodId" Type="java.lang.Integer" isStructured="false"/>
    <ParameterInfo id="name" Type="java.lang.String" isStructured="false"/>
    <ParameterInfo id="image" Type="java.lang.String" isStructured="false"/>
    <ParameterInfo id="description" Type="java.lang.String"
                   isStructured="false"/>
  </MethodAccessor>
  <MethodAccessor IsCollection="false" Type="oracle.srdemo.model.entities.User"
                  BeanClass="oracle.srdemo.model.entities.User" id="createUser"
                  ReturnNodeName="oracle.srdemo.model.entities.User">
    <ParameterInfo id="userId" Type="java.lang.Integer" isStructured="false"/>
    <ParameterInfo id="userRole" Type="java.lang.String" isStructured="false"/>
    <ParameterInfo id="email" Type="java.lang.String" isStructured="false"/>
    <ParameterInfo id="firstName" Type="java.lang.String" isStructured="false"/>
    <ParameterInfo id="lastName" Type="java.lang.String" isStructured="false"/>
    <ParameterInfo id="streetAddress" Type="java.lang.String"
                   isStructured="false"/>
    <ParameterInfo id="city" Type="java.lang.String" isStructured="false"/>
    <ParameterInfo id="stateProvince" Type="java.lang.String"
                   isStructured="false"/>
    <ParameterInfo id="postalCode" Type="java.lang.String"
                   isStructured="false"/>
    <ParameterInfo id="countryId" Type="java.lang.String" isStructured="false"/>
  </MethodAccessor>
  <MethodAccessor IsCollection="true" Type="oracle.srdemo.model.entities.User"
                  BeanClass="oracle.srdemo.model.entities.User"
                  id="findAllStaffWithOpenAssignments"
                  ReturnNodeName="oracle.srdemo.model.entities.User"
                  CollectionBeanClass="UpdateableCollection"/>
```

```
  <MethodAccessor IsCollection="true"
                  Type="oracle.srdemo.model.entities.ExpertiseArea"
                  BeanClass="oracle.srdemo.model.entities.ExpertiseArea"
                  id="findExpertiseByUserId"
                  ReturnNodeName="oracle.srdemo.model.entities.ExpertiseArea"
                  CollectionBeanClass="UpdateableCollection">
    <ParameterInfo id="userIdParam" Type="java.lang.Integer"
                   isStructured="false"/>
  </MethodAccessor>
  <MethodAccessor IsCollection="false" Type="java.lang.Object" id="mergeEntity"
                  ReturnNodeName="Return">
    <ParameterInfo id="entity" Type="java.lang.Object" isStructured="false"/>
  </MethodAccessor>
  <MethodAccessor IsCollection="false" Type="java.lang.Object"
                  id="persistEntity" ReturnNodeName="Return">
    <ParameterInfo id="entity" Type="java.lang.Object" isStructured="false"/>
  </MethodAccessor>
  <MethodAccessor IsCollection="false" Type="java.lang.Object"
                  id="refreshEntity" ReturnNodeName="Return">
    <ParameterInfo id="entity" Type="java.lang.Object" isStructured="false"/>
  </MethodAccessor>
  <MethodAccessor IsCollection="false" Type="void" id="removeEntity"
                  ReturnNodeName="Return">
    <ParameterInfo id="entity" Type="java.lang.Object" isStructured="false"/>
  </MethodAccessor>
</JavaBean>
```

A.5.3 Sample Structure Definition for the <entitybeanname>.xml File

The XML files that get created for a TopLink POJO, EJB entity or a Java bean are very similar, as the constructs listed in each case are generic ADF metadata. The following syntax shows a TopLink entity XML file.

Example A–4 shows the sample Product.xml file from the SRDemo application. Notice the structure of the file, that it is broken up into attributes, accessors, and operations (methods).

Example A–4 Structure Definition of Product.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<JavaBean xmlns="http://xmlns.oracle.com/adfm/beanmodel" version="10.1.3.35.65"
          id="Product" BeanClass="oracle.srdemo.model.entities.Product"
          Package="oracle.srdemo.model.entities" isJavaBased="true">
  <Attribute Name="description" Type="java.lang.String"/>
  <Attribute Name="image" Type="java.lang.String"/>
  <Attribute Name="name" Type="java.lang.String"/>
  <Attribute Name="prodId" Type="java.lang.Integer"/>
  <AccessorAttribute id="expertiseAreaCollection" IsCollection="true"
                     BeanClass="oracle.srdemo.model.entities.ExpertiseArea"
                     CollectionBeanClass="UpdateableCollection"/>
  <AccessorAttribute id="serviceRequestCollection" IsCollection="true"
                     BeanClass="oracle.srdemo.model.entities.ServiceRequest"
                     CollectionBeanClass="UpdateableCollection"/>
  <MethodAccessor IsCollection="false" Type="void" id="addExpertiseArea"
                  ReturnNodeName="Return">
    <ParameterInfo id="anExpertiseArea"
                   Type="oracle.srdemo.model.entities.ExpertiseArea"
                   isStructured="true"/>
  </MethodAccessor>
```

```
  <MethodAccessor IsCollection="false" Type="void" id="addExpertiseArea"
                  ReturnNodeName="Return">
    <ParameterInfo id="index" Type="int" isStructured="false"/>
    <ParameterInfo id="anExpertiseArea"
                   Type="oracle.srdemo.model.entities.ExpertiseArea"
                   isStructured="true"/>
  </MethodAccessor>
  <MethodAccessor IsCollection="false" Type="void" id="addServiceRequest"
                  ReturnNodeName="Return">
    <ParameterInfo id="aServiceRequest"
                   Type="oracle.srdemo.model.entities.ServiceRequest"
                   isStructured="true"/>
  </MethodAccessor>
  <MethodAccessor IsCollection="false" Type="void" id="addServiceRequest"
                  ReturnNodeName="Return">
    <ParameterInfo id="index" Type="int" isStructured="false"/>
    <ParameterInfo id="aServiceRequest"
                   Type="oracle.srdemo.model.entities.ServiceRequest"
                   isStructured="true"/>
  </MethodAccessor>
  <MethodAccessor IsCollection="false" Type="void" id="removeExpertiseArea"
                  ReturnNodeName="Return">
    <ParameterInfo id="anExpertiseArea"
                   Type="oracle.srdemo.model.entities.ExpertiseArea"
                   isStructured="true"/>
  </MethodAccessor>
  <MethodAccessor IsCollection="false" Type="void" id="removeServiceRequest"
                  ReturnNodeName="Return">
    <ParameterInfo id="aServiceRequest"
                   Type="oracle.srdemo.model.entities.ServiceRequest"
                   isStructured="true"/>
  </MethodAccessor>
  <ConstructorMethod IsCollection="false" Type="void" id="Product"/>
</JavaBean>
```

A.5.4 Collection and SingleValue Sample Files

Four additional files are also generated:

- ReadOnlyCollection.xml
- ReadOnlySingleValue.xml
- UpdateableCollection.xml
- UpdateableSingleValue.xml

These files support the Data Control Palette in JDeveloper. The files are used only at design time to specify the list of operations that the Data Control Palette may display for a given accessor. These files are referenced by the accessor's CollectionBeanClass attribute. Typically you do not edit these files, but if you wanted to remove an operation from the Palette, you could remove an item on this list.

Example A–2 shows a read-only collection. The syntax for all four design-time XML files is similar.

Example A–5 Read-only Collection Syntax

```
<?xml version="1.0" encoding="UTF-8" ?>
<JavaBean xmlns="http://xmlns.oracle.com/adfm/beanmodel" version="10.1.3.35.65"
          id="ReadOnlyCollection" BeanClass="ReadOnlyCollection"
          isJavaBased="false">
  <BuiltinOperation id="IteratorExecute"/>
  <BuiltinOperation id="Find"/>
  <BuiltinOperation id="First"/>
  <BuiltinOperation id="Last"/>
  <BuiltinOperation id="Next"/>
  <BuiltinOperation id="Previous"/>
  <BuiltinOperation id="NextSet"/>
  <BuiltinOperation id="PreviousSet"/>
  <BuiltinOperation id="setCurrentRowWithKey"/>
  <BuiltinOperation id="setCurrentRowWithKeyValue"/>
</JavaBean>
```

A.6 DataBindings.cpx

The `DataBindings.cpx` file is created in the user interface project the first time you drop a data control usage onto a web page in the HTML Visual Editor. The `.cpx` file defines the Oracle ADF binding context for the entire application and provides the metadata from which the Oracle ADF binding objects are created at runtime. When you insert a databound UI component into your document, the page will contain binding expressions that access the Oracle ADF binding objects at runtime.

If you are familiar with building ADF applications in earlier releases of JDeveloper, you'll notice that the `.cpx` file no longer contains all the information copied from the `DataControls.dcx` file, but only a reference to it. Therefore, if you need to make changes to the `.cpx` file, you must edit the `DataControls.dcx` file.

The `DataBindings.cpx` file appears in the `/src` directory of the user interface project folder. When you double-click the file node, the binding context description appears in the XML Source Editor. (To edit the binding context parameters, use the Property Inspector and select the desired parameter in the Structure window.)

A.6.1 DataBindings.cpx Syntax

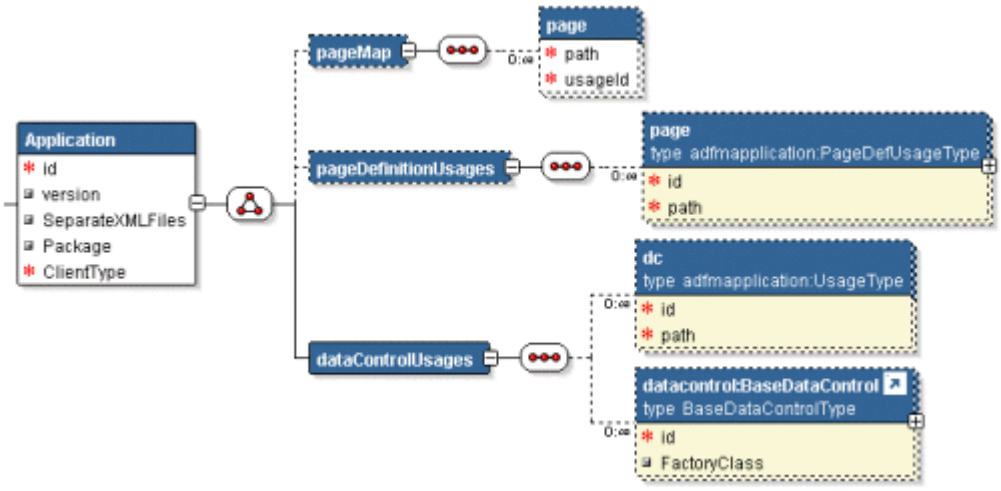
The toplevel element of the DataBindings.cpx file is <DataControlConfigs>:

```
<DataControlConfigs xmlns="http://xmlns.oracle.com/adfm/configuration"
                    version="10.1.3.35.65" Package="oracle.srdemo.model"
                    id="DataControls">
```

where the XML namespace attribute (xmlns) specifies the URI to which the data controls bind at runtime. Only the package name is editable; all other attributes should have the values shown.

Figure A–9 displays the child element hierarchy of the <DataControlConfigs> element. Note that each business service for which you have created a data control, will have its own <dataControlUsages> definition.

Figure A–9 Schema for the Structure Definition of the DataBindings.cpx File



The child elements have the following usages:

- <pageMap> element maps all user interface URLs and the corresponding pageDefinitionUsage name. This map is used at runtime to map an URL to its pageDefinition.
- <pageDefinitionUsages> element maps a PageDefinition Usage (BindingContainer instance) name to the corresponding pageDefinition definition. The id attribute represents the usage id. The path attribute represents the full path to the page definition.
- <dataControlUsages> element declares a list of datacontrol (shortnames) and corresponding path to the datacontrol definition entries in the dcx or xcfg file.

Table A–3 describes the attributes of the DataBindings.cpx elements.

Table A–3 Attributes of the DataBindings.cpx File Elements

Element Syntax	Attributes	Attribute Description

`<pageMap>`
`  <page  />`
`</pageMap>`
	path	The full directory path. Identifies the location of the user interface page.

	usageId	A unique qualifier. Names the page definition id that appears in the ADF page definition file. The ADF binding servlet looks at the incoming URL requests and checks that the bindings variable is pointing to the ADF page definition associated with the URL of the incoming HTTP request.

`<pageDefinitionUsages>`
`  <page/>`
`</pageDefinitionUsages>`
	id	A unique qualifier. References the page definition id that appears in the ADF page definition file.

	path	The fully qualified package name. Identifies the location of the user interface page's ADF page definition file.

`<dataControlUsages>`
`  <dc ...  />`
`</dataControlUsages>`
	id	A unique qualifier. Identifies the data control usage as is defined in the DataControls.dcx file.

	path	The fully qualified package name. Identifies the location of the data control

A.6.2 DataBindings.cpx Sample

Example A–6 shows the syntax for the DataBindings.cpx file in the SR Demo application. It uses two session facades data controls and a URL data control. In the following code, notice the references to the data controls within the DCX. For example, <dc id="SRDemoFAQ" path="oracle.srdemo.faq.SRDemoFAQ"/> finds "SRDemoFAQ" via the data control id within DataControls.dcx.

Example A–6 DataBindings.cpx Sample

```
<?xml version="1.0" encoding="UTF-8" ?>
<Application xmlns="http://xmlns.oracle.com/adfm/application"
             version="10.1.3.35.65" id="DataBindings" SeparateXMLFiles="false"
             Package="oracle.srdemo.view" ClientType="Generic">
  <pageMap>
    <page path="/app/SRList.jspx" usageId="SRListPageDef"/>
    <page path="/app/SRCreate.jspx" usageId="SRCreatePageDef"/>
    <page path="/app/SRCreateConfirm.jspx" usageId="SRCreateConfirmPageDef"/>
    <page path="/app/staff/SREdit.jspx" usageId="SREditPageDef"/>
    <page path="/app/staff/SRStaffSearch.jspx" usageId="SRStaffSearchPageDef"/>
    <page path="/app/staff/SRSearch.jspx" usageId="SRSearchPageDef"/>
    <page path="/app/SRMain.jspx" usageId="SRMainPageDef"/>
    <page path="/app/management/SRManage.jspx" usageId="SRManagePageDef"/>
    <page path="/app/SRFaq.jspx" usageId="SRFaqPageDef"/>
  </pageMap>
  <pageDefinitionUsages>
    <page id="SRListPageDef"
          path="oracle.srdemo.view.pageDefs.app_SRListPageDef"/>
    <page id="UserInfoPageDef"
          path="oracle.srdemo.view.pageDefs.headless_UserInfoPageDef"/>
    <page id="SRCreatePageDef"
          path="oracle.srdemo.view.pageDefs.app_SRCreatePageDef"/>
    <page id="SRCreateConfirmPageDef"
          path="oracle.srdemo.view.pageDefs.app_SRCreateConfirmPageDef"/>
    <page id="SREditPageDef"
          path="oracle.srdemo.view.pageDefs.app_staff_SREditPageDef"/>
    <page id="SRStaffSearchPageDef"
          path="oracle.srdemo.view.pageDefs.app_staff_SRStaffSearchPageDef"/>
    <page id="SRSearchPageDef"
          path="oracle.srdemo.view.pageDefs.app_staff_SRSearchPageDef"/>
    <page id="SRMainPageDef"
          path="oracle.srdemo.view.pageDefs.app_SRMainPageDef"/>
    <page id="SRManagePageDef"
          path="oracle.srdemo.view.pageDefs.app_management_SRManagePageDef"/>
    <page id="SRFaqPageDef"
          path="oracle.srdemo.view.pageDefs.app_SRFaqPageDef"/>
  </pageDefinitionUsages>
  <dataControlUsages>
    <dc id="SRDemoFAQ" path="oracle.srdemo.faq.SRDemoFAQ"/>
    <dc id="SRAdminFacade" path="oracle.srdemo.model.SRAdminFacade"/>
    <dc id="SRPublicFacade"
        path="oracle.srdemo.model.SRPublicFacade"/>
  </dataControlUsages>
</Application>
```

A.7 <pageName>PageDef.xml

The <pageName>PageDef.xml files are created each time you insert a databound component into a web page using the Data Control Palette or Structure window. These XML files define the Oracle ADF binding container for each web page in the application. The binding container provides access to the bindings within the page. Therefore, you will have one XML file for each databound web page.

> **Note:** You cannot rename the <pageName>PageDef.xml file in JDeveloper, but you can rename the file outside of JDeveloper in your MyWork/ViewController/src/view folder. If you do rename the <pageName>PageDef.xml file, you must also update the DataBindings.cpx file references for the id and path attributes in the <pageDefinitionUsages> element.

The PageDef.xml file appears in the /src/view directory of the ViewController project folder. The Application Navigator displays the file in the view package of the Application Sources folder. When you double-click the file node, the page description appears in the XML Source Editor. To edit the page description parameters, use the Property Inspector and select the desired parameter in the Structure window.

There are important differences in how the PageDefs are generated for methods that return a single-value and a collection, so these are listed separately below.

A.7.1 PageDef.xml Syntax

The toplevel element of the PageDef.xml file is <pageDefinition>:

```
<pageDefinition xmlns="http://xmlns.oracle.com/adfm/uimodel"
                version="10.1.3.35.83" id="<pagename>PageDef"
                Package="oracle.srdemo.view.pageDefs">
```

where the XML namespace attribute (xmlns) specifies the URI to which the ADF binding container binds at runtime. Only the package name is editable; all other attributes should have the values shown.

Example A–7 displays the child element hierarchy of the <pageDefinition> element. Note that each business service for which you have created a data control, will have its own <AdapterDataControl> definition.

Example A–7 PageDef.xml Element Hierarchy

```
<?xml version="1.0" encoding="UTF-8" ?>
<pageDefinition>
    <parameters>
        ...
    </parameters>
    <executables>
        ...
    </executables>
    <bindings>
        ...
    </bindings>
</pageDefinition>
```

The child elements have the following usages:

- <parameters> defines page-level parameters that are EL accessible. These parameters store information local to the web page request and may be accessed in the binding expressions.
- <executables> defines the list of items (methods and accessors) to execute during the prepareModel phase of the ADF page lifecycle. Methods to be executed are defined by <methodIterator>. The lifecycle performs the execute in the sequence listed in the <executables> section. Whether or not the method or operation is executed depends on it's refresh or refreshCondition attribute value. Built-in operations on the data control are defined by:

  <page> - definition for a nested page definition (binding container)

  <iterator> - definition to a named collection in DataControls

  <accessorIterator> - definition to get an accessor in a data control hierarchy

  <methodIterator> - definition to get to an iterator returned by an invoked method defined by a methodAction in the same file

  <variableIterator> - internal iterator that contains variables declared for the binding container

  <invokeAction> - definition of which method to invoke as an executable
- <bindings> refers to an entry in <executables> to get to the collection from which bindings extract/submit attribute level data.

Table A–4 describes the attributes of the toplevel <pageDefinition> element.

Table A–4 Attributes of the PageDef.xml File <pageDefinition> Element

Element Syntax	Attributes	Attribute Description
`<pageDefinition>`	ControllerClass	Fully qualified classname to create when controller requests a PageController object for this bindingContainer
	EnableTokenValidation	Enables currency validation for this bindingContainer when a postback occurs. This is to confirm that the web tier state matches the state that particular page was rendered with.
	FindMode	This is for legacy (10.1.2) use only and indicates if this bindingContainer should start out in findMode when initially prepared.
	MsgBundleClass	Fully qualified package name. Identifies the class which contains translation strings for any bindings
	Viewable	An EL expression that should resolve at runtime to whether this binding and the associated component should be rendered or not.

Table A–5 describes the attributes of the child element of <parameters>.

Table A–5 Attributes of the PageDef.xml File <parameters> Element

Element Syntax	Attributes	Attribute Description
`<parameter>`	id	Unique identifier. May be referenced by ADF bindings
	option	Indicates the usage of the variable within the binding container: ■ Final indicates that this parameter cannot be passed in by a usage of this binding container, it must use the default value in the definition. ■ Optional indicates the variable value need not be provided. ■ Mandatory indicates the variable value must be provided or a binding container exception will be thrown.
	readonly	Indicates whether the parameter value may be modified or not. Set to true when you do not want the application to modify the parameter value.
	value	A default value, this can be an EL expression.

Table A–6 describes the attributes of the PageDef.xml <executables> elements.

Table A–6 Attributes of the PageDef.xml File <executables> Element

Element Syntax	Attributes	Attribute Description
`<accessorIterator>`	BeanClass	Identifies the Java type of beans in the associated iterator/collection.
	CacheResults	If true, manage the data collection between requests.
	DataControl	The data control which interprets/returns the collection referred to by this iterator binding.
	id	Unique identifier. May be referenced by any ADF value binding.
	MasterBinding	Reference to the methodIterator (or iterator) that binds the data collection that serves as the master to the accessor iterator's detail collection.
	ObjectType	This is used for ADF Business Components only. A boolean value determines if the collection is an object type or not.
	RangeSize	Specifies the number of data objects in a range to fetch from the bound collection. The range defines a window you can use to access a subset of the data objects in the collection. By default, the range size is set to a range that fetches just ten data objects. Use RangeSize when you want to work with an entire set or when you want to limit the number of data objects to display in the page. Note that the values -1 and 0 have specific meaning: the value -1 returns all available objects from the collection, while the value 0 will return the same number of objects as the collection uses to retrieve from its data source.

Table A–6 (Cont.) Attributes of the PageDef.xml File <executables> Element

Element Syntax	Attributes	Attribute Description
	Refresh	Determines when and whether the executable should be invoked. Set one of the following properties as required: ■ always - causes the executable to be invoked each time the binding container is prepared. This will occur when the page is displayed and when the user submits changes, or when the application posts back to the page. ■ deferred - refresh occurs when another binding requires/refers to this executable.Since refreshing an executable may be a performance concern, you can set the refresh to only occur if it's used in a binding that is being rendered. ■ ifNeeded - (default) whenever the framework needs to refresh the executable because it has not been refreshed to this point. For example, when you have an accessor hierarchy such that a detail is listed first in the page definition, the master could be refreshed twice (once for the detail and again for the master's iterator). Using ifNeeded gives the mean to avoid duplicate refreshes. This is the default behavior for executables. ■ never - When the application itself will call refresh on the executable during one of the controller phases and does not want the framework to refresh it at all. ■ prepareModel - causes the executable to be invoked each time the page's binding container is prepared. ■ prepareModelIfNeeded - causes the executable to be invoked during the prepareModel phase if this executable has not been refreshed to this point. See also ifNeeded above. ■ renderModel - causes the executable to be invoked each time the page is rendered. ■ renderModelIfNeeded - causes the executable to be invoked during the page's renderModel phase on the condition that it is needed. See also ifNeeded above.
	RefreshCondition	An EL expression that when resolved, determines when and whether the executable should be invoked. For example, ${!bindings.findAllServiceRequestIter.findMode} resolves the value of the findMode on the iterator in the ADF binding context AllServiceRequest. Hint: Use the Property Inspector to create expressions from the available objects of the binding context (bindings namespace) or binding context (data namespace), JSF managed beans, and JSP objects.

Table A–6 (Cont.) Attributes of the PageDef.xml File <executables> Element

Element Syntax	Attributes	Attribute Description
`<invokeAction>`	Binds	Determines the action to invoke. This may be on any actionBinding. Additionally, in the case, of the EJB session facade data control, you may bind to the finder method exposed by the data control. Built-in actions supported by the EJB session facade data control include: ■ Execute executes the bound action defined by the data collection. ■ Find retrieves a data object from a collection. ■ First navigates to the first data object in the data collection range. ■ Last navigates to the first data object in the data collection range. ■ Next navigates to the first data object in the data collection range. If the current range position is already on the last data object, then no action is performed. ■ Previous navigates to the first data object in the data collection range. If the current position is already on the first data object, then no action is performed. ■ setCurrentRowWithKey passes the row key as a String converted from the value specified by the input field. The row key is used to set the currency of the data object in the bound data collection. When passing the key, the URL for the form will not display the row key value. You may use this operation when the data collection defines a multipart attribute key. ■ setCurrentRowWithKeyValue is used as above, but when you want to use a primary key value instead of the stringified key.
	id	Unique identifier. May be referenced by any ADF action binding
	Refresh	see Refresh above.
	RefreshCondition	see RefreshCondition above.
`<iterator>` and `<methodIterator>`	BeanClass	Identifies the Java type of beans in the associated iterator/collection
	BindingClass	This is for backward compatibility to indicate which class implements the runtime for this binding definition. Ignored in JDeveloper 10.1.3.
	Binds	see Binds above.
	CacheResults	see CacheResults above
	DataControl	Name of the DataControl usage in the bindingContext (.cpx) which this iterator is associated with.
	DefClass	Used internally for testing.
	id	Unique identifier. May be referenced by any ADF value binding.
	ObjectType	Not used by EJB session facade data control (used by ADF Business Components only).

Table A–6 (Cont.) Attributes of the PageDef.xml File <executables> Element

Element Syntax	Attributes	Attribute Description
	RangeSize	see RangeSize above
	Refresh	see Refresh above
	RefreshCondition	see RefreshCondition above
<page> and <variableIterator>	id	Unique identifier. In the case of <page>, refers to nested page/region that is included in this page. In the case of the <variableIterator> executable, the identifier may be referenced by any ADF value binding
	path	Used by <page> executable only. Advanced, a fully qualified path that may reference another page's binding container.
	Refresh	see Refresh above
	RefreshCondition	see RefreshCondition above

Table A–7 describes the attributes of the PageDef.xml <bindings> element.

Table A–7 Attributes of the PageDef.xml File <bindings> Elements

Element Syntax	Attributes	Attribute Description
<action>	Action	Fully qualified package name. Identifies the class for which the data control is created. In the case of the EJB session facade, this the session bean
	BindingClass	This is for backward compatibility to indicate which class implements the runtime for this binding definition. Ignored in JDeveloper 10.1.3.
	DataControl	Name of the DataControl usage in the bindingContext (.cpx) which this iteratorBinding or actionBinding is associated with.
<attributeValues>	ApplyValidation	Set to True by default. When true, controlBinding executes validators defined on the binding. You can set to False in the case of ADF Business Components, when running in local mode and the same validators are already defined on the corresponding attribute.
	BindingClass	This is for backward compatibility to indicate which class implements the runtime for this binding definition. Ignored in JDeveloper 10.1.3.
	ControlClass	Used internally for testing purposes.
	CustomInputHandler	This is the class name for a oracle.jbo.uicli.binding.JUCtrlValueHandler implementation that is used to process the inputValue for a given value binding.
	DefClass	Used internally for testing.
	id	Unique identifier. May be referenced by any ADF action binding
	IterBinding	Refers to the iteratorBinding instance in this bindingContainer to which this binding is associated.
	NullValueId	Refers to the entry in the message bundle for this bindingContainer that contains the String to indicate the null value in a list display.

Table A–7 (Cont.) Attributes of the PageDef.xml File <bindings> Elements

Element Syntax	Attributes	Attribute Description
<button>	ApplyValidation	Set to True by default. When true, controlBinding executes validators defined on the binding. You can set to False in the case of ADF Business Components, when running in local mode and the same validators are already defined on the corresponding attribute.
	BindingClass	This is for backward compatibility to indicate which class implements the runtime for this binding definition. Ignored in JDeveloper 10.1.3.
	BoolVal	Identifies whether the value at the zero index in the static value list in this boolean list binding represents true or false.
	ControlClass	Used internally for testing purposes.
	CustomInputHandler	This is the class name for a oracle.jbo.uicli.binding.JUCtrlValueHandler implementation that is used to process the inputValue for a given value binding.
	DefClass	Used internally for testing.
	id	Unique identifier. May be referenced by any ADF action binding
	IterBinding	Refers to the iteratorBinding instance in this bindingContainer to which this binding is associated.
	ListIter	Refers to the iteratorBinding that is associated with the source list of this listBinding.
	ListOperMode	Determines if this list binding is for navigation, contains a static list of values or is a LOV type list.
	NullValueFlag	Describes whether this list binding has a null value and if so, whether it should be displayed at the beginning of the list or the end.
	NullValueId	Refers to the entry in the message bundle for this bindingContainer that contains the String to indicate the null value in a list display.
<graph>	ApplyValidation	Set to True by default. When true, controlBinding executes validators defined on the binding. You can set to False in the case of ADF Business Components, when running in local mode and the same validators are already defined on the corresponding attribute.
	BindingClass	This is for backward compatibility to indicate which class implements the runtime for this binding definition. Ignored in JDeveloper 10.1.3.
	BoolVal	Identifies whether the value at the zero index in the static value list in this boolean list binding represents true or false.
	ChildAccessorName	The name of the accessor to invoke to get the next level of nodes for a given Hierarchical Node Type in a tree.
	ControlClass	Used internally for testing purposes.
	CustomInputHandler	This is the class name for a oracle.jbo.uicli.binding.JUCtrlValueHandler implementation that is used to process the inputValue for a given value binding.

Table A–7 (Cont.) Attributes of the PageDef.xml File <bindings> Elements

Element Syntax	Attributes	Attribute Description
	DefClass	Used internally for testing.
	id	Unique identifier. May be referenced by any ADF action binding
	IterBinding	Refers to the iteratorBinding instance in this bindingContainer to which this binding is associated.
	NullValueId	Refers to the entry in the message bundle for this bindingContainer that contains the String to indicate the null value in a list display.
`<list>`	ApplyValidation	Set to True by default. When true, controlBinding executes validators defined on the binding. You can set to False in the case of ADF Business Components, when running in local mode and the same validators are already defined on the corresponding attribute.
	BindingClass	This is for backward compatibility to indicate which class implements the runtime for this binding definition. Ignored in JDeveloper 10.1.3.
	ControlClass	Used internally for testing purposes.
	CustomInputHandler	This is the class name for a oracle.jbo.uicli.binding.JUCtrlValueHandler implementation that is used to process the inputValue for a given value binding.
	DefClass	Used internally for testing.
	id	Unique identifier. May be referenced by any ADF action binding
	IterBinding	Refers to the iteratorBinding instance in this bindingContainer to which this binding is associated.
	ListIter	Refers to the iteratorBinding that is associated with the source list of this listBinding.
	ListOperMode	Determines if this list binding is for navigation, contains a static list of values or is a LOV type list.
	NullValueFlag	Describes whether this list binding has a null value and if so, whether it should be displayed at the beginning of the list or the end.
	NullValueId	Refers to the entry in the message bundle for this bindingContainer that contains the String to indicate the null value in a list display.
	StaticList	Defines a static list of values that will be rendered in the bound list component.
`<methodAction>`	Action	Fully qualified package name. Identifies the class for which the data control is created. In the case of the EJB session facade, this the session bean
	BindingClass	This is for backward compatibility to indicate which class implements the runtime for this binding definition. Ignored in JDeveloper 10.1.3.
	ClassName	This is the class to which the method being invoked belongs.

Table A–7 (Cont.) Attributes of the PageDef.xml File <bindings> Elements

Element Syntax	Attributes	Attribute Description
	CustomInputHandler	This is the class name for a oracle.jbo.uicli.binding.JUCtrlValueHandler implementation that is used to process the inputValue for a given value binding.
	DataControl	Name of the DataControl usage in the bindingContext (.cpx) which this iteratorBinding or actionBinding is associated with.
	DefClass	Used internally for testing.
	id	Unique identifier. May be referenced by any ADF action binding
	InstanceName	A dot-separated EL path to a Java object instance on which the associated method is to be invoked.
	IsLocalObjectReference	Set to True if the instanceName contains an EL path relative to this bindingContainer.
	IterBinding	Refers to the iteratorBinding instance in this bindingContainer to which this binding is associated.
	MethodName	Indicates the name of the operation on the given instance/class that needs to be invoked for this methodActionBinding.
	RequiresUpdateModel	Whether this action requires that the model be updated before the action is to be invoked.
	ReturnName	The EL path of the result returned by the associated method.
`<table>` and `<tree>`	ApplyValidation	Set to True by default. When true, controlBinding executes validators defined on the binding. You can set to False in the case of ADF Business Components, when running in local mode and the same validators are already defined on the corresponding attribute.
	BindingClass	This is for backward compatibility to indicate which class implements the runtime for this binding definition. Ignored in JDeveloper 10.1.3.
	ControlClass	Used internally for testing purposes.
	CustomInputHandler	This is the class name for a oracle.jbo.uicli.binding.JUCtrlValueHandler implementation that is used to process the inputValue for a given value binding.
	DefClass	Used internally for testing.
	DiscrValue	Indicates the discriminator value for a hierarchical type binding (type definition for a tree node). This value is used to determine if a given row in a collection being rendered in a polymorphic tree binding should be rendered using the containing hierarchical type binding.
	id	Unique identifier. May be referenced by any ADF action binding
	IterBinding	Refers to the iteratorBinding instance in this bindingContainer to which this binding is associated.
	NullValueId	Refers to the entry in the message bundle for this bindingContainer that contains the String to indicate the null value in a list display.

A.7.2 PageDef.xml Sample for a Method That Returns a String

This is the page definition file that's created when you drop the method return User from the method findUserByEmail() from the Data Control Palette, SRPublicFacade node, into your open JSP page.

Example A–8 PageDef for findUserByEmail()

```
<?xml version="1.0" encoding="UTF-8" ?>
<pageDefinition xmlns="http://xmlns.oracle.com/adfm/uimodel"
                version="10.1.3.36.7" id="untitled1PageDef"
                Package="project1.pageDefs">
  <parameters/>
  <executables>
    <variableIterator id="variables"/>
    <methodIterator id="findUserByEmailIter" Binds="findUserByEmail.result"
                    DataControl="SRPublicFacade" RangeSize="10"
                    BeanClass="oracle.srdemo.model.entities.User"/>
  </executables>
  <bindings>
    <methodAction id="findUserByEmail"
                  InstanceName="SRPublicFacade.dataProvider"
                  DataControl="SRPublicFacade" MethodName="findUserByEmail"
                  RequiresUpdateModel="true" Action="999"
                  ReturnName="SRPublicFacade.methodResults.SRPublicFacade_
dataProvider_findUserByEmail_result">
      <NamedData NDName="emailParam" NDValue="mkorf@oracle.com"
                 NDType="java.lang.String"/>
    </methodAction>
    <table id="findUserByEmail1" IterBinding="findUserByEmailIter">
      <AttrNames>
        <Item Value="city"/>
        <Item Value="countryId"/>
        <Item Value="email"/>
        <Item Value="firstName"/>
        <Item Value="lastName"/>
        <Item Value="postalCode"/>
        <Item Value="stateProvince"/>
        <Item Value="streetAddress"/>
        <Item Value="userId"/>
        <Item Value="userRole"/>
      </AttrNames>
    </table>
  </bindings>
</pageDefinition>
```

A.7.3 PageDef.xml Sample for a Method that Returns a Collection

This is the page definition file that's created when you drop the User node from the findAllStaff() method in the Data Control Palette, SRPublicFacade node, into your open JSP page. This one is a collection.

Example A–9 PageDef for Method that Returns a Collection

```
<?xml version="1.0" encoding="UTF-8" ?>
<pageDefinition xmlns="http://xmlns.oracle.com/adfm/uimodel"
                version="10.1.3.36.7" id="untitled2PageDef"
                Package="project1.pageDefs">
  <parameters/>
```

```
  <executables>
    <methodIterator id="findAllStaffIter" Binds="findAllStaff.result"
                    DataControl="SRPublicFacade" RangeSize="10"
                    BeanClass="oracle.srdemo.model.entities.User"/>
  </executables>
  <bindings>
    <methodAction id="findAllStaff" InstanceName="SRPublicFacade.dataProvider"
                  DataControl="SRPublicFacade" MethodName="findAllStaff"
                  RequiresUpdateModel="true" Action="999"
                  ReturnName="SRPublicFacade.methodResults.SRPublicFacade_
dataProvider_findAllStaff_result"/>
    <table id="findAllStaff1" IterBinding="findAllStaffIter">
      <AttrNames>
        <Item Value="city"/>
        <Item Value="countryId"/>
        <Item Value="email"/>
        <Item Value="firstName"/>
        <Item Value="lastName"/>
        <Item Value="postalCode"/>
        <Item Value="stateProvince"/>
        <Item Value="streetAddress"/>
        <Item Value="userId"/>
        <Item Value="userRole"/>
      </AttrNames>
    </table>
  </bindings>
</pageDefinition>
```

A.8 web.xml

This section describes Oracle ADF configuration settings specific to the standard `web.xml` deployment descriptor file.

In JDeveloper when you create a project that uses JSF technology, a starter `web.xml` file with default settings is created for you in `/WEB-INF`. To edit the file, double-click **web.xml** in the Application Navigator to open it in the XML editor.

The following must be configured in `web.xml` for all applications that use JSF and ADF Faces:

- JSF servlet and mapping—The servlet `javax.faces.webapp.FacesServlet` that manages the request processing lifecycle for web applications utilizing JSF to construct the user interface.
- ADF Faces filter and mapping—A servlet filter to ensure that ADF Faces is properly initialized by establishing a `AdfFacesContext` object. This filter also processes file uploads.
- ADF resource servlet and mapping—A servlet to serve up web application resources (images, style sheets, JavaScript libraries) by delegating to a `ResourceLoader`.

The JSF servlet and mapping configuration settings are automatically added to the starter `web.xml` file when you first create a JSF project. When you insert an ADF Faces component into a JSF page for the first time, JDeveloper automatically inserts the configuration settings for ADF Faces filter and mapping, and resource servlet and mapping.

Example A–10 Configuring web.xml for ADF Faces and JSF

```
<?xml version = '1.0' encoding = 'windows-1252'?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee">
  <description>Empty web.xml file for Web Application</description>

  <!-- Installs the ADF Faces Filter -- >
  <filter>
    <filter-name>adfFaces</filter-name>
    <filter-class>oracle.adf.view.faces.webapp.AdfFacesFilter</filter-class>
  </filter>

  <!-- Adds the mapping to ADF Faces Filter -- >
  <filter-mapping>
    <filter-name>adfFaces</filter-name>
    <servlet-name>Faces Servlet</servlet-name>
  </filter-mapping>

  <!-- Maps the JSF servlet to a symbolic name -->
  <servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <!-- Maps ADF Faces ResourceServlet to a symbolic name -- >
  <servlet>
    <servlet-name>resources</servlet-name>
    <servlet-class>oracle.adf.view.faces.webapp.ResourceServlet</servlet-class>
  </servlet>

  <!-- Maps URL pattern to the JSF servlet's symbolic name -->
  <!-- You can use either a path prefix or extension suffix pattern -->
  <servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>/faces/*</url-pattern>
  </servlet-mapping>

  <!-- Maps URL pattern to the ResourceServlet's symbolic name -->
  <servlet-mapping>
    <servlet-name>resources</servlet-name>
    <url-pattern>/adf/*</url-pattern>
  </servlet-mapping>
...
</web-app>
```

Appendix A.8.1.1, "Configuring for State Saving" through Appendix A.8.1.7, "What You May Need to Know" detail the context parameters you could use in `web.xml` when you work with JSF and ADF Faces.

A.8.1 Tasks Supported by the web.xml File

The following JSF and ADF Faces tasks are supported by the `web.xml` file.

A.8.1.1 Configuring for State Saving

You can specify the following state-saving context parameters:

- `javax.faces.STATE_SAVING_METHOD`—Specifies where to store the application's view state. By default this value is `server`, which stores the application's view state on the server. If you wish to store the view state on the browser client, set this value to `client`. JDeveloper then automatically uses token-based client-side state saving (see `oracle.adf.view.faces.CLIENT_STATE_METHOD` below). You can specify the number of tokens to use instead of using the default number of 15 (see `oracle.adf.view.faces.CLIENT_STATE_MAX_TOKENS` below).
- `oracle.adf.view.faces.CLIENT_STATE_METHOD`—Specifies the type of client-side state saving to be used when client-side state saving is enabled. The values are:
  - `token`—(Default) Stores the page state in the session, but persists a token to the client. The simple token, which identifies a block of state stored back on the `HttpSession, is stored on the client`. This enables ADF Faces to disambiguate multiple appearances of the same page. Failover `HttpSession` is supported. This matches the default server-side behavior that will be provided in JSF 1.2.
  - `all`—Stores all state on the client in a (potentially large) hidden form field. This matches the client-side state saving behavior in JSF 1.1, but it is useful for developers who do not want to use `HttpSession`.
- `oracle.adf.view.faces.CLIENT_STATE_MAX_TOKENS`—Specifies how many tokens should be stored at any one time per user. The default is 15. When this value is exceeded, the state is lost for the least recently viewed pages, which affects users who actively use the **Back** button or who have multiple windows opened at the same time. If you're building HTML applications that rely heavily on frames, you would want to increase this value.

Example A–11 shows part of a `web.xml` file that contains state-saving parameters.

Example A–11 Context Parameters for State Saving in web.xml

```
<context-param>
  <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
  <param-value>client</param-value>
</context-param>

<context-param>
  <param-name>oracle.adf.view.faces.CLIENT_STATE_MAX_TOKENS</param-name>
  <param-value>20</param-value>
</context-param>
```

A.8.1.2 Configuring for Application View Caching

You can specify the following application view caching context parameter:

- `oracle.adf.view.faces.USE_APPLICATION_VIEW_CACHE`—Specifies whether to enable the application view caching feature. When application view caching is enabled, the first time a page is viewed by any user, ADF Faces caches the initial page state at an application level. Subsequently, all users can reuse the page's cached state coming and going, significantly improving application performance. Default is `false`.

Example A–12 shows part of a `web.xml` file that contains the application view caching parameter.

Example A–12 Context Parameters for Application View Caching in web.xml

```
<context-param>
  <param-name>oracle.adf.view.faces.USE_APPLICATION_VIEW_CACHE</param-name>
  <param-value>true</param-value>
</context-param>
```

A.8.1.3 Configuring for Debugging

You can specify the following debugging context parameters:

- `oracle.adf.view.faces.DEBUG_JAVASCRIPT`—ADF Faces by default obfuscates the JavaScript it delivers to the client, as well as strip comments and whitespace. This dramatically reduces the size of the ADF Faces JavaScript download, but also makes it tricky to debug the JavaScript. Set to `true` to turn off the obfuscation during application development. Set to `false` for application deployment.
- `oracle.adf.view.faces.CHECK_FILE_MODIFICATION`—By default this parameter is `false`. If it is set to `true`, ADF Faces will automatically check the modification date of your JSPs, and discard saved state when they change. When set to `true`, this makes development easier, but adds overhead that should be avoided when your application is deployed. Set to `false` for application deployment.

  For testing and debugging in JDeveloper's embedded OC4J, you don't need to explicitly set this parameter to `true` because ADF Faces automatically detects the embedded OC4J and runs with the file modification checks enabled.

Example A–13 shows part of a `web.xml` file that contains debugging parameters.

Example A–13 Context Parameters for Debugging in web.xml

```
<context-param>
  <param-name>oracle.adf.view.faces.DEBUG_JAVASCRIPT</param-name>
  <param-value>true</param-value>
</context-param>

<context-param>
  <param-name>oracle.adf.view.faces.CHECK_FILE_MODIFICATION</param-name>
  <param-value>true</param-value>
</context-param>
```

A.8.1.4 Configuring for File Uploading

You can specify the following file upload context parameters:

- `oracle.adf.view.faces.UPLOAD_TEMP_DIR`—Specifies the directory where temporary files are to be stored during file uploading. The default is the user's temporary directory.
- `oracle.adf.view.faces.UPLOAD_MAX_DISK_SPACE`—Specifies the maximum amount of disk space that can be used in a single request to store uploaded files. The default is 2000K.
- `oracle.adf.view.faces.UPLOAD_MAX_MEMORY`—Specifies the maximum amount of memory that can be used in a single request to store uploaded files. The default is 100K.

Example A–14 shows part of a `web.xml` file that contains file upload parameters.

Example A–14 Context Parameters for File Uploading in web.xml

```
<context-param>
  <param-name>oracle.adf.view.faces.UPLOAD_TEMP_DIR</param-name>
  <param-value>/tmp/Adfuploads</param-value>
</context-param>

<context-param>
  <param-name>oracle.adf.view.faces.UPLOAD_MAX_DISK_SPACE</param-name>
  <param-value>5120000</param-value>
</context-param>

<context-param>
  <param-name>oracle.adf.view.faces.UPLOAD_MAX_MEMORY</param-name>
  <param-value>512000</param-value>
</context-param>
```

> **Note:** The file upload initialization parameters are processed by the default `UploadedFileProcessor` only. If you replace the default processor with a custom `UploadedFileProcessor` implementation, the parameters are not processed.

For information about file uploading, see Section 11.6, "Providing File Upload Capability".

A.8.1.5 Configuring for ADF Model Binding

When you use ADF data controls to build web pages, the following must be configured in `web.xml`:

- ADF binding filter—A servlet filter to create the `ADFContext`, which contains context information about ADF, including the security context and the environment class that contains the request and response object. ADF applications use this filter to preprocess any HTTP requests that may require access to the binding context.
- Servlet context parameter for the application binding container—Specifies which CPX file the filter reads at runtime to define the application binding context. For information about CPX files, see Section 5.3, "Working with the DataBindings.cpx File".

In JDeveloper when you first use the Data Control Palette to build your databound JSF page, the ADF data binding configuration settings are automatically added to the `web.xml` file.

Example A–15 shows part of a `web.xml` file that contains ADF Model binding settings. For more information about the Data Control Palette and binding objects, see Chapter 5, "Displaying Data on a Page".

Example A–15 ADF Model Binding Configuration Settings in web.xml

```
<context-param>
  <param-name>CpxFileName</param-name>
  <param-value>view.DataBindings</param-value>
</context-param>

<filter>
  <filter-name>adfBindings</filter-name>
  <filter-class>oracle.adf.model.servlet.ADFBindingFilter</filter-class>
</filter>

<filter-mapping>
  <filter-name>adfBindings</filter-name>
  <url-pattern>*.jsp</url-pattern>
</filter-mapping>

<filter-mapping>
  <filter-name>adfBindings</filter-name>
  <url-pattern>*.jspx</url-pattern>
</filter-mapping>
```

A.8.1.6 Other Context Configuration Parameters for JSF

Other optional, application-wide parameters for JSF are:

- `javax.faces.CONFIG_FILES`—Specifies paths to JSF application configuration resource files. Use a comma-separated list of application-context relative paths for the value (see Example A–16). You need to set this parameter if you use more than one JSF configuration file in your application, as described in Appendix A.10.1, "Tasks Supported by the faces-config.xml".
- `javax.faces.DEFAULT_SUFFIX`—Specifies a file extension (suffix) for JSP pages that contain JSF components. The default value is `.jsp`.
- `javax.faces.LIFECYCLE_ID`—Specifies a lifecycle identifier other than the default set by the `javax.faces.lifecycle.LifecycleFactory.DEFAULT_LIFECYCLE` constant.

Example A–16 Configuring for Multiple JSF Configuration Files in web.xml

```
<context-param>
  <param-name>javax.faces.CONFIG_FILES</param-name>
  <param-value>/WEB-INF/faces-config1.xml,/WEB-INF/faces-config2.xml</param-value>
</context-param>
```

A.8.1.7 What You May Need to Know

If you have multiple filters for your application, make sure they are listed in `web.xml` in the order in which you want to run them. At runtime, the filters are called in the sequence listed in that file.

A.9 j2ee-logging.xml

ADF Faces leverages the Java Logging API (`java.util.logging.Logger`) to provide logging functionality when you run a debugging session. Java Logging is a standard API that is available in the Java Platform, starting with JDK 1.4. For the key elements, see the section "Java Logging Overview" at `http://java.sun.com/j2se/1.4.2/docs/guide/util/logging/overview.html`.

Typically you would want to configure the following in `j2ee-logging.xml`:

- Change the logging level for Oracle ADF packages. See Section A.9.1.1.
- Redirect the log output to a location, like a file, in addition to the default Log window in JDeveloper. See Section A.9.1.2.
- Change the directory path that determines where your log file resides. See Section A.9.1.3.

A.9.1 Tasks Supported by the j2ee-logging.xml

The following JSF tasks are supported by the `j2ee-logging.xml` file.

A.9.1.1 Change the Logging Level for Oracle ADF Packages

When you want to change the logging level of individual Oracle ADF packages, edit `<logger>` in the `<loggers>` element of `j2ee-logging.xml` (see Example A–17). The default level of logging is `INFO`. Oracle recommends `level="FINE"` for detailed log messages. Note that package names are hierarchically inclusive. For instance, if you change the level of `oracle.adf`, the level specified will also apply to all classes that begin with the path `oracle.adf`. To change the level of specific classes, supply the full path; for instance, a level set for the package name `oracle.adf.controller` will not apply to other branches of the `oracle.adf` package.

For details about setting logging when debugging ADF applications, see Section 16.4.2, "Creating an Oracle ADF Debugging Configuration".

Example A–17 Changing the Logging Level in j2ee-logging.xml

```
<loggers>
   <logger name="oracle.adf" level="FINE" />
   ...
</loggers>
```

A.9.1.2 Redirect the Log Output

The default logger (`name="oracle"`) is associated with two handlers: one for file output and another for console output (JDeveloper Log window). By default log messages are output to both locations at the same time. When you want to redirect the output for the log messages, edit `<handler>` in the `<logger>` element of `j2ee-logging.xml` (see Example A–18). For example, you can comment out the `<handler name="oc4j-handler"/>` when you want the output to only go to the JDeveloper Log window.

Example A–18 Changing the Logger Handler in j2ee-logging.xml

```
<loggers>
  <logger name="oracle" level="NOTIFICATION:1" useParentHandlers="false">
     <handler name="oc4j-handler"/>
     <handler name="console-handler"/>
  </logger>
  ...
</loggers>
```

A.9.1.3 Change the Location of the Log File

When you want to change where the log files reside, edit `<log_handler>` in the `<log_handlers>` element of `j2ee-logging.xml` (see Example A–19). The default directory for the log file is `../log/oc4j`.

Example A–19 Changing the Location of the Log File in j2ee-logging.xml

```
<log_handler name="oc4j-handler"
     class="oracle.core.ojdl.loggin.ODLHandlerFactory">
  <property name="path" value="C:/temp/adf-log"/>
  <property name-"maxFileSize" value="10485760"/>
```

A.10 faces-config.xml

You register a JSF application's resources—such as validators, converters, managed beans, and the application navigation rules—in the application's configuration file. Typically you have one configuration file named `faces-config.xml`.

> **Note:** A JSF application can have more than one JSF configuration file. For example if you need individual JSF configuration files for separate areas of your application, or if you choose to package libraries containing custom components or renderers, you can create a separate JSF configuration file for each area or library. For details see, Section 4.2.3, "What You May Need to Know About Multiple JSF Configuration Files".

In JDeveloper, when you create a project that uses JSF technology, an empty `faces-config.xml` file is created for you in `/WEB-INF`.

Typically you would want to configure the following in `faces-config.xml`:

- Application resources such as default render kit, message bundles, and supported locales. Refer to Section A.10.1.1, Section A.10.1.3. and Section A.10.1.4.
- Page-to-page navigation rules. See Section A.10.1.5.
- Custom validators and converters. See Section A.10.1.6.
- Managed beans for holding and processing data, handling UI events, and performing business logic.

If you use ADF data controls to build databound web pages, you also need to register the ADF phase listener in `faces-config.xml`. Refer to Section A.10.1.2.

A.10.1 Tasks Supported by the faces-config.xml

The following JSF tasks are supported by the `faces-config.xml` file.

A.10.1.1 Registering a Render Kit for ADF Faces Components

When you use ADF Faces components in your application, you must add the ADF default render kit in the `<application>` element of `faces-config.xml`. As mentioned earlier, JDeveloper creates one empty `faces-config.xml` file for you when you create a new project that uses JSF technology. When you insert an ADF Faces component into a JSF page for the first time, JDeveloper automatically inserts the default render kit for ADF components into `faces-config.xml` (see Example A–20).

Example A–20 Configuring faces-config.xml for ADF Faces Components

```
<?xml version="1.0" encoding="windows-1252"?>
<!DOCTYPE faces-config PUBLIC
  "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.1//EN"
  "http://java.sun.com/dtd/web-facesconfig_1_1.dtd">
<faces-config xmlns="http://java.sun.com/JSF/Configuration">
  ...
  <!-- Default render kit for ADF components -->
  <application>
    <default-render-kit-id>oracle.adf.core</default-render-kit-id>
  </application>
  ...
</faces-config>
```

A.10.1.2 Registering a Phase Listener for ADF Binding

The ADF phase listener is used to execute the ADF page lifecycle. When you use ADF data binding, you need to specify a phase listener for ADF lifecycle phases. In JDeveloper when an ADF data control is inserted into a JSF page for the first time, a standard ADF phase listener is added to `faces-config.xml` in the `<lifecycle>` element.

The ADF phase listener listens for all the JSF phases before which and after which it needs to execute its own phases concerned with preparing the model, validating model updates, and preparing pages to be rendered. See Section 6.2.3, "What Happens at Runtime: The JSF and ADF Lifecycles", for more information about how the ADF lifecycle phases integrate with the JSF lifecycle phases. Example A–21 shows part of a `faces-config.xml` that contains the ADF phase listener.

You may want to subclass the standard ADF phase listener when custom behavior, such as error handling, is desired. See Section 12.8, "Handling and Displaying Exceptions in an ADF Application" for details about subclassing the ADF phase listener. JDeveloper will not read the standard phase listener to `faces-config.xml` if it detects a subclass.

Example A–21 Registering the ADF Phase Listener in faces-config.xml

```
<faces-config xmlns="http://java.sun.com/JSF/Configuration">
  ...
<lifecycle>
  <phase-listener>
    oracle.adf.controller.faces.lifecycle.ADFPhaseListener
  </phase-listener>
</lifecycle>
  ...
</faces-config>
```

A.10.1.3 Registering a Message Resource Bundle

When you use a resource bundle for localized labels and messages, add the resource as a `<message-bundle>` in the `<application>` element of `faces-config.xml` (see Example A–22). The SRDemo application uses a resource properties file to hold the strings for the UI.

Example A–22 Registering a Message Bundle in faces-config.xml

```
<faces-config xmlns="http://java.sun.com/JSF/Configuration">
  ...
  <application>
    ...
    <message-bundle>oracle.srdemo.view.resources.UIResources</message-bundle>
    ...
  </application>
  ...
</faces-config>
```

To reference a message bundle in a page, see Section 14.4, "Internationalizing Your Application".

A.10.1.4 Configuring for Supported Locales

Register the default and all supported locales for your application in the `<application>` element of `faces-config.xml` (see Example A–23).

Example A–23 Registering Default and Supported Locales in faces-config.xml

```
<faces-config xmlns="http://java.sun.com/JSF/Configuration">
  ...
  <application>
    ...
    <locale-config>
      <default-locale>en</default-locale>
      <supported-locale>en-US</supported-locale>
      <supported-locale>es</supported-locale>
      <supported-locale>fr</supported-locale>
    </locale-config>
  </application>
  ...
</faces-config>
```

A.10.1.4.1 What You May Need to Know

JSF allows more than one `<application>` element in a single `faces-config.xml` file. The JSF Configuration Editor only allows you to edit the first instance in the file. You'll need to edit the file directly using the XML editor for any other `<application>` elements.

A.10.1.5 Creating Navigation Rules and Cases

While you can enter navigation rules and cases directly in the `faces-config.xml` file, Oracle recommends you use the JSF Navigation Modeler. The Navigation Modeler enables you to lay out the pages in your JSF application and add navigation between the pages in the form of a diagram. To open the Navigation Modeler, double-click the `faces-config.xml` file in the Application Navigator. In the visual editor, activate the **Diagram** tab to display the Navigation Modeler.

When JDeveloper first creates an empty `faces-config.xml`, it also creates a diagram file to hold diagram details such as layout and annotations. JDeveloper always maintains this diagram file alongside the `faces-config.xml` file, which holds all the settings needed by your application. This means that if you are using versioning or source control, the diagram file is included along with the `faces-config.xml` file it represents.

The navigation cases you add to the diagram are reflected in `faces-config.xml`, without your needing to edit the file directly.

A navigation rule defines one or more cases that specify an outcome value. A navigation component in a web page specifies an outcome value in its `action` attribute, which triggers a specific navigation case when a user clicks that component. For example, in the SRList page of the sample application, when the user clicks the **View** button, the application displays the SRMain page. The `action` attribute on the **View** button has the string value `View` (see Example A–24). The corresponding code for the navigation case within the navigation rule for the SRList page is shown in Example A–25.

Example A–24 Action Outcome String Defined on View Button

```
<af:commandButton text="#{res['srlist.buttonbar.view']}"
                  action="View"/>
```

Example A–25 Creating Static Navigation Cases in faces-config.xml

```
<faces-config xmlns="http://java.sun.com/JSF/Configuration">
  ...
  <navigation-rule>
    <from-view-id>/SRList.jspx</from-view-id>

    <navigation-case>
      <from-outcome>Edit</from-outcome>
      <to-view-id>/SREdit.jspx</to-view-id>
    </navigation-case>

    <navigation-case>
      <from-outcome>View</from-outcome>
      <to-view-id>/SRMain.jspx</to-view-id>
    </navigation-case>

    <navigation-case>
      <from-outcome>Search</from-outcome>
      <to-view-id>/SRSearch.jspx</to-view-id>
    </navigation-case>

    <navigation-case>
      <from-outcome>Create</from-outcome>
      <to-view-id>/SRCreate.jspx</to-view-id>
    </navigation-case>
  </navigation-rule>
  ...
</faces-config>
```

For information about creating JSF navigation rules and cases, as well as creating navigation components, see Chapter 9, "Adding Page Navigation".

A.10.1.6 Registering Custom Validators and Converters

JSF and ADF Faces standard validators and converters provide common validation checks for numeric ranges and string lengths, and the most common datatype conversions. If you need more complex validation rules and checks, or if you need to convert a component's data to a type other than a standard type, you can create your own custom validator or converter.

The custom validator or converter must implement the `javax.faces.validator.Validator` or `javax.faces.convert.Converter` interface, respectively. To make use of your custom validator or converter in an application, you have to register it in `faces-config.xml` using the `<validator>` or `<converter>` element (see Example A–26). For a custom validator, you can register it under an identifier (ID); for a custom converter you can register it under an ID or a fully qualified class name for a specific datatype.

Example A–26 Registering Custom Validators and Converters in faces-config.xml

```
<faces-config xmlns="http://java.sun.com/JSF/Configuration">
  ...
  <validator>
    <validator-id>oracle.srdemo.core.CreditCard</validator-id>
    <validator-class>oracle.srdemo.core.CreditCardValidator</validator-class>
  </validator>
  <converter>
    <converter-id>oracle.srdemo.core.CreditCard</validator-id>
    <converter-class>oracle.srdemo.core.CreditCardConverter</converter-class>
  </converter>
  ...
</faces-config>
```

A.10.1.7 Registering Managed Beans

In JSF, managed beans are the JavaBeans used to manage data between the web tier and the business tier of the application (similar to a data transfer object). At runtime, whenever the bean is referenced in a page through a value or method binding expression, the JSF implementation instantiates a bean, populates it with any declared, default values, and places it in the managed bean scope as defined in the `faces-config.xml`.

To register a managed bean in `faces-config.xml`, use the `<managed-bean>` element (see Example A–27). You have to specify the following for a managed bean:

- Name—Determines how the bean will be referred to within the application using EL expressions, instead of using the bean's fully qualified class name.
- Class—This is the JavaBean that contains the properties that hold the data, along with the corresponding accessor methods and/or any other methods (such as navigation or validation) used by the bean. This can be an existing class (such as a data transfer class), or it can be a class specific to the page (such as a backing bean).

- Scope—This determines the scope within which the bean is stored. The valid scopes are:
  - `application`—The bean is available for the duration of the web application. This is helpful for global beans such as LDAP directories.
  - `request`—The bean is available from the time it is instantiated until a response is sent back to the client. This is usually the life of the current page.
  - `session`—The bean is available to the client throughout the client's session.
  - `none`—The bean is instantiated each time it is referenced.

Managed properties are any properties of the bean that you would like populated with a value when the bean is instantiated. The set method for each declared property is run once the bean is constructed. To initialize a managed bean's properties with set values, including those for a bean's map or list property, use the `<managed-property>` element. When you configure a managed property for a managed bean, you declare the property name, its class type, and its default value.

Managed beans and managed bean properties can be initialized as lists or maps, provided that the bean or property type is a List or Map, or implements `java.util.Map` or `java.util.List`. The default for the values within a list or map is `java.lang.String`.

Example A–27 Registering Managed Beans in faces-config.xml

```
<faces-config xmlns="http://java.sun.com/JSF/Configuration">
  ...
  <!-- This managed bean uses application scope -->
  <managed-bean>
    <managed-bean-name>resources</managed-bean-name>
    <managed-bean-class>
     oracle.srdemo.view.resources.ResourceAdapter
    </managed-bean-class>
    <managed-bean-scope>application</managed-bean-scope>
  </managed-bean>

  <!-- Page backing beans typically use request scope-->
  <managed-bean>
    <managed-bean-name>backing_SRCreate</managed-bean-name>
    <managed-bean-class>oracle.srdemo.view.backing.SRCreate</managed-bean-class>
    <managed-bean-scope>request</managed-bean-scope>
    <!--oracle-jdev-comment:managed-bean-jsp-link:1app/SRCreate.jspx-->
    <managed-property>
      <property-name>bindings</property-name>
      <value>#{bindings}</value>
    </managed-property>
  </managed-bean>

  <managed-bean>
    <managed-bean-name>backing_SRManage</managed-bean-name>
    <managed-bean-class>
      oracle.srdemo.view.backing.management.SRManage
    </managed-bean-class>
    <managed-bean-scope>request</managed-bean-scope>
    <!--oracle-jdev-comment:managed-bean-jsp-link:1app/management/SRManage.jspx-->
    <managed-property>
      <property-name>bindings</property-name>
      <value>#{bindings}</value>
    </managed-property>
  </managed-bean>
```

```
  <!-- This managed bean uses session scope -->
  <managed-bean>
    <managed-bean-name>userState</managed-bean-name>
    <managed-bean-class>oracle.srdemo.view.UserSystemState</managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
  </managed-bean>
  ...
</faces-config>
```

A.11 adf-faces-config.xml

When you create a JSF application using ADF Faces components, besides configuring elements in `faces-config.xml` you can configure ADF Faces-specific features in the `adf-faces-config.xml` file. The `adf-faces-config.xml` file has a simple XML structure that enables you to define element properties using the JSF expression language (EL) or static values.

In JDeveloper when you insert an ADF Faces component into a JSF page for the first time, a starter `adf-faces-config.xml` file is automatically created for you in the `/WEB-INF` directory of your ViewController project. Example A–28 shows the starter `adf-faces-config.xml` file.

Typically you would want to configure the following in `adf-faces-config.xml`:

- Page accessibility levels
- Skin family
- Time zone
- Enhanced debugging
- Oracle Help for the Web (OHW) URL

Example A–28 Starter adf-faces-config.xml Created by JDeveloper

```
<?xml version="1.0" encoding="windows-1252"?>
<adf-faces-config xmlns="http://xmlns.oracle.com/adf/view/faces/config">

  <skin-family>oracle</skin-family>

</adf-faces-config>
```

A.11.1 Tasks Supported by adf-faces-config.xml

The following JSF tasks are supported by the `adf-faces-config.xml` file.

A.11.1.1 Configuring Accessibility Levels

To define the level of accessibility support in an application, use `<accessibility-mode>`. The supported values are:

- `default`—Output supports accessibility features.
- `inaccessible`—Accessibility-specific constructs are removed to optimize output size.
- `screenReader`—Accessibility-specific constructs are added to improve behavior under a screen reader (but may have a negative affect on other users. For example access keys are not displayed if the accessibility mode is set to screen reader mode).

Example A–29 Configuring an Accessibility Level

```
<!-- Set the accessibility mode to screenReader -->
<accessibility-mode>screenReader</accessibility-mode>
```

A.11.1.2 Configuring Currency Code and Separators for Number Groups and Decimals

To set the currency code to use for formatting currency fields, and define the separator to use for groups of numbers and the decimal point, use the following elements:

- `<currency-code>`—Defines the default ISO 4217 currency code used by `oracle.adf.view.faces.converter.NumberConverter` to format currency fields that do not specify a currency code in their own converter.
- `<number-grouping-separator>`—Defines the separator used for groups of numbers (for example, a comma). ADF Faces automatically derives the separator from the current locale, but you can override this default by specifying a value in this element. If set, this value is used by `oracle.adf.view.faces.converter.NumberConverter` while parsing and formatting.
- `<decimal-separator>`—Defines the separator (e.g., a period or a comma) used for the decimal point. ADF Faces automatically derives the separator from the current locale, but you can override this default by specifying a value in this element. If set, this value is used by `oracle.adf.view.faces.converter.NumberConverter` while parsing and formatting.

Example A–30 Configuring Currency Code and Separators For Numbers and Decimal Point

```
<!-- Set the currency code to US dollars. -->
<currency-code>USD</currency-code>

<!-- Set the number grouping separator to period for German -->
<!-- and comma for all other languages -->
<number-grouping-separator>
 #{view.locale.language=='de' ? '.' : ','}
</number-grouping-separator>

<!-- Set the decimal separator to comma for German -->
<!-- and period for all other languages -->
<decimal-separator>
 #{view.locale.language=='de' ? ',' : '.'}
</decimal-separator>
```

A.11.1.3 Configuring For Enhanced Debugging Output

ADF Faces enhances debugging output when you set `<debug-output>` to "true". The following features are then added to debug output:

- Automatic indenting.
- Comments identifying which component was responsible for a block of HTML.
- Detection of unbalanced elements, repeated use of the same attribute in a single element, or other malformed markup problems.
- Detection of common HTML errors (for example, `<form>` tags inside other `<form>` tags or `<tr>` or `<td>` tags used in illegal locations).

Example A–31 Enabling Enhanced Debugging

```
<!-- Activate the ADF Faces enhanced debugging features -->
<debug-output>true</debug-output>
```

A.11.1.4 Configuring for Client-Side Validation and Conversion

ADF Faces validators and converters support client-side validation and conversion as well as server-side validation and conversion. ADF Faces client-side validators and converters work the same way as the server-side validators and converters, except that JavaScript is used on the client. ADF Faces JavaScript-enabled validators and converters run on the client when the form is submitted; thus errors can be caught without a server round trip. You can, however, turn off client-side conversion and validation in your ADF Faces application by setting `<client-validation-disabled>` to "true".

Example A–32 Turning Off Client-Side Validation and Conversion

```
<!-- Disable client validation -->
<client-validation-disabled>true</client-validation-disabled>
```

A.11.1.5 Configuring the Language Reading Direction

By default, ADF Faces page rendering direction is based on the language being used by the browser. However, you can explicitly set the default page rendering direction in the `<right-to-left>` element by using "true" or "false".

Example A–33 Configuring For Right-to-Left Page Rendering

```
<!-- Render the page right-to-left for Arabic -->
<!-- and left-to-right for all other languages -->
<right-to-left>
 #{view.locale.language=='ar' ? 'true' : 'false'}
</right-to-left>
```

A.11.1.6 Configuring the Skin Family

By default, ADF Faces uses the Oracle `<skin-family>` for all pages. You can change this to specify a custom `<skin-family>`. See also Section A.12.1, "Tasks Supported by adf-faces-skins.xml".

For information about creating custom skins, see Section 14.3, "Using Skins to Change the Look and Feel".

Example A–34 Configuring a Skin to be Used For All Pages

```
<!-- Specify custom skin instead of Oracle skin -->
<skin-family>srdemo<skin-family>
```

A.11.1.7 Configuring the Output Mode

To change the output mode ADF Faces uses, set the `<output-mode>` element, using one of these values:

- `default`—The default page output mode (usually display).
- `printable`—An output mode suitable for printable pages.
- `email`—An output mode suitable for e-mailing a page's content.

Example A–35 Configuring an Output Mode

```
<!-- Set the output mode to printable -->
<output-mode>printable</output-mode>
```

A.11.1.8 Configuring the Number of Active ProcessScope Instances

By default ADF Faces sets the maximum number of active `processScope` instances at 15. Use the `<process-scope-lifetime>` element to change the number. A static value must be used.

Example A–36 Configuring the Number of Active ProcessScope Instances

```
<!-- Set the maximum number of processScope instances to 10 -->
<process-scope-lifetime>10</process-scope-lifetime>
```

A.11.1.9 Configuring the Time Zone and Year Offset

To set the time zone used for processing and displaying dates, and the year offset that should be used for parsing years with only two digits, use the following elements:

- `<time-zone>`—ADF Faces defaults to the time zone used by the client browser. This value is used by `oracle.adf.view.faces.converter.DateTimeConverter` while converting strings to `Date`.
- `<two-digit-year-start>`—Defaults to the year 1950 if no value is set. This value is used by `oracle.adf.view.faces.converter.DateTimeConverter` to convert strings to `Date`.

Example A–37 Configuring the Time Zone and Year Offset

```
<!-- Set the time zone to Pacific Daylight Savings Time -->
<time-zone>PDT</time-zone>

<!-- Set the year offset to 2000 -->
<two-digit-year-start>2000</two-digit-year-start>
```

A.11.1.10 Configuring a Custom Uploaded File Processor

Most applications don't need to replace the default `UploadedFileProcessor` instance provided by ADF Faces, but if your application needs to support uploading of very large files or rely heavily on file uploads, you may wish to replace the default processor with a custom `UploadedFileProcessor` implementation. For example you could improve performance by using an implementation that immediately stores files in their final destination, instead of requiring ADF Faces to handle temporary storage during the request. To replace the default processor, specify a custom implementation using the `<uploaded-file-processor>` element.

Example A–38 Configuring a Custom Uploaded File Processor

```
<!-- Use my UploadFileProcessor class -->
<uploaded-file-processor>
 com.mycompany.faces.myUploadedFileProcessor
</uploaded-file-processor>
```

A.11.1.11 Configuring the Help Site URL

If you use Oracle Help for the Web (OHW) to provide help in your application, you can attach help content to any JSF tag that accepts a URL. Before you can do this, you must configure your help site URL by using the `<oracle-help-servlet-url>` element. ADF Faces supports OHW Version 2.0 as well as earlier versions

Use the `adfFacesContext.helpTopic` EL object to attach help content to the JSF tag. For example:

```
<h:outputLink value="#{adfFacesContext.helpTopic.someTopicID}">
   <h:outputText value="Help!"/>
</h:outputLink>
```

Example A–39 Configuring the Help Site URL

```
<!-- Set the help site URL -->
<oracle-help-servlet-url>mywebsite.com/project_one/help</oracle-help-servlet-url>
```

A.11.1.12 Retrieving Configuration Property Values From adf-faces-config.xml

Once you have configured elements in the `adf-faces-config.xml` file, you can retrieve property values using one of the following approaches:

- Programmatically using the `AdfFacesContext` class.

  The `AdfFacesContext` class is a context class for all per-request and per-webapp information required by ADF Faces. One instance of the `AdfFacesContext` class exists per request. Although it is similar to the JSF `FacesContext` class, the `AdfFacesContext` class does not extend `FacesContext`.

  To retrieve an ADF Faces configuration property programmatically, first call the static `getCurrentInstance()` method to get an instance of the `AdfFacesContext` object, then call the method that retrieves the desired property, as shown in the following example:

```
// Get an instance of the AdfFacesContext object
AdfFacesContext context = AdfFacesContext.getCurrentInstance();

// Get the time-zone property
TimeZone zone = context.getTimeZone();

// Get the right-to-left property
if (context.isRightToLeft())
{
  ...
}
```

For the list of methods to retrieve ADF Faces configuration properties, refer to the Javadoc for `oracle.adf.view.faces.context.AdfFacesContext`.

- Using a JSF EL expression to bind a component attribute value to one of the properties of the ADF Faces implicit object (`adfFacesContext`).

  The `AdfFacesContext` class contains an EL implicit variable, called `adfFacesContext`, that exposes the context object properties for use in JSF EL expressions. Using a JSF EL expression, you can bind a component attribute value to one of the properties of the `adfFacesContext` object. For example in the EL expression below, the `<currency-code>` property is bound to the `currencyCode` attribute value of the JSF `ConvertNumber` component:

```
<af:outputText>
  <f:convertNumber currencyCode="#{adfFacesContext.currencyCode}"/>
</af:outputText>
```

A.12 adf-faces-skins.xml

The `adf-faces-skins.xml` file is optional; you need this file only if you are using a custom skin for your application. To create the file, simply use a text editor; store the file in `/WEB-INF`.

You can specify one or more custom skins in `adf-faces-skins.xml`.

Example A–40 Adf-faces-skins.xml

```
<?xml version="1.0" encoding="windows-1252"?>
<skins xmlns="http://xmlns.oracle.com/adf/view/faces/skin">
  <skin>
    <id>purple.desktop</id>
    <family>purple</family>
    <render-kit-id>oracle.adf.desktop</render-kit-id>
    <style-sheet-name>skins/purple/purpleSkin.css</style-sheet-name>
    <bundle-name>oracle.adfdemo.view.faces.resource.SkinBundle</bundle-name>
  </skin>
</skins>
```

A.12.1 Tasks Supported by adf-faces-skins.xml

The value of `<family>` is what you would specify in `adf-faces-config.xml` for the `<skin-family>` element when you wish to configure your application to use a custom skin. See Section A.11.1.6, "Configuring the Skin Family".

For information about creating custom skins, see Section 14.3, "Using Skins to Change the Look and Feel".

B

Reference ADF Binding Properties

This appendix provides a reference for the properties of the ADF bindings.

B.1 EL Properties of Oracle ADF Bindings

Table B–1 shows the properties that you can use in EL expressions to access values of the ADF binding objects at runtime. The properties appear in alphabetical order.

> **Note:** When you use the EL Expression Builder dialog in JDeveloper, you may see properties listed below the ADF bindings and ADF data variables that do not appear in this appendix. Properties that do not appear in this appendix will become deprecated in a future release. For the full list of deprecated binding properties, please refer to the JDeveloper Release Notes.

Table B–1 EL Properties of Oracle ADF Bindings

Runtime Property	Description	Iterator	Action	Attribute	Button	List	Table	Tree
`actionEnabled`	Use `operationEnabled` instead.	n/a	yes	n/a	n/a	n/a	n/a	n/a
`allRowsInRange`	Returns an array of current set of rows from the associated collection. Calls `getAllRowsInRange()` on the RowSetIterator.	yes	n/a	n/a	n/a	n/a	n/a	n/a
`attributeDef`	Returns the attribute definition for the first attribute to which the binding is associated.	n/a	n/a	yes	yes	yes	n/a	n/a
`attributeDefs`	Returns the attribute definitions for all the attributes to which the binding is associated.	n/a	n/a	yes	yes	yes	n/a	n/a
`attributeValue`	Returns an unformatted and typed (appropriate Java type) value in the current row, for the attribute to which the control binding is bound. Note this property is not visible in the EL expression builder dialog.	n/a	n/a	yes	yes	yes	n/a	n/a

Table B–1 (Cont.) EL Properties of Oracle ADF Bindings

Runtime Property	Description	Iterator	Action	Attribute	Button	List	Table	Tree
`attributeValues`	Returns the value of all the attributes to which the binding is associated in an ordered array.Returns an array of an unformatted and typed (appropriate Java type) values in the current row for all the attributes to which the control binding is bound. Note this property is not visible in the EL expression builder dialog.	n/a	n/a	yes	yes	yes	n/a	n/a
`children`	Returns the child nodes of a tree node binding.	n/a	n/a	n/a	n/a	n/a	n/a	yes
`currentRow`	Returns the current row on an action binding bound to an iterator (for example, built-in navigation actions).	n/a	yes	n/a	n/a	n/a	n/a	n/a
`dataControl`	Returns the iterator's associated data provider.	yes	n/a	n/a	n/a	n/a	n/a	n/a
`displayData`	Returns a list of map elements. Each map entry contains the following elements: ▪ `selected`: A boolean `true` if current entry should be selected. ▪ `index`: The index value of the current entry. ▪ `prompt`: A string value that may be used to render the entry in the UI. ▪ `displayValues`: An ordered list of display attribute values for all display attributes in the list binding. Note this property is not visible in the EL expression builder dialog.	n/a	n/a	n/a	n/a	yes	n/a	n/a
`displayHint`	Returns the display hint for the first attribute to which the binding is associated. The hint identifies whether the attribute should be displayed or not. For more information, see `oracle.jbo.AttributeHints.displayHint`. Note this property is not visible in the EL expression builder dialog.	n/a	n/a	n/a	n/a	yes	n/a	n/a

Table B–1 (Cont.) EL Properties of Oracle ADF Bindings

Runtime Property	Description	Iterator	Action	Attribute	Button	List	Table	Tree
`displayHints`	Returns a list of name-value pairs for UI hints for all display attributes to which the binding is associated. The map contains the following elements: ■ `label`: The label to display for the current attribute. ■ `tooltip`: The tooltip to display for the current attribute. ■ `displayHint`: The display hint for the current attribute. ■ `displayHeight`: The height in lines for the current attribute. ■ `displayWidth`: The width in characters for the current attribute. ■ `controlType`: The control type hint for the current attribute. ■ `format`: The format to be used for the current attribute. Note this property is not visible in the EL expression builder dialog.	n/a	n/a	n/a	yes	yes	n/a	n/a
`enabled`	Use operationEnabled.	n/a	n/a	n/a	n/a	n/a	n/a	n/a
`enabledString`	Returns `disabled` if the action binding is not ready to be invoked. Otherwise, returns `""`.	n/a	yes	n/a	n/a	n/a	n/a	n/a
`error`	Returns any exception that was cached while updating the associated attribute value for a a value binding or when invoking an operation bound by an operation binding.	yes	yes	yes	yes	yes	yes	yes
`estimatedRowCount`	Returns the maximum row count of the rows in the collection with which this iterator binding is associated	yes	n/a	n/a	n/a	n/a	yes	yes
`findMode`	Return `true` if the iterator is currently operating in find mode. Otherwise, returns `false`.	yes	n/a	n/a	n/a	n/a	n/a	n/a
`fullName`	Returns the fully qualified name of the binding object in the Oracle ADF binding context.	yes	yes	yes	yes	yes	yes	yes

Table B–1 (Cont.) EL Properties of Oracle ADF Bindings

Runtime Property	Description	Iterator	Action	Attribute	Button	List	Table	Tree
`inputValue`	Returns the value of the first attribute to which the binding is associated. If the binding was used to set the value on the attribute and the set operation failed, this method returns the invalid value that was being set.	n/a	n/a	yes	yes	yes	yes	yes
`iteratorBinding`	Returns the iterator binding that provides access to the data collection.	n/a	yes	yes	yes	yes	yes	yes
`label`	Returns the label (if supplied by Control Hints) for the first attribute of the binding.	n/a	n/a	yes	yes	yes	n/a	n/a
`labels`	Returns a map of labels (if supplied by Control Hints) keyed by attribute name for all attributes to which the binding is associated. Note this property is not visible in the EL expression builder dialog.	n/a	n/a	yes	yes	yes	yes	n/a
`labelSet`	Returns an ordered set of labels for all the attributes to which the binding is associated. Note this property is not visible in the EL expression builder dialog.	n/a	n/a	yes	yes	yes	yes	n/a
`mandatory`	Returns whether the first attribute to which the binding is associated is required.	n/a	n/a	yes	yes	yes	n/a	n/a
`name`	Returns the name of the binding object in the context of the binding container to which it is registered. Note this property is not visible in the EL expression builder dialog.	yes	yes	yes	yes	yes	yes	yes
`operationEnabled`	Returns `true` or `false` depending on the state of the action binding. For example, the action binding may be enabled (`true`) or disabled (`false`) based on the currency (as determined, for example, when the user clicks the First, Next, Previous, Last navigation buttons.	n/a	yes	n/a	n/a	n/a	n/a	n/a

Table B–1 (Cont.) EL Properties of Oracle ADF Bindings

Runtime Property	Description	Iterator	Action	Attribute	Button	List	Table	Tree
`rangeSet`	Returns a list of map elements over the range of rows from the associated iterator binding. The elements in this list are wrapper objects over the indexed row in the range that restricts access to only the attributes to which the binding is bound. The properties returned on the reference object are: ■ `index` — The range index of the row this reference is pointing to. ■ `key` — The key of the row this reference is pointing to. ■ `keyStr` — The String format of the key of the row this reference is pointing to. ■ `currencyString` — The current indexed row as a String. Returns "*" if the current entry belongs to the current row; otherwise, returns " ". This property is useful in JSP applications to display the current row. ■ `attributeValues` — The array of applicable attribute values from the row. And you may also access an attribute value by name on a range set like `rangeSet.dname` if `dname` is a bound attribute in the range binding.	n/a	n/a	n/a	n/a	n/a	yes	yes
`rangeSize`	Returns the range size of the ADF iterator binding's row set. This allows you to determine the number or data objects to bind from the data source.	yes	n/a	n/a	n/a	n/a	yes	yes
`rangeStart`	Returns the absolute index in a collection of the first row in range. See javadoc for `oracle.jbo.RowSetIterator.getRangeStart()`.	yes	n/a	n/a	n/a	n/a	yes	yes
`result`	Returns the result of a method that is bound and invoked by a method action binding.	n/a	yes	n/a	n/a	n/a	n/a	n/a
`rootNodeBinding`	Returns the root node of a tree binding.	n/a	n/a	n/a	n/a	n/a	n/a	yes

Table B–1 (Cont.) EL Properties of Oracle ADF Bindings

Runtime Property	Description	Iterator	Action	Attribute	Button	List	Table	Tree
`selectedValue`	Returns the value corresponding to the current selected index in the list or button binding.	n/a	n/a	n/a	yes	yes	n/a	n/a
`tooltip`	Returns the tooltip hint for the first attribute to which the binding is associated.	n/a	n/a	yes	yes	yes	n/a	n/a
`updateable`	Returns `true` if the first attribute to which the binding is associated is updateable. Otherwise, returns `false`.	n/a	n/a	yes	yes	yes	n/a	n/a

Index

A

access keys, 4-37
access method, specifying, 19-3
accessibility support levels, 4-36
accessor iterators, 5-15
accessor returns, 5-4
`accessorIterator` element, A-23
`Accessors` element, 8-15
acquiring a unit of work, 3-19
Action Binding Editor, 10-5, 10-15
action bindings
  about, 5-18
  debugging, 16-22
  `disabled` attribute, 5-25, 6-16
  `enabled` property, 5-25, 6-16
  for iterator bindings, 6-4
  for methods, 10-20
  for operations, 6-14
  for page navigation, 9-17
`action` element, A-26
action events, 6-17
action listeners
  in navigation operations, 6-17
  in page navigation components, 9-20, 9-21
action methods, in page navigation components, 9-16
`actionEnabled` binding property, B-1
`actionListener` attribute
  command buttons for methods, 10-7
  navigation operations, 6-17
  See also action listeners
actions
  adding ADF bindings to existing, 13-11
  in page navigation, 9-14
`actions` facet, 11-45
`AdapterDataControl` element, A-8
ADF
  Declarative Development with JavaServer Faces, 1-3
  overview, 1-1
  Supported Business Services Technologies, 1-2
  Supported Technologies, 1-1
    Controller Layer, 1-2
    View Layer, 1-2
ADF binding context. *See* binding context
ADF binding filter. *See* binding filter
ADF bindings. *See* bindings
ADF Command Button. *See* command buttons
ADF Controller library, 5-7
ADF Faces
  accessibility support levels, 4-36
  configuration files, A-4
  converters, 12-17
  dependencies and libraries, 4-13
  enhanced debugging output, 4-36
  file uploading, 11-49
  filter and mapping configuration settings, 4-17
  internationalization, 14-11
  partial page rendering, 11-35
  resource servlet and mapping configuration settings, 4-17
  skins, 14-3
  supported platforms, 4-2
  tag libraries for, 4-13
  validation, 12-3
  validators, 12-6
ADF Faces Cache
  AFC Statistics servlet, 15-7
  logging, 15-7
  types of content to cache, 15-2
  visual diagnostics, 15-8
ADF Faces components
  access keys for, 4-37
  adding bindings to existing, 13-3
  adding to JSF pages, 4-14
  changing appearance, 14-3
  creating
    `commandButton` components, 10-5, 10-7
    `commandButton` components for page navigation, 9-15, 9-17
    `commandButton` components, for navigation operations, 6-13
    `commandLink` components for page navigation, 9-15, 9-17
    `commandLink` components, for navigation operations, 6-13
    `inputText` components, 6-2
    `outputText` components, 6-2
    `selectManyShuttle` components, 11-62
    `selectOneChoice` components, 11-56
    `selectRangeChoiceBar` components, 7-6

setActionListener components, 10-8
table components, 7-3
tableSelectMany components, 7-18
tableSelectOne components, 7-4
tree components, 8-9
treeTable components, 8-15
creating from the Data Control Palette, 5-5
layout and panel components, 4-27
skinning, 14-3
style properties, changing, 14-2
translating, 14-11
See also UI components
ADF Faces Core tag library, 4-13
ADF Faces HTML tag library, 4-13
ADF Faces lifecycle, overriding, 12-25
ADF Form. *See* forms
ADF Input Text. *See* text fields
ADF Input Text with a Label. *See* text fields
ADF Label. *See* text fields
ADF Logger, 16-7
ADF Master Form, Detail Form. *See* master-detail objects
ADF Master Form, Detail Table. *See* master-detail objects
ADF Master Table, Detail Form. *See* master-detail objects
ADF Master Table, Detail Table. *See* master-detail objects
ADF Master Table, Inline Detail Table. *See* master-detail objects
ADF Model layer
exception handling, 12-23
lifecycle, 6-6
validation, 12-7
ADF Model Runtime library, 5-7
ADF Output Text. *See* text fields
ADF Output Text with a Label. *See* text fields
ADF Parameter Form. *See* forms
ADF phase listener, 6-6
creating custom, 12-30
registering in the web.xml file, 5-7
ADF Read-Only Dynamic Table. *See* dynamic tables
ADF Read-Only Form. *See* forms
ADF Read-Only Table. *See* tables
ADF runtime libraries
active versions, 22-19
ADF Controller library, 5-7
ADF Model Runtime library, 5-7
adf-controller.jar file, 5-7
ADFm.jar file, 5-7
deleting, 22-27
in the project properties, 5-7
installing
from JDeveloper, 22-17
manually, 22-24
list of files, 22-24
ADF Table. *See* tables
ADF Tree. *See* tree components
ADF Tree Table. *See* treeTable components
ADFBindingFilter class, 5-10, 5-11
adf-config.xml file, 18-22
ADFContext object, 5-12
adf-controller.jar file, 5-7
ADFDG3002 | Implementing Services with EJB Session Beans, 3-2
ADFDG4003 | Creating a Web Page, 4-9
ADFDG503 | Working with the DataBindings.cpx File, 5-9
adf-faces-config.xml file
about, A-4, A-44
accessibility, A-45
client-side validation, 12-4, A-46
client-validation-disabled element, 12-4
conversion, 12-17
currency, numbers, and decimals, A-45
currency-code element, 14-19
custom upload file processor, A-48
decimal-separator element, 14-19
editing, 4-19
enhanced debugging output, A-46
example of, 4-18
help site URL, A-48
language reading direction, A-46
localization properties, 14-19
number-grouping-separator element, 14-19
output mode, A-47
ProcessScope instances, A-47
retrieving property values, A-48
right-to-left element, 14-19
skin family, A-47
skin-family element, 14-10
supported tasks, A-44
time zone and year offset, A-47
time-zone element, 14-19
adf-faces-skins.xml file, 14-8
about, A-49
bundle-name element, 14-9
family element, 14-9
id element, 14-9
render-kit-id element, 14-9
skins element, 14-9
style-sheet-name element, 14-9
supported tasks, A-49
adfm.jar file, 5-7
adfm.xml element, A-11
AFC Statistics servlet, 15-7
aggregate collection mappings, 3-14
aggregate object mappings, 3-14
allDetailsEnabled attribute, 7-12
allRowsInRange binding property, B-1
amendment methods for TopLink descriptors, 3-10
application templates, 4-3
application view caching, 4-37
Apply Request Values phase, 6-8
attribute bindings
about, 5-17, 6-4
EL expressions for, 6-5
setting ADF authorization grants, 18-24
Attribute element, A-13
AttributeAccessor element, A-14

attributeDef binding property, B-1
attributeDefs binding property, B-1
attributes
about, 5-5
binding to text fields, 6-2
on the Data Control Palette, 5-5
attributeValue binding property, B-1
attributeValues bindings property, B-2
attributeValues element, A-26
AttrNames element, 5-19, 8-15
authentication
enabling ADF Security, 18-9
enabling J2EE security, 18-4
authorization
ADF Security permissions, 18-21
enabling for ADF Security, 18-20
authorizationEnforce property, 18-22
automatic component binding, 4-31, 9-16, 9-17
automatic form submission, 4-36
autoSubmit attribute
for table row selection components, 7-15
use of, 4-36

B

Back button, issues
in forms, 6-17
in page navigation, 9-11
in tables, 7-8
backing beans
ADF data controls and, 4-34
automatic component binding and, 4-31
binding attribute and, 4-31
overriding declarative methods in, 10-10
referencing in JSF pages, 4-30
registering in faces-config.xml, 4-29
uses of, 4-28
using for validation method, 12-11
using in page navigation components, 9-16
Bind Action Property dialog, 9-17, 10-11
Bind Existing ADF Read-Only Table. *See* binding to existing components
Bind Existing ADF Table. *See* binding to existing components
Bind Existing CommandButton. *See* binding to existing components
Bind Existing CommandLink. *See* binding to existing components
Bind Existing Input Text. *See* binding to existing components
Bind Existing Tree. *See* binding to existing components
Bind Validator Property dialog, 12-12
binding attribute, 4-31
binding containers
about, 5-19
accessing from other pages, 5-27
debugging, 16-15
overriding declarative methods, 10-14
runtime usage, 5-19
scope, 5-19
setting ADF authorization grants, 18-23
binding context
about, 5-8
initializing using the ADF binding filter, 5-12
binding context, debugging, 16-11
binding features, 1-7
design-time support, 1-8
Binding Metadata Code Insight, 1-8
Data Control Palette, 1-8
data control wizards, 1-8
Page Definition Editor, 1-8
binding filter, 5-7, 5-10
binding objects
action, 1-6, 5-18
attribute, 1-6, 5-17
defined in page definition files, 5-13
EL properties of, B-1
invokeAction, 5-15
iterator, 1-6, 5-15
list, 5-17
lists, 11-58, 11-61
method action, 5-18
referencing in EL expressions, 5-20
runtime properties, 5-27
scope, 5-19, 5-28
table, 5-17
tree, 5-17
value, 5-17
binding properties
accessing in the Expression Builder, 5-21
configuring, 1-18
EL reference, B-1
hiding and showing groups of components, 1-19
in EL expressions, 5-20
binding to existing components
commandButton components, 13-12
commandLink components, 13-12
inputText components, 13-8
outputText components, 13-7
selection lists, 13-13
table components, 13-9
tree components, 13-15
bindings
action, 6-4
action for methods, 10-20
action for operations, 6-14
adding to existing components, 13-1
adding to UI components, 6-19
attribute, 6-4
changing, 6-20
changing for tables, 7-9
deleting for UI components, 6-19
iterator, about, 6-3
rebinding tables, 7-11
rebinding UI components, 6-20
required objects for ADF, 5-7
table, 7-4
text fields, 6-2
value, 6-4

bindings element, 5-17
bindings variable, 5-20
breakpoints
  types of, 16-8
bundle-name element, 14-9
business services
  web services, 21-1
button element, A-27
buttons, command. *See* command buttons

C

cacheResults attribute, 10-18
caching with ADF Faces Cache, 15-1 to 15-9
change policy for unit of work, 3-22
character encoding, in the ADF binding filter, 5-12
children binding property, B-2
classes
  persistent, 3-8
  representing tables, 3-8
client-side state saving, 4-35
client-side validation
  creating custom JSF, 12-15
  using custom JSF, 12-15
client-validation-disabled element, 12-4
CollectionModel class, 7-5
collections, about, 5-3
columns
  attributes for, 7-6
  column tag, 7-5
command buttons
  adding ADF bindings to existing, 13-11
  binding to backing beans, 9-16
  binding to methods, 10-7
  creating using a method, 10-5
  in navigation operations, 6-13
  in page navigation, 9-15, 9-17
command components
  executing methods with, 10-4
  ID for, 10-7
  passing parameter values with, 10-8
command links
  in page navigation, 9-15, 9-17
  navigation operations, 6-13
  setting current row with, 7-22
commandButton components. *See* command buttons
commandLink components. *See* command links
components. *See* UI components
conditionally displaying components, 10-32
configuration files for JSF
  creating page navigation rules in, 9-2
  editing, 4-6
  starter file in JDeveloper, 4-6
  using multiple, 4-8
  wizard for creating, 4-8
  See also faces-config.xml file
constructors
  about, 10-16
  create methods, about, 5-4
  procedures for using, 10-17
  refresh condition, 10-20
Context class, 6-7
conversion
  about, 12-16
  in an application, 12-2
  lifecycle, 12-2
converters
  ADF Faces, 12-17
  creating custom, 12-20
  creating custom JSF, 12-19
  using, 12-18
.cpx file. *See* DataBindings.cpx file
CreatableTypes element, A-8, A-9
Create Application dialog, 4-3
Create Cascading Style Sheet dialog, 14-6
Create Java Objects from Tables wizard, 3-8
Create JSF JSP wizard, 4-9
Create Managed Bean dialog, 4-29, 10-3
create methods, about, 5-4
Create Web Service Data Control wizard, 21-4
createPageLifecycle method, 12-30
CRUD methods, 3-2
CSS style properties, 14-2
currency-code element, 14-19
current row, setting programmatically, 7-22
currentRow binding property, B-2
CVS
  client, 17-1
  commit comments, 17-2
  preferences, 17-1

D

data binding files
  about, A-3
data control files
  about, 3-27, A-3
Data Control Palette
  about, 5-2
  accessor returns, 5-4
  attributes, 5-5
  constructors, 5-4
  context menu, 5-5
  create methods, 5-4
  data control, 5-4
  default UI component features, 5-6
  displaying, 5-2
  icons defined, 5-3
  identifying master-detail objects, 8-2
  method returns, 5-4
  objects created, 5-7
  operations, 5-5
  parameters, 5-5
  using to create UI components, 5-5
data control registry, A-11
Data Control Security wizard, 21-9
data control security, defining for web services, 21-9
data controls
  about, 3-27
  creating, 3-26

displayed on the Data Control Palette, 5-3
from sessions.xml file, 19-4
from TopLink Map, 19-5
from web services, 21-4
using to create UI components, 5-1
data variable, 5-27
database tables, representing as classes, 3-8
DataBindings.cpx file, A-3
about, 5-7, 5-9, A-18
changing a page definition filename, 5-13
dataControlUsages element, A-19
elements, A-19
elements, defined, 5-10
PageDefinitionUsages element, A-19
pageMap element, A-19
runtime usage, 5-19
sample, A-20
syntax, A-18
dataControl binding property, B-2
DataControlConfigs element, A-7
DataControls.dcx file, 3-27, A-7
elements, A-8
sample file, A-9
syntax, A-7
dataControlUsages element, 5-10, A-19
data-sources.xml file, not including in deployment, 22-28
debugging
ADF binding context, 16-11
ADF Model in JDeveloper, 16-6
ADF Model layer, 16-10
binding container, 16-15
runtime errors, 16-4
decimal-separator element, 14-19
default page navigation cases, 9-6
defName attribute, 8-15
deleteAllObjects method, 3-22
deleteObject method, 3-22
deletes, performing before inserts, 19-2
deploying ADF applications, 22-1
for testing purposes, 22-1
from JDeveloper, 22-8
overview, 22-1
steps for, 22-2
techniques for, 22-7
to EAR file, 22-8
to JBoss, 22-11
to Oracle Application Server, 22-9
to Tomcat, 22-15
to WebLogic, 22-13
to WebSphere, 22-14
troubleshooting, 22-27
using Ant, 22-8
using scripts, 22-8
deploying SRDemo application, 22-8
descriptors, TopLink, 3-8
detailStamp facet
about, 7-11
DisclosureEvent event, 7-13
used to display inline detail table, 8-18
using, 7-12
dialog navigation rules, 11-22
digital signatures, setting for web services, 21-12
direct collection mappings, 3-14
direct mappings
creating, 3-12, 3-15
types of, 3-12
direct-to-field mappings, 3-12
disabled attribute
about, 5-25, 6-16
enabled property, 5-25, 6-16
DisclosureAllEvent event, 8-18
DisclosureEvent event
detailStamp facet, 7-13
in tree components, 8-15
in treeTable components, 8-18
disclosureListener attribute
detailStamp facet, 7-13
in tree components, 8-15
in treeTable components, 8-18
displayData binding property, B-2
displayHint binding property, B-2
displayHints binding property, B-3
dropdown lists
adding ADF bindings to existing, 13-13
dynamic list of values, 11-59
fixed list of values, 11-56
List Binding Editor, 11-57, 11-59, 13-13
list binding object, 11-58, 11-61
dynamic menus. *See* menus
dynamic outcomes. *See* outcomes
dynamic tables, 7-3

E

Edit Form Fields dialog, 6-10
Edit Table Columns dialog, 7-3, 7-9, 7-16
ejb-definition element, A-9
EL expressions
accessing results in a managed bean, 10-12
accessing security properties, 18-25
ADF binding properties, 5-20, 5-27
binding attributes with, 6-5
binding object properties reference, B-1
bindings variable, 5-20
creating, 5-19, 5-20
data variable, 5-27
editing, 5-20
examples of ADF binding expressions, 5-23
Expression Builder, using to create, 5-21
navigation operations, 6-16
referencing binding objects, 5-20
syntax for ADF binding expressions, 5-20
tracing in JDeveloper, 16-27
using to bind to ADF data control objects, 5-19
embedded OC4J server, deploying for testing, 22-2
enabled binding property, B-3
enabled property, 5-25, 6-16
enabledString binding property, B-3
entity definition file, 3-27

error binding property, B-3
error messages
  about, 12-22
  disabling client-side, 12-23
  displaying server-side, 12-23
  parameters in, 12-5
estimatedRowCount binding property, B-3
evaluating page navigation rules, 9-10
events
  action, 6-17
  DisclosureAllEvent event, 8-18
  DisclosureEvent event, 7-13, 8-15, 8-18
  FocusEvent event, 8-17
  LaunchEvent event, 11-25
  PrepareRender event, 6-9
  RangeChangeEvent event, 7-8
  ReturnEvent event, 11-27
  SelectionEvent event, 7-17
examples
  stored procedure call, 3-25
exception handling
  about, 12-23
  changing, 12-24
  custom handler, 12-24
  customizing the lifecycle, 12-25
  single page, overriding for, 12-31
executables element, 5-15
execute property, 6-16
existing components. *See* binding to existing components
Expression Builder
  about, 5-21
  about the object list, 5-23
  icons used, 5-23
  using, 5-22
expression language. *See* EL expressions
external transaction controller, 3-18

F

faces-config.oxd_faces file, 4-11, 9-8
faces-config.xml file
  about, A-4, A-38
  configuring for ADF Faces, 4-18
  converters, 12-20
  custom validators and converters, A-42
  editing, 4-6
  example of, 4-6
  from-action element, 9-3
  from-outcome element, 9-3
  from-view-id element, 9-3
  locales, 14-18
  managed bean configuration, 10-2, 10-4
  managed beans, A-42
  managed-bean element, 10-4
  message resource bundle, A-40
  navigation rules and classes, A-40
  navigation-case element, 9-3
  page navigation rule elements, 9-3
  phase listener
    default, 6-6
    registering new, 12-30
  phase listener for ADF Binding, A-39
  redirect element, 9-3
  render kit for ADF Faces, A-39
  required elements for ADF, 5-7
  supported locales, A-40
  supported tasks, A-38
  to-view-id element, 9-3
  using to define page navigation rules, 9-2
  validation, 12-15
  wizard for creating, 4-8
FacesContext class, 6-6
FacesServlet class, 6-6
facets
  about, 6-12
  actions facet, 11-45
  adding or removing, 4-25
  detailStamp facet, 7-11, 8-18
  footer facet, 6-12
  in panelPage components, 4-24
  in tree components, 8-13
  in tree table components, 8-17
  location facet, 11-45
  menu, 11-2
  nodeStamp facet, 8-13, 8-17, 11-14
  pathStamp facet, 8-17
  selection facet, 7-14
family element, 14-9
file dependencies, A-2
file uploading
  context parameters for, 11-55
  custom file processor for, 11-55
  disk space and memory for, 11-55
  inputFile component, 11-51
  storage location for files, 11-50
  supporting, 11-50
filter mappings, 5-11
filter-class element, 5-11
filter-name element, 5-11
findAll() method, 3-7
findByPrimaryKey, 3-17
finding objects by primary key, 3-17
findMode binding property, B-3
FocusEvent event, 8-17
FocusListener listener, 8-17
footer facet, 6-12
foreign-key relationships, 8-1
forms
  adding UI components, 6-19
  changing order of UI components, 6-19
  creating basic, 6-9
  creating input, 10-16
  creating search, 10-23
  creating using parameters, 10-14, 10-15
  deleting UI components, 6-19
  footer facet, 6-12
  modifying the default, 6-18
  navigation operations, 6-13

parameter forms, 10-20
using the Data Control Palette to create, 6-11
using to display master-detail objects, 8-4
widgets for basic, 6-10
from-action element, 9-3
from-outcome element, 9-3
from-view-id element, 9-3
fullName binding property, B-3

G

getAsObject method, 12-19
getAsString method, 12-19
getBundle method, 14-14, 14-15
getClientConversion() method, 12-21
getClientScript() method, 12-15, 12-21
getClientValidation() method, 12-15
getContents() method, 14-17
global buttons, 11-2
global page navigation rules, 9-6
graph element, A-27

H

hierarchical menus. *See* menus

I

id attribute, 5-13
id element, 14-9
immediate attribute, 6-8
Initialize Context phase, 6-7, 6-8
init-param element, 5-12
inline tables, 8-18
input forms
about, 10-16
constructors versus custom methods, 10-16
input parameter in stored procedures, 3-23, 3-25
inputFile components, use of, 11-51
inputText components. *See* text fields; forms
inputValue binding property, B-4
Insert ActionListener dialog, 10-8
Insert SelectManyShuttle dialog, 11-69
inserts, performing after deletes, 19-2
internationalization
about, 14-11
procedures for, 14-14
See also localizing
Invoke Application phase, 6-8
invokeAction bindings, 5-15
invokeAction element, A-25
isExpanded method, 8-15
iterator bindings
about, 5-15, 6-3
method, 6-3
range, 5-16
rangeSize attribute, 6-15
setting ADF authorization grants, 18-23
tables, 7-4
iterator element, A-25
iteratorBinding binding property, B-4
iterators
about, 5-15
accessor, 5-15
method, 5-15
RowSetIterator object, 5-16
variable, 5-15
iterBinding attribute, 5-18

J

j_security_check login method, 18-12
j2ee-logging.xml file, A-37
Java classes
creating from database tables, 3-8
mapping to database tables, 3-11
javax.faces.CONFIG_FILES context parameter, 4-8
javax.faces.STATE_SAVING_METHOD context parameter, 4-35
jazn-data.xml file, 22-28
jazn-data.xml, with web services, 21-10
JBoss, deploying applications to, 22-11
JSF
basic application architecture, 1-5
features offered, 1-4
JSF components
creating from the Data Control Palette, 5-5
See also UI components
JSF Configuration Editor
launching, 4-6
using to create managed beans, 10-2
using to define page navigation rules, 9-2, 9-5, 9-7
JSF Core tag library, 4-13
JSF HTML tag library, 4-13
JSF lifecycle, with ADF, 6-6
JSF Navigation Case, 9-5
JSF navigation diagrams
deleting JSF pages on, 4-12
faces-config.oxd_faces files for, 4-11
opening, 4-10
renaming JSF pages on, 4-12
using to define page navigation rules, 9-2
See also JSF Navigation Modeler
JSF Navigation Modeler
deleting pages, 9-13
refreshing, 9-13
using to define page navigation rules, 9-3, 9-13
JSF Page Flow & Configuration wizard, 4-8
JSF pages
automatic component binding in, 4-31
backing beans for, 4-28
creating from the JSF navigation diagram, 4-10
designing for ADF bindings, 13-2
editing, 4-14
effects of name change, 5-9
example in XML, 4-11
inserting UI components, 4-14
laying out, 4-22 to 4-28
loading a resource bundle, 14-17
referencing backing beans in, 4-30

ways to create, 4-9
JSF servlet and mapping configuration settings, 4-5
JSF tag libraries, 4-13
JSP documents, 4-9
JTA unit of work, 3-18
JUnit extension, installing, 2-10

K

keystores
creating, 21-6
exporting public key, 21-8
requesting certificates, 21-7
using with web services data controls, 21-6

L

`label` attribute, 6-19, 7-3
`label` binding property, B-4
`label` property, table columns, 7-5
`labels` binding property, B-4
labels, changing for UI components, 6-19
`labelSet` binding property, B-4
`LaunchEvent` event, 11-25
`LaunchListener` listener, 11-29
libraries
ADF Controller, 5-7
ADF Model Runtime, 5-7
ADF runtime, 5-7
`adf-controller.jar` file, 5-7
`adfm.jar`, 5-7
lifecycle
error handling, 12-2
JSF and ADF, 6-6
phases in an ADF application, 6-7
lifecycle phases.*See names of individual phases*
links, command. *See* command links
List Binding Editor, 11-57, 11-59, 13-13
list bindings, 5-17
list components, creating, 11-56
`list` element, A-28
list of values
adding ADF bindings to existing, 13-13
dynamic list, 11-59
fixed list, 11-56
List Binding Editor, 11-57, 11-59, 13-13
list binding object, 11-58, 11-61
listeners
`DisclosureListener` listener, 8-15
`FocusListener` listener, 8-17
`LaunchListener` listener, 11-29
`ReturnListener` listener, 11-27
`ListResourceBundle`
about, 14-13
creating, 14-16
`loadBundle` tag
about, 14-12
using, 14-17
local interface, 3-2
locales, registering, 14-18
localizing
about, 14-11
`ListResourceBundle`, 14-13
property files, requirements, 14-13
See also internationalization
`location` facet, 11-45
logging, A-37
changing logging level for ADF packages, A-37
log file location, A-38
redirecting output, A-37
login page, 18-12
logout page, 18-17

M

managed beans
accessing EL expression results, 10-12
automatic component binding and, 4-32
chaining, 11-11
compatible scopes, 11-11
configuring for menus, 11-4, 11-9, 11-11
configuring for process trains, 11-41, 11-42, 11-44
configuring in `faces-config.xml`, 4-29, 10-2
creation at runtime, 10-2
managed properties in, 4-33
multiple page usage, 10-3
multiple pages, 10-8
overriding declarative methods in, 10-10
scope for backing beans with method overrides, 10-12
scope types, 4-29
setting parameters on, 10-9
storing information on, 10-2, 10-33
using in page navigation components, 9-16
validation method, 12-12
value binding expressions for chaining, 11-7
`managed-bean` element, 10-4
`mandatory` binding property, B-4
`mandatory` property, 12-5, 12-8
many-to-many mappings, 3-14
mappings
ADF binding filter, 5-11
direct mappings, 3-12
object-relational, 3-12
object-relational mappings, 3-14
relational, 3-12
relationship mappings, 3-14
mappings, TopLink, 3-11
Master Form, Detail Form. *See* master-detail objects
Master Form, Detail Table. *See* master-detail objects
Master Table, Detail Form. *See* master-detail objects
Master Table, Detail Table. *See* master-detail objects
Master Table, Inline Detail. *See* master-detail objects
`masterBinding` attribute, 8-7
master-detail objects
about, 8-1
displaying in
`detailStamp` facet, 8-18
forms, 8-4
separate pages, 8-8

tables, 8-4
tree components, 8-9
treeTable components, 8-15
example of, 8-4
in the Data Control Palette, 8-2
managing row currency, 8-8
managing synchronization of data, 8-8
masterBinding attribute, 8-7
MasterTable, Inline Detail widget, 8-19
RowSetIterator objects, 8-8
treeTable components, 8-16
widgets, 8-5
Master-Details widgets, 8-5
MenuModel class, 11-3
menus
components for, 11-13
facets for, 11-2
managed beans for, 11-4, 11-9, 11-11
menu model creation, 11-3 to 11-11
menu tree model, 11-8
navigation rules for, 11-16
nodeStamp facet, 11-14
startDepth attribute, 11-15
ViewIdPropertyMenuModel instance, 11-9
ways to create, 11-2
messages tag, 12-8, 12-22
messages, error
about, 12-22
disabling client-side, 12-23
displaying server-side, 12-23
parameters in, 12-5
Metadata Commit phase, 6-8
metadata files
about, A-1
metadata in TopLink, 3-8
method accessing, 19-3
method action binding objects, 5-18
method action bindings
setting ADF authorization grants, 18-24
method iterator bindings, 6-3
method iterators, 5-15
method returns, 5-4
methodAction element, A-28
methodIterator element, A-25
methods
adding logic to, 10-10, 10-13
binding to command components, 10-4, 10-5
create, 5-4
creating input forms with, 10-20
creating search forms with, 10-23
CRUD, 3-2
findAll(), 3-7
in page navigation components, 9-14
overriding declarative, 10-10, 10-13
populating parameters at runtime, 10-22, 10-26
providing parameters when binding, 10-5
.mwp file, 3-8

N

name binding property, B-4
NamedData element
about, 10-20
creating, 10-15
navigation menus. *See* menus
navigation modeler. *See* JSF Navigation Modeler
navigation operations
action events, 6-17
Back button, issues, 6-17
EL expressions for, 6-16
inserting, 6-13
types, 6-16
navigation rules, page
about, 9-2
conflicts, 9-12
creating, 9-2
default cases, 9-6
deleting, 9-13
dialogs, for launching, 11-22
evaluating at runtime, 9-10
examples of, 9-9
global, 9-2, 9-6
in multiple configuration files, 9-11
menus, for, 11-16
overlapping, 9-11
pattern-based, 9-2
splitting, 9-12
navigation, page
about, 9-1
binding to a backing bean, 9-17
binding to a data control method, 9-17
default cases, 9-20
dialogs, for launching, 11-22
dynamic outcomes, 9-1, 9-16
from-action element, 9-3
from-outcome element, 9-3
from-view-id element, 9-3
global rules, 9-2
menus, for, 11-16
navigation-case element, 9-3
NavigationHandler handler, 9-10
navigation-rule element, 9-3
pattern-based rules, 9-2
redirect element, 9-3
rules
about, 9-2
conflicts, 9-12
creating, 9-2
default cases, 9-6
deleting, 9-13
evaluating at runtime, 9-10
examples of, 9-9
global, 9-2, 9-6
in multiple configuration files, 9-11
overlapping, 9-11
pattern-based, 9-2, 9-5
using the JSF Configuration Editor, 9-7
using the JSF Navigation Modeler, 9-3

static outcomes, 9-1, 9-14
to-view-id element, 9-3
using action listeners, 9-21
using outcomes, 9-1
using the JSF Navigation Modeler, 9-13
navigation, range
forms, 6-13
row attribute, 7-7
tables, 7-6
navigation-case element, 9-3
navigation-case Properties dialog, 9-5
NavigationHandler handler, 9-10
navigation-rule element, 9-3
nested unit of work, 3-23
nodeDefinition element, 8-15
nodeStamp facet, 8-13, 8-17, 11-14
number-grouping-separator element, 14-19

O

object hierarchies, 8-1
object type mappings, 3-12
object-relational mappings, 3-12, 3-14
one-to-many mappings, 3-14
one-to-one mappings, 3-14
operationEnabled binding property, B-4
operations
accessing from the Data Control Palette, 5-5
action events for navigation, 6-17
EL expressions for navigation, 6-16
in page navigation components, 9-14
navigation, 6-13, 6-14, 6-16
Oracle ADF
debugging the Model layer, 16-10
file syntax, A-4
security features, 18-1
Oracle Wallet, 21-6
oracle.adf.view.faces.CHECK_FILE_MODIFICATION context parameter, 4-35
oracle.adf.view.faces.DEBUG_JAVASCRIPT context parameter, 4-36
oracle.adf.view.faces.USE_APPLICATION_VIEW_CACHE context parameter, 4-37
outcomes
dynamic, 9-1, 9-16
page navigation, 9-1
static, 9-1, 9-14
output parameter
in stored procedures, 3-24
length in stored procedures, 3-25
output parameter event in stored procedures, 3-25
output parameter in stored procedures, 3-25
outputText components. *See* text fields; forms

P

page controllers, 4-7
page definition files
about, 5-7, 5-12, A-3, A-21
action bindings, 5-18
at runtime, 5-19
attribute bindings, 5-17
AttrNames element, 5-19
binding containers, 5-19
binding objects, 5-13
bindings element, 5-17
creating, 5-12
effects of name change, 5-9, 5-12
elements, 5-13, A-22
executables element, 5-15
id attribute, 5-13
invokeAction bindings, 5-15
iterator bindings, 5-15
iterBinding attribute, 5-18
list bindings, 5-17
location, 5-12
mapped in the DataBindings.cpx file, 5-13
masterBinding attribute, 8-7
method bindings, 5-18
naming, 5-12
nodeDefinition element, 8-15
parameters, 5-14
parameters element, 5-14
rangeSize attribute, 5-17
refresh attribute, 5-16, 5-17
refreshCondition attribute, 5-16
renaming, A-21
sample, A-30
syntax, A-21
table bindings, 5-17
tree bindings, 5-17, 8-13
value bindings, 5-17
page element, A-26
page layouts, 4-22 to 4-28
page navigation. *See* navigation, page
Page Properties dialog, 4-31
pageDefinition element, A-22
pageDefinitionUsages element, 5-10, A-19
PageDef.xml file. *See* page definition files
pageMap element, 5-10, A-19
panelButtonBar components, 6-16
panelPage components
facets in, 4-24
inserting into pages, 4-14
uses of, 4-13
parallel unit of work, 3-23
parameter element, A-23
parameter forms
at runtime, 10-22, 10-26
creating, 10-20
parameter methods
creating forms with, 10-14
creating input forms with, 10-20
creating search forms with, 10-23
passing values to, 10-8

ParameterInfo element, A-14
parameters
  accessing values, 10-5, 10-6
  Apply Request Values phase, 6-8
  bindings for, 10-6
  creating forms with, 10-14, 10-15
  creating input forms with, 10-20
  creating search forms with, 10-23
  defined in page definition file, 5-14
  for messages, 12-5
  for methods, 10-6
  NamedData element, 10-6
  on the Data Control Palette, 5-5
  passing values for, 10-8
  Prepare Model phase, 6-7, 6-9
  providing for methods, 10-5
  setting for methods, 10-5
  setting on setActionListener components, 10-8
parameters element, 5-14
param-name element, 5-11
partial page rendering
  attributes for enabling, 11-36
  autoSubmit attribute and, 4-36
  command components and, 11-37
  panelPartialRoot tag and, 4-21
partialSubmit attribute, 11-24
partialTriggers attribute, 11-36
pathStamp facet, 8-17
pattern attribute, 12-18
pattern-based page navigation rules, 9-5
permission grants for ADF Security, 18-21
persistent classes, 3-8
phase listeners
  creating custom, 12-30
  registering in the web.xml file, 5-7
phase-listener element, 5-7
popup dialogs
  closing and returning from, 11-25
  components with built-in support for, 11-35
  conditions for supporting, 11-22
  creating, 11-22 to 11-28
  launch event, 11-25
  launch listener, 11-29
  launching from command components, 11-23
  navigation rules for launching, 11-22
  passing values into, 11-29
  return event and return listener, 11-27
  return value handling, 11-28
  tasks for supporting, 11-22
postback property, using in refreshCondition attribute, 6-9
PPR. *See* partial page rendering
Prepare Model phase
  about, 6-7
  when navigating, 6-9
Prepare Render phase
  about, 6-9
  exception handling, 12-24
  overriding, 12-25
prepareModel method, 12-25
PrepareRender event, 6-9
primary key, finding objects by, 3-17
process trains
  page access control, 11-44
  processChoiceBar components, binding to train models, 11-45
  processTrain components, binding to train models, 11-45
  train model creation, 11-41 to 11-44
Process Validations phase, 6-8
processChoiceBar components, binding to train models, 11-45
ProcessMenuModel class, 11-40, 11-42
processScope scope, 11-29
processTrain components, binding to train models, 11-45
Project Properties dialog, 4-8
projects
  creating from WAR files, 4-3
  dependencies on, 4-8
  JSF technology in, 4-4
  properties of, 4-4, 4-11
  renaming, 4-3
  view or user interface, 4-7
property files
  creating for resource bundles, 14-15
  requirements for resource bundles, 14-13
pseudo class, 14-6
  creating, 14-7
  referencing, 14-7
pseudo elements, 14-6

Q

queries, TopLink Named Query, 3-17
query sequencing, 3-26

R

range navigation. *See* navigation, range
range, iterator, 5-16
RangeChangeEvent event, 7-8
rangeSize attribute, 5-17, 6-15, 7-7
rangeSize binding property, B-5
rangeStart binding property, B-5
ReadOnlyCollection.xml file, 3-27, A-17
ReadOnlySingleValue.xml file, 3-27, A-17
rebinding
  input components, 6-20
  tables, 7-11
redirect element, 9-3
refresh attribute, 5-16, 5-17
  about, 6-7
refreshCondition attribute, 5-16
  about, 6-7
registerObject method, 3-19
relational mappings, 3-12
relationship mappings, 3-14
remote interface, 3-2

Render Response phase, 6-9
render-kit-id element, 14-9
reportErrors method, 12-24 to 12-25, 12-31
reportException method, 12-24, 12-31
required attribute
table row selection components, 7-15
validation, 12-4, 12-8
resetRange binding property, B-5
resource bundles
creating as a property file, 14-15
creating as Java classes, 14-16
for skins
creating, 14-8
registering, 14-8
using, 14-6
ListResourceBundle, 14-13
loading onto a JSF page, 14-17
property files, 14-13
property files versus Java classes, 14-13
Restore View phase, 6-7
restoreState method, 12-14
result binding property, B-5
returnActionListener tag, 11-25
ReturnEvent event, 11-27
ReturnListener listener, 11-27
right-to-left element, 14-19
rootNodeBinding binding property, B-5
row, 5-3
row currency
on master-detail objects, 8-8
setting programmatically, 7-22
rows attribute
about, 7-7
binding to rangeSize attribute, 7-7
first attribute, 7-7
setting, 7-7
rowsByDepth attribute, 8-18
rowset, 5-3
RowSetIterator objects
about, 5-16
scope, 5-19
used to manage master-detail objects, 8-8
rules, page navigation
about, 9-2
conflicts, 9-12
creating, 9-2
default cases, 9-6
deleting, 9-13
dialogs, for launching, 11-22
evaluating at runtime, 9-10
examples of, 9-9
global, 9-2, 9-6
in multiple configuration files, 9-11
menus, for, 11-16
pattern-based, 9-2

S

SAML assertion tokens, for web services, 21-11
saveState method, 12-14
scope, binding containers and objects, 5-19
search forms
about, 10-23
conditionally displaying results table, 10-32
creating, 10-23
security
for ADF web applications, 18-1
for web service data controls, 21-5
selectBooleanCheckbox components, in a table, 7-10
selectedValue binding property, B-6
selection facet, 7-14, 7-17
selection list components
adding ADF bindings to existing, 13-13
creating, 11-56
SelectionEvent event, 7-17
selectionState attribute, 7-17
selectItems tag, 11-58
selectManyShuttle components, creating, 11-62
selectOneChoice components
creating, 11-56
in a table, 7-10
selectOneListbox components, in a table, 7-10
selectOneRadio components, in a table, 7-10
selectors, 14-5
selectRangeChoiceBar components
about, 7-6
at runtime, 7-7
RangeChangeEvent event, 7-8
sequencing
displaying before commit operations, 19-3
stored procedures, 3-26
TopLink queries, 3-26
serialized object mappings, 3-12
services, building with TopLink, 3-1
servlet context parameter, 5-11
session beans, 3-2
bean class example, 3-4
creating, 3-2
EJB version, 3-2
interface example, 3-3
interface types, 3-2
updating, 3-8
session facades, 3-2
generating methods, 3-2
updating, 3-8
sessions, specifying sessions.xml file, 19-2
sessions.xml file
specifying, 19-2
using with data control, 19-4
setActionListener components
about, 10-8
search pages, conditionally displaying results, 10-33
setting, 10-8
setActionListener tag, 11-19

setCurrentRowWithKey operation
setting programmatically, 7-22
setting programmatically for tableSelectMany components, 7-20
setSubmittedValue method, 6-8
skin element, 14-9
skin-family element, 14-10
skins
about, 14-3
alias pseudo class, 14-6
configuring an application to use, 14-10
creating, 14-6
creating a resource bundle for, 14-8
icons, for, 14-7
minimal, 14-3
Oracle, 14-3
pseudo class
about, 14-6
creating, 14-7
pseudo elements, 14-6
registering, 14-8
resource bundles
about, 14-6
creating, 14-8
registering, 14-8
rtl pseudo class, 14-7
selectors, 14-5
simple, 14-4
using, 14-5
SOAP, and web services, 21-2
Source element, A-8
SRDemo application
functionality, 2-11
installing, 2-4
JUnit tests, running, 2-9
overview, 2-1
requirements, 2-2
schema, 2-2
stack trace, reporting information in, 16-10
standalone OC4J, deploying for testing, 22-2
startDepth attribute, 11-15
state saving, 4-35
StateHolder interface, 12-14
static outcomes. *See* outcomes
stored procedure call example, 3-24
stored procedures
input parameter, 3-23, 3-25
output parameter, 3-24, 3-25
output parameter event, 3-25
output parameter length, 3-25
sequencing with, 3-26
TopLink with, 3-23
using StoredFunctionCall method, 3-25
StoredFunctionCall method, 3-25
StoredProcedureCall method, 3-23
structure definition
Attribute element, A-13
AttributeAccessor element, A-14
entity example, A-16
ParameterInfo element, A-14
schema, A-11
session bean example, A-14
structure definition file, 3-27
style properties, changing, 14-2
StyleClass dialog, 14-2
style-sheet-name element, 14-9
submitForm method, 12-4

T

table binding objects, 5-17
table element, A-29
table tag, 7-5
tables
about, 7-2
adding ADF bindings to existing, 13-8
attributes for, 7-6
Back button, using, 7-8
bindings for, 7-4
changing default, 7-9
conditionally displaying on search page, 10-32
creating, 7-2
detailStamp facet
about, 7-11
using, 7-12
dynamic tables, 7-3
master table with inline detail table, 8-18
master-detail objects, displaying in, 8-4
read-only, 7-3
rebinding, 7-11
rows attribute, setting, 7-7
selection facet, 7-14
selectRangeChoiceBar components, 7-6
about, 7-6
table tag, 7-5
var attribute, 7-5
versus forms, 7-2
widgets for, 7-3
tableSelectMany components
about, 7-14
autoSubmit attribute, 7-15
required attribute, 7-15
text attribute, 7-15
using, 7-18
tableSelectOne components
about, 7-14
adding to a table, 7-4, 7-10
autoSubmit attribute, 7-15
required attribute, 7-15
text attribute, 7-15
using, 7-16
tag libraries for JSF and ADF Faces, 4-13
text attribute, 7-15

text fields
adding ADF bindings to existing, 13-7
binding, 6-2
creating for attributes, 6-2
input text widgets, 6-2
label widgets, 6-2
output text widgets, 6-2
using the Data Control Palette to create, 6-3
time-zone element, 14-19
token validation
forms, 6-17
setting, 6-17
tables, 7-8
Tomcat, deploying applications to, 22-15
tooltip binding property, B-6
TopLink
building services, 3-1
descriptors, 3-8
TopLink map
about, 3-8
in Mapping Editor, 3-12
using multiple, 19-5
TopLink mappings. *See* mappings
TopLink Named Query, 3-17
to-view-id element, 9-3
transactions
overview, 3-18
see also unit of work
transformation mappings, 3-12
Tree Binding Editor, 8-10, 8-19
tree components
about, 8-9
Accessors element, 8-15
adding ADF bindings to existing, 13-14
AttrNames element, 8-15
binding objects created for, 8-13
defName attribute, 8-15
DisclosureEvent event, 8-15
disclosureListener attribute, 8-15
example of, 8-9
facet tag, 8-13
FocusEvent event, 8-17
FocusListener listener, 8-17
isExpanded method, 8-15
nodeDefinition tag, 8-15
nodeStamp facet, 8-13
Tree Binding Editor, 8-10
treeModel property, 8-13
using to display master-detail objects, 8-9
var attribute, 8-13
tree element, A-29
TreeModel class, 8-17
treeModel property, 8-13, 8-17
treeState attribute, 8-18
treeTable components
about, 8-15
Accessors element, 8-15
adding ADF bindings to existing, 13-14
AttrNames element, 8-15
creating from Data Control Palette, 8-16
defName attribute, 8-15
DisclosureAllEvent event, 8-18
DisclosureEvent event, 8-18
disclosureListener attribute, 8-18
displaying master-detail objects, 8-15
example of, 8-15
facet tag, 8-17
nodeStamp facet, 8-17
pathStamp facet, 8-17
rowsByDepth attribute, 8-18
TreeModel class, 8-17
treeModel property, 8-17
treeState attribute, 8-18
var attribute, 8-17
type conversion mappings, 3-12

U

UDDI, and web services, 21-2
UI components
adding ADF bindings to existing, 13-3
adding binding for, 6-19
adding to a form, 6-19
binding instances of, 4-31
changing labels for, 6-19
changing the display order on forms, 6-19
conditionally displaying, 10-32
creating with the Data Control Palette, 5-1, 5-5, 5-6
default ADF features, 5-6
deleting bindings for, 6-19
deleting from a form, 6-19
editing for tables, 7-9
inserting into JSF pages, 4-14
modifying, 6-18
rebinding, 6-18, 6-20
skins, 14-3
style properties, 14-2
See also ADF Faces components
See also JSF components
unit of work
about, 3-18
acquiring, 3-19
change policy, 3-22
creating, 3-19
creating objects, 3-19
deleting objects, 3-22
deleting objects from database, 3-22
example, 3-20
lifecycle, 3-20
modifying objects, 3-21
nested, 3-23
parallel, 3-23
Update Model Values phase, 6-8
updateable binding property, B-6
UpdateableCollection.xml file, 3-27, A-17
UpdateableSingleValue.xml file, A-17
username token authentication, for web services, 21-11

V

`validate` method, 12-10, 12-14, 12-16
Validate Model Updates phase, 6-8
validation
  ADF Faces, 12-3
  ADF Faces attributes, 12-4
  ADF Faces validators, 12-6
  ADF Model validation rules
    adding, 12-7
    types of, 12-7
  client-side custom JSF validators
    creating, 12-15
    using, 12-15
  custom JSF validators
    about, 12-11
    creating, 12-13
  debugging in ADF Model layer, 16-25
  lifecycle, 12-2
  method, overriding, 12-12
  parameters in messages, 12-5
  `required` attribute, 12-4
  runtime, 12-10
  working in an application, 12-2
Validation Rules Editor dialog, 12-8
`Validator` interface, 12-12
`validator` tag, 12-8
validators
  ADF Faces, 12-6
  custom JSF, creating, 12-14
value bindings
  about, 5-17, 6-4
  changing, 6-20
  table, 7-4
`var` attribute
  tables, 7-5
  tree tables, 8-17
  trees, 8-13
variable iterators
  about, 5-15
  using, 10-21
variable one-to-one mappings, 3-14
`variableIterator` element, A-26
variables
  about, 10-21
  at runtime, 10-22, 10-26
  input forms, 10-21
versioning
  committing ADF work to CVS, 17-2
  developer-level activities, 17-4
  name consistency, 17-2
  team-level activities, 17-3
view caching, 4-37
`view.PageDefs` package, 5-12

W

web configuration files, A-4
web pages. *See* JSF pages
web services
  about, 21-1
  authentication, 21-13
  creating data controls, 21-4
  defining data control security, 21-9
  encrypting and decrypting, 21-13
  JAZN, 21-10
  keystores, 21-6, 21-13
  SAML assertion tokens, 21-11
  securing data controls, 21-5
  setting authentication, 21-9
  setting digital signatures, 21-12
  SOAP, 21-2
  testing authentication, 21-10
  UDDI, 21-2
  username token authentication, 21-11
  WSDL, 21-2
  WS-Security, 21-5
  X509 authentication, 21-11
WebLogic, deploying applications to, 22-13
WebSphere
  configuring to run ADF applications, 22-20
  deploying applications to, 22-14
`web.xml` file, 5-7, A-4, A-31
  ADF filter mappings, 5-11
  ADF model binding, A-35
  application view caching, A-34
  configuring for ADF Faces, 4-17
  debugging, A-34
  editing, 4-6
  example of, 4-5
  JSF parameters, A-36
  registering the ADF binding filter, 5-10
  saving state, A-33
  servlet context parameter, defining, 5-11
  tasks supported by, A-33
  uploading, A-35
WSDL, and web services, 21-2
WS-Security, about, 21-5

X

X509 authentication, for web services, 21-11